Mobile Applications
CSCI 448
Lecture 19

Databases and the Room Library

Learning Outcomes For Today

- Discuss the role of the Database, DAO, Repository
- Discuss the purpose of adding Room annotations
- Implement a database via the Room library
- Define Singleton & Façade Design Patterns

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

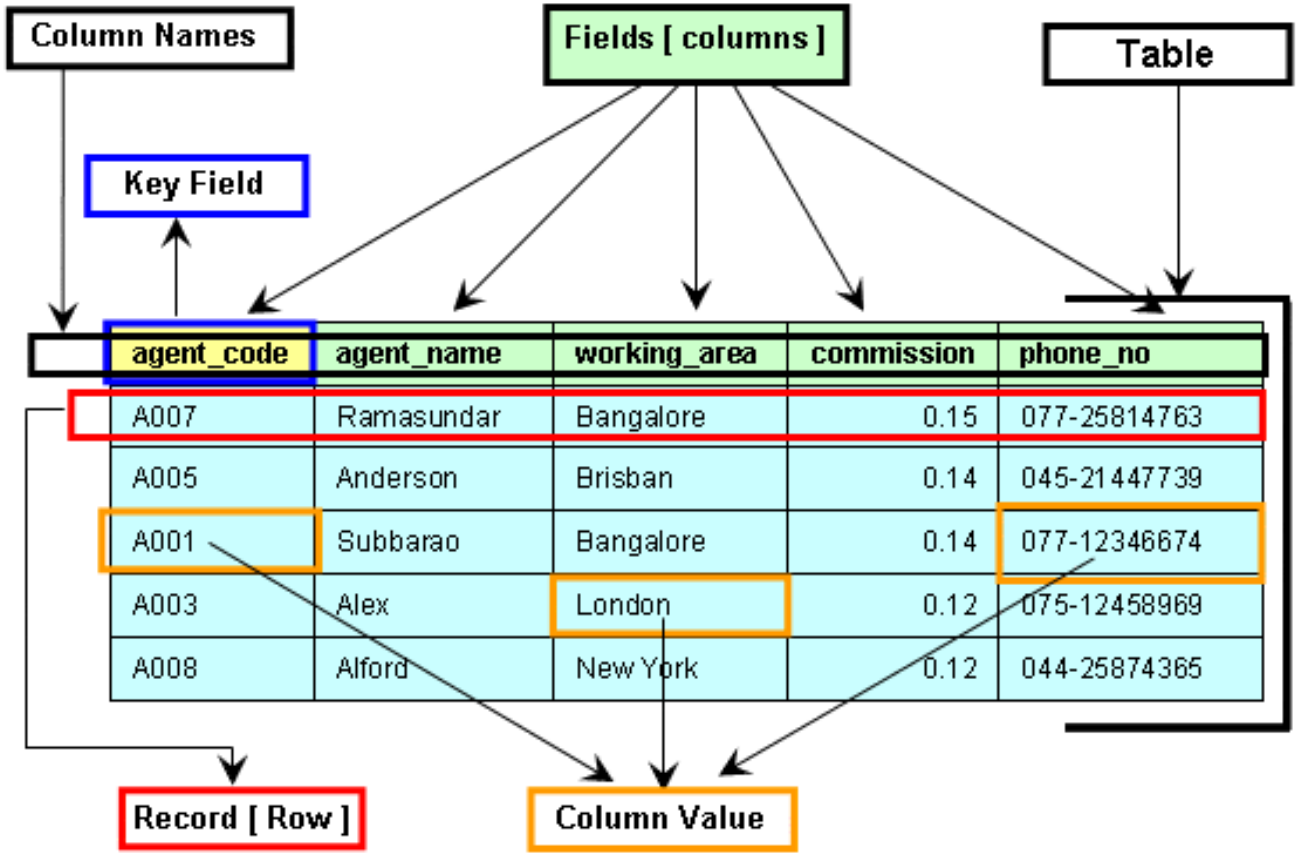
On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

