Android Template Code

Jacob Conner

October 7, 2021

Contents

1 Gradle

This section examines the *build.gradle* files in the Project and App folders and the project *Build.settings* files.

1.1 ProjectBuildGradle

```
1  // Top-level build file where you can add configuration options common
       to all sub-projects/modules.
2  buildscript {
3      ext {
4          compose_version = '1.0.1'
5      }
6      repositories {
7          google()
8          mavenCentral()
9      }
10     dependencies {
11         classpath "com.android.tools.build:gradle:7.0.2"
12         classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:1.5.21"
13
14         // NOTE: Do not place your application dependencies here; they
       belong
15         // in the individual module build.gradle files
16     }
17 }
18
19 task clean(type: Delete) {
20     delete rootProject.buildDir
21 }
```

Listing 1: Project build.gradle file

All Android projects consist of at least two *build.gradle* files, a project level *build.gradle* file and an application level *build.gradle* file. The project level *build.gradle* file is shown in Listing 1. Generally this *build.gradle* file contains a `buildscript` tag that has a number of subsections. The section mostly likely to be editted is the `ext` section. In this section gradle environment variables can be defined particularly to keep track of versions of various dependencies. In this example the `compose_version` variable is defined with `compose_version = '1.0.1'`. Dependencies can be defined in this section but generally should be defined using the application level *build.gradle* file.

1.2 AppBuildGradle

```
plugins {
    id 'com.android.application'
    id 'kotlin-android'
    id 'kotlin-kapt'
    id("org.jetbrains.dokka") version "1.4.0"
}

android {
    compileSdk 31

    defaultConfig {
        applicationId "com.example.roomandapi"
        minSdk 21
        targetSdk 31
        versionCode 1
        versionName "1.0"

        testInstrumentationRunner "androidx.test.runner.
    AndroidJUnitRunner"
        vectorDrawables {
            useSupportLibrary true
        }
    }

    buildTypes {
        release {
            minifyEnabled false
            proguardFiles getDefaultProguardFile('proguard-android-
    optimize.txt'), 'proguard-rules.pro'
        }
    }
    compileOptions {
        sourceCompatibility JavaVersion.VERSION_1_8
        targetCompatibility JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = '1.8'
        useIR = true
    }
    buildFeatures {
        compose true
    }
    composeOptions {
        kotlinCompilerExtensionVersion compose_version
        kotlinCompilerVersion '1.5.21'
    }
```

```
    packagingOptions {
        resources {
            excludes += '/META-INF/{AL2.0,LGPL2.1}'
        }
    }
}

tasks.dokkaHtml.configure {
    outputDirectory.set(file("../documentation/html"))
}

dependencies {

    implementation 'androidx.core:core-ktx:1.6.0'
    implementation 'androidx.appcompat:appcompat:1.3.1'
    implementation 'com.google.android.material:material:1.4.0'
    implementation "androidx.compose.ui:ui:$compose_version"
    implementation "androidx.compose.material:material:$compose_version
    "
    implementation "androidx.compose.ui:ui-tooling-preview:
    $compose_version"
    implementation 'androidx.lifecycle:lifecycle-runtime-ktx:2.3.1'
    implementation 'androidx.activity:activity-compose:1.3.1'
    testImplementation 'junit:junit:4.+'
    androidTestImplementation 'androidx.test.ext:junit:1.1.3'
    androidTestImplementation 'androidx.test.espresso:espresso-core
    :3.4.0'
    androidTestImplementation "androidx.compose.ui:ui-test-junit4:
    $compose_version"
    debugImplementation "androidx.compose.ui:ui-tooling:
    $compose_version"

    //Navigation()
    implementation "androidx.navigation:navigation-compose:2.4.0-
    alpha04"

    def room_version = "2.3.0"
    implementation "androidx.room:room-runtime:$room_version"
    annotationProcessor "androidx.room:room-compiler:$room_version"
    implementation "androidx.room:room-ktx:2.3.0"
    kapt "androidx.room:room-compiler:2.3.0"

    implementation "androidx.compose.runtime:runtime-livedata:
    $compose_version"
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.3.1"

    implementation("io.coil-kt:coil-compose:1.3.1")
```

```
85
86      implementation "androidx.constraintlayout:constraintlayout-compose
        :1.0.0-beta02"
87
88      //Compose Material
89      //implementation 'com.google.android.material:material:1.4.0'
90      implementation "io.github.vanpra.compose-material-dialogs:datetime
        :0.6.0"
91      implementation "io.github.vanpra.compose-material-dialogs:core
        :0.6.0"
92      //API handling
93    //  implementation "org.jetbrains.anko:anko-common:0.10.8"
94      implementation 'com.google.code.gson:gson:2.8.7'
95      implementation 'com.squareup.retrofit2:retrofit:2.9.0'
96      implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
97
98  }
```

