Architectural Components (Kotlin Example)

Components Introduction:

- Room Database - wrapper for SQLite in Android. Provides compile time safety.
  - Entity - turn class in Entity which is a table in the database
  - DAO - data access object talks to the database
- ViewModel - holds and prepares user data for the user interface
  - Allows for configuration changes
- Activity / Fragment - connects to the ViewModel and draws the required data and reporting user interactions to the ViewModel which then loads more data
  - NEVER do any database interactions
- Repository - used as a class which mitigates between Room database and ViewModel. Can connect to different places and act like a GET API
- LiveData - observed by Activity / Fragment and notifies it's change. It is also life cycle aware - won't update when the Activity / Fragment is in the background

Model View View Model

- View - Activity / Fragment
- ViewModel - ViewModel
- Model - Repository and Room

Entity

- Entity - Annotated class that describes a database table when working with Room
  - Annotation which needs to be put above the class declaration which will signal to make a new table (Also makes it an actual table)
- Each property represents a column in the database
- PrimaryKey - annotation which indexes into the database
  - Can generate unique keys by annotating with autoGenerate = true and an id int property
- ColumnInfo - specifies the name of the column in the table if you want it to be different from the name of the member variable

```
@Entity(tableName = "word_table")
data class Word(@PrimaryKey(autoGenerate = true) val id: Int,
```

```
                @ColumnInfo(name = "word") val word: String)
```

Entity Creation

- [ ] Annotate with Entity
- [ ] Give table name
- [ ] Decide a PrimaryKey
- [ ] Create the columns (properties)

DAO

- DAO - Annotated class that describes a database table when working with Room.
  - Annotation which needs to be put above the interface declaration which signals this is a DAO interface
  - In general, one DAO per Entity
  - When you use a DAO you call the methods and Room takes care of the rest
- Insert - annotation above the function declaration which allows for SQLite insert operation
- Delete - annotation above the function declaration which allows for SQLite delete operation
- Update - annotation above the function declaration which allows for SQLite update operation
- Query - annotation above the function declaration which allows for a unique SQLite operation
  - Room can return LiveData which will automatically be observable
- By default all operations need to be launched on a separate thread (Coroutines!)

```
@Dao
interface WordDao {
    @Query("<Custom SQL query>")
    fun getAlphabetizedWords(): LiveData<List<Word>> // Used f
or observing the words

    @Insert
    suspend fun insert(word: Word)

    @Delete
    suspend fun delete(word: Word)
```

```
}
```

DAO Creation

- [ ] Create an interface for your DAO
- [ ] Annotate with Dao
- [ ] Create your queries with proper annotations
- [ ] Ensure they are coroutines when working with Room

Room Database

- Simplifies database work and serves as an access point to the underlying SQLite database
  - Room database uses the DAO to issue queries to the underlying database
- Should be an abstract class following the singleton pattern
  - Annotate with the Database annotation passing in the array of entities and a reference to the DAO which will be doing the work. The array of entities is each table which will be created in your database
- Database - annotation to say your class is the database class. This gives you access to your multiple DAOs
- When creating the singleton pattern a companion object for getting the instance since we need to pass in the context and you can't pass in a parameter into a property's custom getter
  - Because there is multi-threading, we have ensure there is only one instance so we don't open up multiple instances of the database

```
@Database(entities = arrayOf(Word::class), version = 1, export
Schema = false
abstract class WordRoomDatabase : RoomDatabase() {
    abstract fun wordDao(): WordDao // Provide access to each
DAO for each table

    companion object {
        @Volatile // Ensure that its state change is always pr
esent in main memory
        private var instance: WordRoomDatabase? = null

        fun getDatabase(context: Context): WordRoomDatabase {
```

```
            synchronized(this) { // Synchronize access to the
database
                if (instance == null) {
                    instance = Room.databaseBuilder(context.ap
plicationContext,
                        WordRoomDatabase::class.java, "word_da
tabase").build()
                }

                return instance!!
            }
        }

        // Add a callback here and override onOpen depending o
n what you want to
        // happen everytime the database is opened for the fir
st time in a session
        // (Delete everything and start fresh, reload the prev
ious session, etc.
    }
}
```