SQLite Databases

- Relational Database Management System
  - Not client-server but embedded in application
  - Weakly typed
  - Database kept on file on device

Databases



https://www.w3resource.com/sql/sql-basic/the-components-of-a-table.php

Table Creation

- ```
  CREATE TABLE character(
      id INT PRIMARY KEY NOT NULL,
      name TEXT NOT NULL,
      health INT
  )
  ```
  ```

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

CRUD Interface

- 4 operations
  - Create
  - Read
  - Update
  - Delete

Create

- ```
  INSERT INTO character
  (id, name, health)
  VALUES
  (1, "Fred Lightfoot", 100)
  ```
  ```

Read

- ```
  SELECT *
  FROM character
  ```

- ```
  SELECT name
  FROM character
  WHERE health < 50
  ```

Update

- ```
  UPDATE character
  SET health = 200
  WHERE name = "Fred Lightfoot"
  ```
  ```

Delete

- ```
  DELETE FROM character
  WHERE health = 0
  ```
  ```

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

Room

- Jetpack Architecture Component!

- Contains
  - API
  - Annotations
  - Compiler

Room

- Jetpack Architecture Component!

- Contains
  - API – classes to extend to define database
  - Annotations
  - Compiler

Room

- Jetpack Architecture Component!

- Contains
  - API – classes to extend to define database
  - Annotations – specify classes to comprise the database
  - Compiler

Room

- Jetpack Architecture Component!

- Contains
  - API – classes to extend to define database
  - Annotations – specify classes to comprise the database
  - Compiler – processes the annotations to make the database

Android Dependencies

- Inside of app/build.gradle
  - Add the Kotlin Annotation Processor Tool plugin
  - Add the Room dependencies
    - Note: current version is 2.5.0

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

Steps to Create Database

1. Annotate data classes to denote tables

2. Create the database class

1. Annotate Data Classes

```
data class Crime(
      val id: UUID,
      var title: String,
      var date: Date,
      var isSolved: Boolean
)
```

1. Annotate Data Classes

```
@Entity(tableName = "crimes")
data class Crime(
      @PrimaryKey
      val id: UUID,
      var title: String,
      var date: Date,
      var isSolved: Boolean
)
```

1. Annotate Data Classes

```
@Entity(tableName = "crimes")
data class Crime(
      @PrimaryKey
      val id: UUID,
      var title: String,
      @ColumnInfo(name = "crimeDate")
      var date: Date,
      var isSolved: Boolean
)
```

2. Create Database I

- Declare the database

```
abstract class CrimeDatabase : RoomDatabase() {



}
```

2. Create Database II

- Associate tables with the database

```
@Database(entities=[Crime::class], version=1)
abstract class CrimeDatabase : RoomDatabase() {




}
```

2. Create Database III

- How to create the database?

```
@Database(entities=[Crime::class], version=1)
abstract class CrimeDatabase : RoomDatabase() {




}
```

2. Create Database IV

- Follows Singleton Design Pattern

```
@Database(entities=[Crime::class], version=1)
abstract class CrimeDatabase : RoomDatabase() {
  companion object {
    @Volatile private var INSTANCE: CrimeDatabase? = null
    fun getInstance(context: Context): CrimeDatabase {
      synchronized(this) {
        var instance = INSTANCE
        if(instance == null) {
          instance = Room.databaseBuilder(context, CrimeDatabase::class.java,
                                          "crime-database").build()
          INSTANCE = instance
        }
        return instance
      }
    }
  }
}
```

Android Design Patterns

- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

DAO

- Data Access Object
  - Defines CRUD interface to work with the database

```
@Dao
interface CrimeDao {

}
```

DAO implements CRUD

- DAO Create

```
@Dao
interface CrimeDao {
  @Insert
  fun addCrime(crime: Crime)









}
```

DAO implements CRUD

- DAO Read

```
@Dao
interface CrimeDao {
  @Insert
  fun addCrime(crime: Crime)
  @Query("SELECT * FROM crime")
  fun getCrimes(): List<Crime>
  @Query("SELECT * FROM crime WHERE id=(:id)")
  fun getCrime(id: UUID): Crime?




}
```