Listing 2: App build.gradle file

This app-level *build.gradle* file is used to define the various plugins, application dependencies and gradle scripts pertinent to the application. In the example file in Listing 2 the file starts with plugins section. The plugins section consists of four plugins.:

- com.android.application
- kotlin-android
- Kapt plugin - kotlin-kapt
- Dokka plugin - org.jetbrains.dokka

`com.android.application` and `kotlin-android` are all included in the application by default. This example has added the plugin `kotlin-kapt` to enable the program to load the room compiler dependency using the kapt command. The dokka plugin needed to create documentation using KDoc is added with the `id("org.jetbrains.dokka") version "1.4.0"`. Various gradle tasks should be added in the application *build.gradle* file.

```
tasks.dokkaHtml.configure {
    outputDirectory.set(file("../documentation/html"))
}
```

Listing 3: Task to generate Dokka documentation

The task to generate dokka documentation is defined using the code in Listing 3 . Gradle documentation can be created using the terminal with the command `gradlew dokkaHtml`. Dokka will then parse all KDoc comment lines in Kotlin classes and generate the documentation in the build/dokka folder.

Dependencies are stored in the dependencies section of the *build.gradle* file. When a new empty compose application is created, a number of compose dependencies are included as well as various testing frameworks. The Jetpack compose navigation library and the jetpack compose lifecycle libraries are not included by default but are present in the example app *build.gradle* file. The project also makes use of RoomDatabase library so those libraries are included. In order to allow for the downloading and display of images from the internet the `io.coil-kt:coil-compose` library is included.

2 AndroidManifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.roomandapi">
    <uses-permission android:name="android.permission.INTERNET" />
    <uses-permission android:name="android.permission.
   ACCESS_NETWORK_STATE"/>
    <application
        android:usesCleartextTraffic="true"
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.RoomAndApi">
        <activity
            android:name=".MainActivity"
            android:exported="true"
            android:label="@string/app_name"
            android:theme="@style/Theme.RoomAndApi.NoActionBar">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.
   LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Listing 4: Android Manifests File

The *android.manifest* file shown in 4 provides information about the various permissions in the application.

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

Listing 5: Line to give permission to allow internet access

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

Listing 6: Line to give permission to allow network access

The most common reason to edit this file is when an application needs to have access to a website or other resource. Internet access is enabled with the line shown in Listing 5. Network access is also needed to allow the application to access the web, and this is enabled with the line in Listing 6.

3 RoomDatabase

The RoomDatabase library is an object relational mapping (ORM) library in Android that serves as a layer for SQLite queries [1] .

3.1 Dependencies

The implementations are briefly discussed in Section 2, but a list of the required implementations in the *build.gradle* file are listed below.

- RoomDatabase Libraries
  - implementation ”androidx.room:room-runtime:$room_version”
  - annotationProcessor ”androidx.room:room-compiler:$room_version”
  - implementation ”androidx.room:room-ktx:2.3.0”
  - kapt ”androidx.room:room-compiler:2.3.0”
- Lifecycle Libraries
  - implementation ”androidx.compose.runtime:runtime-livedata:$compose_version”
  - implementation ”androidx.lifecycle:lifecycle-livedata-ktx:2.3.1”

The model view view model approach taken in this document is largely derived from the Make it Easy tutorial [2] [3] [4].