Database Creation

- [ ] Create an abstract class which extends RoomDatabase
- [ ] Annotate with Database and the tables in the database, version, and exportSchema
- [ ] Provide access through an abstract function to all of you DAOs
- [ ] Create a getInstance which is synchronized for the database

Repository

- A repository class abstracts access to multiple data sources. Provides a clean API for data access to the rest of the application
  - Encouraged for separation from Android architecture components
  - Can access the network or a Room database

```
// We don't need to pass in the whole database in this case be
cause we only need
// access to one DAO for writing and reading methods
class WordRepository(private val wordDao: WordDao) {
    // Observed LiveData will notify observer when the data ha
s changed
    // public visibility so it can be registerd with the ViewM
odel
    val allWords: LiveData<List<Word>> = wordDao.getAlphabetiz
edWords()

    suspend fun insert(word: Word) {
        wordDao.insert(word)
    }
}
```

LiveData

- A data holder class that can be observed. Always holds the latest version of data and notifies the observers when data has changed.
  - Lifecycle aware
  - UI components just observe relevant data and don't stop or resume observation

ViewModel

- Acts as a communication center between the Repository (data) and the UI
  - Advantage: UI doesn't need to care about the origin of the data and the ViewModel lasts over configuration changes
- ViewModel should use LiveData for changeable data that the UI will use or display
  - Put observer on the data instead of constant polling and update only on change
- No database calls are done in the ViewModel this is done in the Repository
- Don't keep references to a context that has a shorter lifecycle than a ViewModel: Activity, Fragment, View

```
class WordViewModel(application: Application) : AndroidViewMod
el(application) {

    private val repository: WordRepository
    val allWords: LiveData<List<Word>> // Cache the list of wo
rds

    init {
        val wordsDao = WordRoomDatabase.getDatabase(applicatio
n).wordDao()

        repository = WordRepository(wordsDao)
        allWords = repository.allWords
    }

    // This is the scope where the suspend functions declared
earlier are launched in
    fun insert(word: Word) = viewModelScope.launch(Dispatcher
s.IO) {
        repository.insert(word)
    }
}
```

ViewModel Creation

- [ ] Pass in the application to the constructor
- [ ] Extend AndroidViewModel
- [ ] Get your repository
- [ ] Provide a method for your UI to interface with the repository launched from a coroutine

Linking It All Up

- Connect UI to the database by adding an observer to the LiveData of words in the ViewModel

    - When the data changes, onChanged is invoked which calls the adapter's setWords method to update the adapter's cached data and refresh the displayed list
- Use ViewModelProvider to associate your ViewModel with your Activity

```
// In your Activity / Fragment
private lateinit var wordViewModel: WordViewModel
// In onCreate
wordViewModel = ViewModelProvider(this).get(WordViewModel::class.java)
wordViewModel.allWords.observe(this, Observer { words ->
    // Update the cached copy of the words in the adapter
    words?.let { adapter.setWords(it) } // it is the words
})
// To insert a new word
wordViewModel.insert(word)
```

Linking It Together Creation

- [ ] In your Activity reference your ViewModel
- [ ] Have the ViewModel get associated with the present Activity
- [ ] Have the LiveData in the ViewModel observe the LifeCycle of the Fragment
- [ ] Describe the action when the LiveData changes

Kotlin Coroutines

- A thread allows you to run code off the main thread, but you might end up with a lot of callbacks which might not scale well
- Kotlin coroutines allow for less callbacks and cleaner code

```
suspend fun networkCall() {
    val res = backgroundThreadResults() // Should be suspend fun too

    textView.text = res.name
}
```

- A 'suspend fun' in Kotlin signifies that the method is a suspended method and should only be called from a coroutine scope

- There are three coroutine scopes: Main, IO, Default
  - Main - runs on the main Android Thread. Only good for interacting with the UI and performing quick work
  - IO - optimized to perform disk or network I/O outside the main thread
  - Default - optimized to perform CPU intensive work outside the main / thread

```
// With the viewModelScope extension function you can launch a coroutine easier
fun insert(word: Word) = viewModelScope.launch(Dispatchers.IO)
{
    repository.insert(word)
}
// Dispatchers.IO, Dispatchers.Main, Dispatchers.Default
```