DAO implements CRUD

- DAO Update

```
@Dao
interface CrimeDao {
  @Insert
  fun addCrime(crime: Crime)
  @Query("SELECT * FROM crime")
  fun getCrimes(): List<Crime>
  @Query("SELECT * FROM crime WHERE id=(:id)")
  fun getCrime(id: UUID): Crime?
  @Update
  fun updateCrime(crime: Crime)



}
```

DAO implements CRUD

- DAO Delete

```
@Dao
interface CrimeDao {
  @Insert
  fun addCrime(crime: Crime)
  @Query("SELECT * FROM crime")
  fun getCrimes(): List<Crime>
  @Query("SELECT * FROM crime WHERE id=(:id)")
  fun getCrime(id: UUID): Crime?
  @Update
  fun updateCrime(crime: Crime)
  @Delete
  fun deleteCrime(crime: Crime)


}
```

Register DAO with Database

```
@Database(entities=[Crime::class], version=1)
abstract class CrimeDatabase : RoomDatabase() {

  ...
  abstract val crimeDao: CrimeDao
}
```

Design Pattern #8: Façade

- Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.
- Participants:
  - `Façade`: Knows which subsystem classes are responsible for a request, delegates client requests to appropriate subsystem object
  - `Subsystem Classes`: Implement subsystem functionality, handle work assigned by Façade object, have no knowledge of the Façade

Android Design Patterns

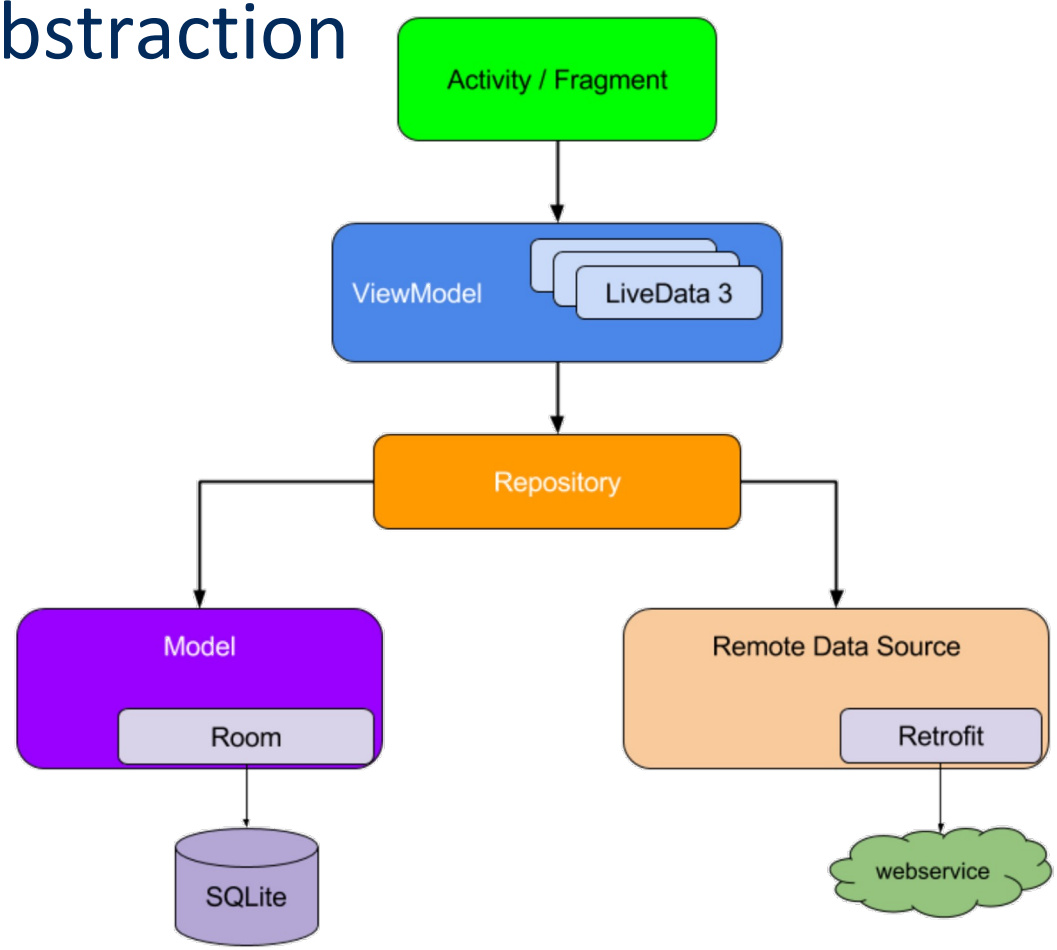
- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State, Flow
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model
  8. Façade – DAO