3.2 Entities

The lowest level of RoomDatabase is the entity or data model. An entity is simply a data class that describes the schema for a time using Room annotations. The contents of each entity should correspond to the columns of a single SQLite table.

```
1  package com.example.roomandapi.entity
2
3  import androidx.room.ColumnInfo
4  import androidx.room.Entity
5  import androidx.room.PrimaryKey
6
7  @Entity(tableName = "team")
8  data class TeamMember(
9      @PrimaryKey(autoGenerate = true)
10     @ColumnInfo(name = "id", index=true)
11     var id: Int,
12
13     @ColumnInfo(name = "firstName")
14     var firstName: String,
15
16     @ColumnInfo(name = "lastName")
17     var lastName: String,
18
19     @ColumnInfo(name = "bio")
20     var bio: String,
21
22     @ColumnInfo(name = "imageUrl")
23     var imageUrl: String
24 )
```

Listing 7: Example entity class

In Listing 7, an example entity class `TeamMember` is provided for a table called team that stores information about members of a team. In this class, the first Room notation encountered is the `@Entity`notation. This notation is called before the data class is defined and is used to help the Room library identify where this model's data will reside in the SQLite database. In this case the table is called `Team` but the data class is called `TeamMember`. Next the data class is created using the `data class <nameOfClass>( <contents>)`. It is important to note that a data class in Kotlin does not contain any methods and is simply defined with the parameters using the parentheses that initialize it.

The next annotation used in this example class is `@PrimaryKey`. This annotation is used

to define the column below as a primary key for the table, which means it has to be unique and cannot be null. The notation also has the possible parameter of `(autoGenerate = true)` which advises SQLite to autoincrement that column. The last annotation in the example entity class is `@ColumnInfo` which is used to define the name of the table column using the parameter `name="<nameOfColumn>"`. Once the column name has been defined, a variable in kotlin is created for the model which is tied to the table column set. This process is repeated for all columns needed in the table and each column is separated by commas (”,”).

3.3 DAO

Once an entity has been created, a data abstraction object (DAO) has to be created. This is an interface for the entity that maps SQL queries to various methods that can be called.

```
1  package com.example.roomandapi.dao
2
3  import androidx.lifecycle.LiveData
4  import androidx.room.*
5  import com.example.roomandapi.entity.TeamMember
6
7  @Dao
8  interface TeamMemberDao {
9      @Query("SELECT * FROM team")
10     fun getAllTeamMembers(): LiveData<List<TeamMember>>
11
12     @Query("SELECT * FROM team WHERE id = :id")
13     suspend fun getById(id: Int): TeamMember
14
15     @Query("SELECT * FROM team WHERE firstName = :s1 and lastName = :s2
       ")
16     suspend fun getByName(s1: String, s2: String): TeamMember
17
18     @Insert(onConflict = OnConflictStrategy.REPLACE)
19     suspend fun insert(item: List<TeamMember>)
20
21     @Update
22     suspend fun update(item: TeamMember)
23
24     @Delete
25     suspend fun delete(item: TeamMember)
26
27     @Query("DELETE FROM team")
28     suspend fun deleteAll()
29
30
31 }
```

Listing 8: Example DAO

An example of an DAO interface is shown in Listing 8. In this file, the interface is annotated as a DAO with the `@DAO` annotation. After the interface header is created with `interface <nameOfDAO>` the various SQL methods are defined with an annotation followed by a method. The first type of SQL Query annotation is `@Query\verb. This annotation takes a SQL query as a paramet` parameterized sql queries as shown in the `getById function` and the parameter in the query is used as a parameter in the method tied to that query. Room provides a few SQL query annotations that do not require you to specifically type out the SQL query. The `@Insert` annotation takes a list of entities and inserts them into a database using the method below. The option parameter `onConflict = OnConflictStrategy.REPLACE` allows you to specify the conflict strategy with foreign key constraints when a value is inserted. The `@Update` annotation is used to create an update query where an entity is provided and the values of the entity are used to update the value of the entity in the database. The annotation `@Delete` writes the query to delete the entity in the database that matches the entity provided in the method.

3.4 Repository

```
1  package com.example.roomandapi.repository
2
3  import androidx.lifecycle.LiveData
4  import com.example.roomandapi.dao.TeamMemberDao
5  import com.example.roomandapi.entity.TeamMember
6
7  class TeamMemberRepository (private val teamMemberDao: TeamMemberDao){
8      val readALlData: LiveData<List<TeamMember>> = teamMemberDao.
       getAllTeamMembers()
9
10     suspend fun addTeamMember(item: List<TeamMember>){
11         teamMemberDao.insert(item)
12     }
13
14     suspend fun updateTeamMember(item: TeamMember){
15         teamMemberDao.update(item)
16     }
17
18     suspend fun deleteTeamMember(item: TeamMember){
19         teamMemberDao.delete(item)
20     }
21
22     suspend fun deleteAll(){
23         teamMemberDao.deleteAll()
24     }
25 }
```

Listing 9: Example repository class

In order to simplify the ability to to call the functions in the DAO interface, a repository class is

created.An example repository is shown in Listing 9 This repository class takes the DAO interface as an argument. Class methods for each of the DAO functions. In order to support asynchronous methods since a SQL query may not run instantaneously, each method is marked with suspend. Suspend is part of the coroutines library and is used to force a method to run asynchronously on a coroutine. ”A coroutine is an instance of suspendable computation” that is ”not bound to any particular thread” that can be suspended and resumed on different threads [5].

3.5 Database

```
package com.example.roomandapi.database

import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase
import androidx.room.TypeConverters
import com.example.roomandapi.dao.ProjectDao
import com.example.roomandapi.dao.TeamMemberDao
import com.example.roomandapi.entity.ProjectEntity
import com.example.roomandapi.entity.TeamMember
import com.example.workout_companion.utility.DateTimeConverter

@Database(entities = [
    TeamMember::class,
    ProjectEntity::class
                    ],
    version = 5,
    exportSchema = false)
@TypeConverters(DateTimeConverter::class)
abstract class WCDatabase: RoomDatabase() {
    abstract fun teamMemberDao(): TeamMemberDao
    abstract fun projectDao(): ProjectDao

    companion object{
        @Volatile
        private var INSTANCE: WCDatabase? = null

        fun getInstance(context: Context): WCDatabase {
            val sampleInstance = INSTANCE
            if(sampleInstance != null){
                return sampleInstance
            }
            synchronized(this){
                val instance = Room.databaseBuilder(
                    context.applicationContext,
                    WCDatabase::class.java,
                    "workout_companion_database"
```

```
                ).fallbackToDestructiveMigration().build()

                INSTANCE = instance
                return instance
            }


        }
    }
}
```

Listing 10: Example database class

Listing 10 shows the database class. There should be one database class for each database in the project, but generally one database should suffice for most applications. The database class starts with the `@Database` annotation which abstracts the requirements for any SQLite database using the SQLite connector in a given language. This annotation requires a list of entity classes in the database, a database version, and exportSchema. The database is an abstract class extending the RoomDatabase using the line `abstract class <nameOfDatabase>: RoomDatabase()`. An abstract constructor for each DAO is provided to allow the database to access the sql queries needed by the entity managed by the DAO. The database class has one method, `getInstance` which takes the context of the application. and returns a returns a database object. A database instance is created with the Room.databaseBuilder method and it takes the context of the application provided, the database class and a string with the name of the database. The synchronized method chains fallbackToDestructiveMigration(), which is used for migrations, when they fail and rebuilds the database if the migration fails. The build method is the method that actually builds the database [6].