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

Access Data via Repository

- Adds a layer of abstraction



Create Repository I

- Create the class

```
class CrimeRepository                                                    {




}
```

Create Repository II

- Follow Singleton Design Pattern again

```
class CrimeRepository private constructor(                                    ) {
  companion object {
    @Volatile private var INSTANCE: CrimeRepository? = null
    fun getInstance(context: Context): CrimeRepository {
      synchronized(this) {
        var instance = INSTANCE
        if(instance == null) {

          instance = CrimeRepository(                    )
          INSTANCE = instance
        }
        return instance
      }
    }
  }
}
```

Create Repository III

- Connect to Database via DAO

```
class CrimeRepository private constructor(private val crimeDao: CrimeDao) {
  companion object {
    @Volatile private var INSTANCE: CrimeRepository? = null
    fun getInstance(context: Context): CrimeRepository {
      synchronized(this) {
        var instance = INSTANCE
        if(instance == null) {
          val database = CrimeDatabase.getInstance(context)
          instance = CrimeRepository(database.crimeDao)
          INSTANCE = instance
        }
        return instance
      }
    }
  }
}
```

Create Repository IV

- Expose CRUD interface – Repo also follows ? Pattern

```
class CrimeRepository private constructor(private val crimeDao: CrimeDao) {
  companion object {
    ...
  }

  fun addCrime(crime: Crime) {

      crimeDao.addCrime(crime)

  }
  fun getCrimes(): List<Crime> = crimeDao.getCrimes()
  fun getCrime(id: UUID): Crime? = crimeDao.getCrime(id)
  fun updateCrime(crime: Crime) {

      crimeDao.updateCrime(crime)

  }
  // deleteCrime() too
}
```

Android Design Patterns

- Behavioral Patterns
  1. Command – UI Event Handling
  2. Observer – State, Flow
  3. Template Method - IScreenSpec
- Creational Patterns
  4. Builder – Compose NavGraph
  5. Factory – ViewModelFactory
  6. Singleton – ViewModelProvider, Repository, Room Database
- Structural Patterns
  7. Decorator – View Model
  8. Façade – DAO, Repository

Access Repository via ViewModel

- ViewModel loads data from repository

```
class CrimeListViewModel(private val crimeRepo: CrimeRepository) : ViewModel() {
  val crimes = crimeRepo.getCrimes()
}
```

Access Repository via ViewModel

- ViewModel loads data from repository

```
class CrimeListViewModel(private val crimeRepo: CrimeRepository) : ViewModel() {
  val crimes = crimeRepo.getCrimes()
}
```

- ViewModelFactory responsible for getting the Repository instance

```
class CrimeListViewModelFactory(private val context: Context)
        : ViewModelProvider.Factory {
  override fun <T : ViewModel> create(modelClass: Class<T>): T {
    return modelClass.getConstructor(CrimeRepository::class.java)
        .newInstance(CrimeRepository.getInstance(context))
  }
}
```

Abstraction Review

- Database is a Singleton
  - Created via Room.databaseBuilder()
- Repository is a Singleton and requires DAO
  - Gets Database instance
  - Accesses data from Database via DAO
- ViewModel requires Repository
  - Accesses data from Repository
- ViewModelFactory requires context
  - Gets Repository instance & creates ViewModel
- View requires ViewModel
  - Provides context to ViewModelFactory

Get the data!