3.6 Supporting Dates

```
package com.example.workout_companion.utility

import android.os.Build
import androidx.annotation.RequiresApi
import androidx.room.TypeConverter
import java.time.*
import java.time.format.DateTimeFormatter
import java.time.temporal.TemporalQueries.localDate

/***
 * Helper object used to convert Strings to LocalTime objects and vis
   versa
 *
 * Modified from https://medium.com/androiddevelopers/room-time-2
   b4cf9672b98
 */
object DateTimeConverter {
```

```
16      /**
17       * DataTimeFormatter object with the pattern used to format
    LocalDateStrings
18       */
19      @RequiresApi(Build.VERSION_CODES.O)
20      private val formatter = DateTimeFormatter.ISO_LOCAL_DATE
21
22      /**
23       * Method that takes a string, parses it and then returns a
    LocalDate object
24       * @param value, a string
25       * @return LocalDate
26       */
27      @RequiresApi(Build.VERSION_CODES.O)
28      @TypeConverter
29      @JvmStatic
30      fun toLocalDate(value: String?): LocalDate? {
31
32          return value?.let {
33              return LocalDate.parse(value, formatter)
34          }
35      }
36
37      /**
38       * Method that takes a LocalDate object, parses it and then returns
    a String
39       * @param value, a LocalDate object
40       * @return String, date formatted as a String
41       */
42      @RequiresApi(Build.VERSION_CODES.O)
43      @TypeConverter
44      @JvmStatic
45      fun fromLocalDate(date: LocalDate?): String? {
46          return date?.format(formatter)
47      }
48  }
```

Listing 11: Converter Class for supporting Dates

SQLite supports 5 data types.

- NULL
- INTEGER
- REAL
- TEXT

- BLOB

A notable omission in the SQLite database is the absence of a Date type. Dates are supported in a variety of ways. They can be stored as an INTEGER as timestamp with the seconds since January 1st, 1970. Another alternative is it can be stored as TEXT variable using ISO8601 strings such as ”YYYY-MM-DD”. A final option is the Date can be stored as Julian days numbers since November 24th 4714 BC. RoomDatabase does not provide out of the box support for dates likely due to the variety of options to store Dates in SQLite [7]. Instead the developer has to create a helper converter class to convert the type of Date variable used in Kotlin to the type used in the SQLite design [8] [9].

In Listing 11 an example type converter is provided to convert a date string formatted using the ISO86601 LOCAL_DATE string. This class contains a formatter which is used to define how the DateTime string is formatted. There are numerous options for date time string formatting supported in Java and Kotlin [10] [11]. Once the formatting string has been defined, the converter class contains two methods for converting to and from the LocalDate class [12].

```
@TypeConverters(DateTimeConverter::class)
```

Listing 12: Task to generate Dokka documentation

Once the converter class has been created, it needs to be added to the Database class shown in 10.The `@TypeConverters` annotation is used to allow the Database to support those conversions between SQLite types and Kotlin classes. This annotation is added before the database class header and takes the class for the file used as shown in Listing 12

3.7 ViewModel

```
1  package com.example.roomandapi.viewmodel
2
3  import android.app.Application
4  import androidx.lifecycle.*
5  import com.example.roomandapi.database.WCDatabase
6  import com.example.roomandapi.entity.TeamMember
7  import com.example.roomandapi.repository.TeamMemberRepository
8  import kotlinx.coroutines.Dispatchers
9  import kotlinx.coroutines.launch
10 import java.lang.IllegalArgumentException
11
12 class TeamMemberViewModel(application: Application): AndroidViewModel(
       application) {
13     val readAllTeamMembers: LiveData<List<TeamMember>>
14     private val repository: TeamMemberRepository
15
16     init{
```

```
17          val teamMemberDao = WCDatabase.getInstance(application).
    teamMemberDao()
18          repository = TeamMemberRepository(teamMemberDao = teamMemberDao
    )
19          readAllTeamMembers = repository.readALlData
20      }
21
22      fun addTeamMember(item: List<TeamMember>){
23          viewModelScope.launch(Dispatchers.IO){
24              repository.addTeamMember(item = item)
25          }
26      }
27
28      fun updateTeamMember(item: TeamMember){
29          viewModelScope.launch(Dispatchers.IO){
30              repository.updateTeamMember(item = item)
31          }
32      }
33
34      fun deleteTeamMember(item: TeamMember){
35          viewModelScope.launch(Dispatchers.IO){
36              repository.deleteTeamMember(item = item)
37          }
38      }
39
40      fun deleteAllTeamMembers(){
41          viewModelScope.launch(Dispatchers.IO){
42              repository.deleteAll()
43          }
44      }
45  }
46
47  class TeamMemberViewModelFactory(
48      private val application: Application
49  ): ViewModelProvider.Factory{
50      override fun <T: ViewModel?> create(modelClass: Class<T>): T{
51          @Suppress("UNCHECKED_CAST")
52          if(modelClass.isAssignableFrom(TeamMemberViewModel::class.java)
    ){
53              return TeamMemberViewModel(application) as T
54          }
55          throw IllegalArgumentException("Unknown ViewModel class")
56      }
57  }
```

Listing 13: Example ViewModel class

The view model is the class that GUI will interact with to run queries. This final layer of abstraction

The view model, class extends the AndroidViewModel class and takes the argument of application as the context. The init function initializes a DAO with the database class's get instance method using application as context and calls the teamMemberDao function to generate the Dao.

4 Testing Room

In Android projects there are two types of unit tests. There are standard Unit Tests that make use of the JUnit library and run like a normal Java or Kotlin project would. The second type of test is the Android Test which runs a virtualized environment to access the Android Libraries to run the functions that require the Android Framework to run [13]. Room Database is one of those examples.

4.1 Creating Tests

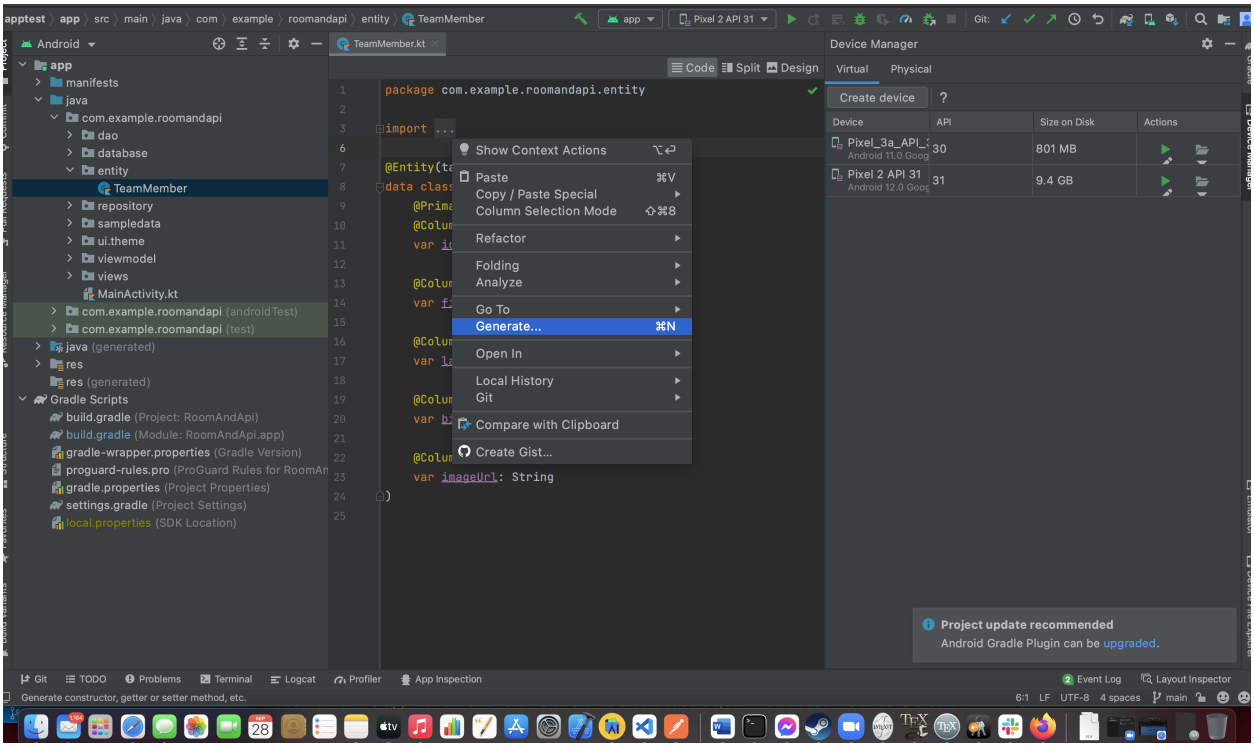

Figure 1: Step 1. Open the file you want to test and right click it.