- // somewhere in View

  crimeListViewModel.getCrimes()

Get the data!

- // somewhere in View

  crimeListViewModel.getCrimes()

  java.lang.IllegalStateException: Cannot access database on the main thread since it may potentially lock the UI for a long period of time.

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

Enter: `Flow` or `suspend`

- Wait for updates from the database

- Or note that this function blocks
  - And thus can only be called from another coroutine

Enter: `Flow` or `suspend`

- Update DAO to return `Flow` for Read ops
  - Or mark as suspending function

```
@Dao
interface CrimeDao {
  @Insert
  suspend fun addCrime(crime: Crime)
  @Query("SELECT * FROM crime")
  fun getCrimes(): Flow<List<Crime>>
  @Query("SELECT * FROM crime WHERE id=(:id)")
  suspend fun getCrime(id: UUID): Crime?
  @Update
  suspend fun updateCrime(crime: Crime)
  @Delete
  suspend fun deleteCrime(crime: Crime)


}
```

Update Repository

- Update Repo to return **`Flow`** or **`suspend`** for Read ops

```
class CrimeRepository
    private constructor(private val crimeDao: CrimeDao) {

  ...
  fun addCrime(crime: Crime) {

      crimeDao.addCrime(crime)

  }
  fun getCrimes(): Flow<List<Crime>> = crimeDao.getCrimes()
  suspend fun getCrime(id: UUID): Crime? = crimeDao.getCrime(id)
  fun updateCrime(crime: Crime) {

      crimeDao.updateCrime(crime)

  }
  // deleteCrime() too
}
```

Update Repository

- Update Repo to use coroutine for CUD ops

```
class CrimeRepository
    private constructor(private val crimeDao: CrimeDao
                        private val coroutineScope: CoroutineScope) {
  ...
  fun addCrime(crime: Crime) {
    coroutineScope.launch {
      crimeDao.addCrime(crime)
    }
  }
  fun getCrimes(): Flow<List<Crime>> = crimeDao.getCrimes()
  suspend fun getCrime(id: UUID): Crime? = crimeDao.getCrime(id)
  fun updateCrime(crime: Crime) {
    coroutineScope.launch {
      crimeDao.updateCrime(crime)
    }
  }
  // deleteCrime() too
}
```

Update ViewModel

- Reflect that object returned is a **`StateFlow`** object
  - Start new coroutine to collect any changes to lsit from Repository

```
class CrimeListViewModel(private val crimeRepo: CrimeRepository) : ViewModel() {
  val crimes = crimeRepo.getCrimes()
  private val mCrimes: MutableStateFlow<List<Crime>> = MutableStateFlow(emptyList())
  val crimesListState: StateFlow<List<Crime>>
    get() = mCrimes.asStateFlow()

  init {
    viewModelScope.launch {
      crimeRepo.getCrimes().collect { crimeList ->
        mCrimes.update { crimeList }
      }
    }
  }

}
```

Update View

- View observes **`StateFlow`** waiting for updates to collect

```
object CrimeListScreenSpec : IScreenSpec {

  @Composable
  override fun Content(...) {
    val crimesState = viewModel.crimesListState.collectAsState()
    CrimeListScreen(crimes = crimesState.value)
  }

}
```

On Tap For Today

- SQLite Databases
  - CRUD Interface
- Room Library
  - Database
  - DAO
  - Repository
  - Observing
- Practice

To Do For Next Time

- Exam 1 due Fri Mar 03
- Lab06 due Tue Mar 07
- Lab07 due Fri Mar 10
- Alpha Release due Mon Mar 13 – have NavGraph in place
- A2 due Tue Mar 14
- Lab08 due Fri Mar 17
- Alpha Feedback due Fri Mar 17
- Spring Break !!!