The process for creating an Android Test with room begins by first navigating to the file that you would like to test. Right click anywhere in the open file and select `Generate...` as shown in Figure 1

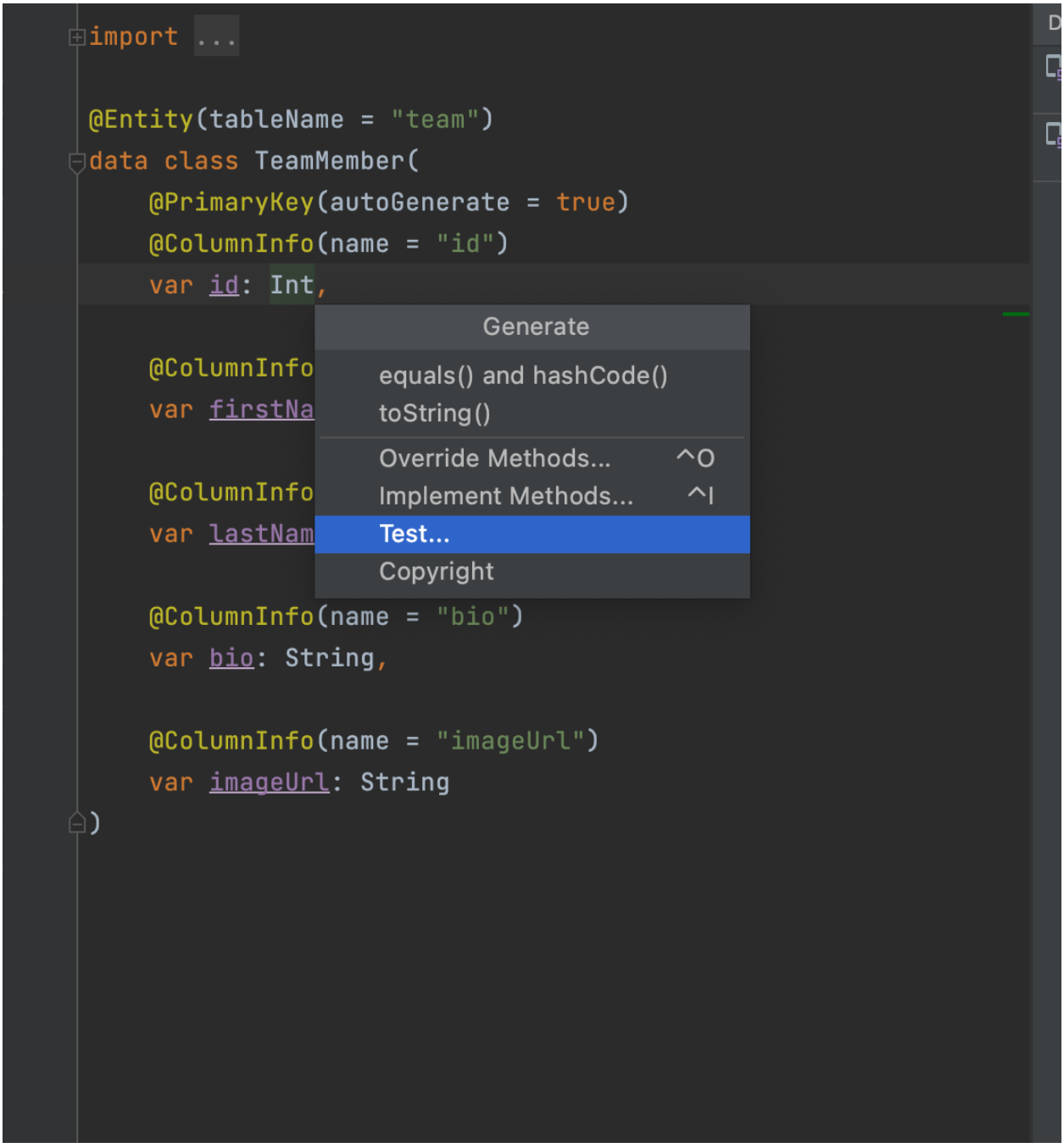

Figure 2: Step 2. Click Generate and then Click Test

From the `Generate` menu, select `Test...` to create a new test file as shown in Figure 2,

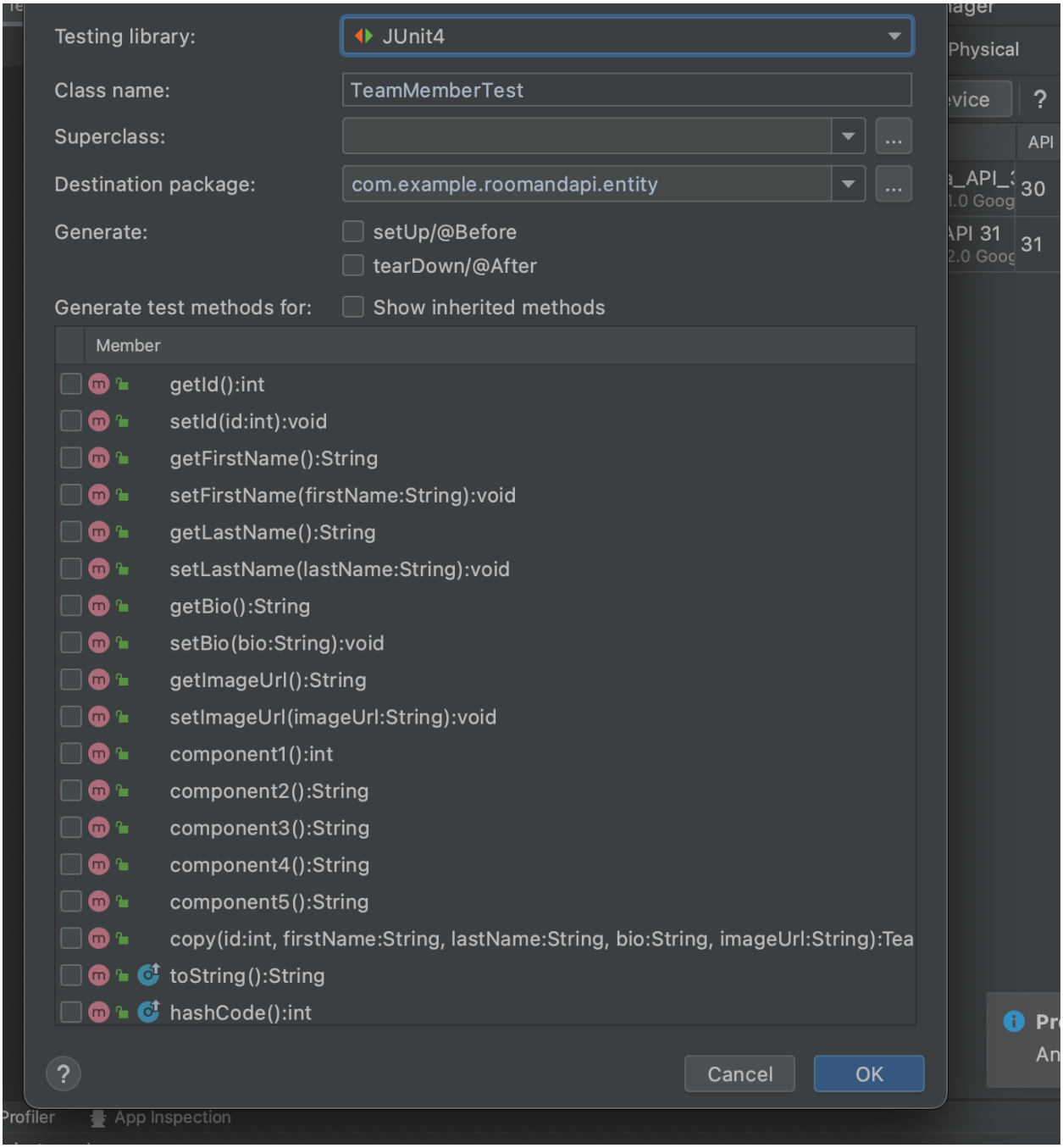

Figure 3: Step 3, Select the version of JUnit to use

The `Create Test` dialog box appears after selecting `Test...`. In this dialog as shown in Figure 3, the user can specify the type of testing framework to use with the `Testing Library` dropdown. In this example, the JUnit4 testing library is used.

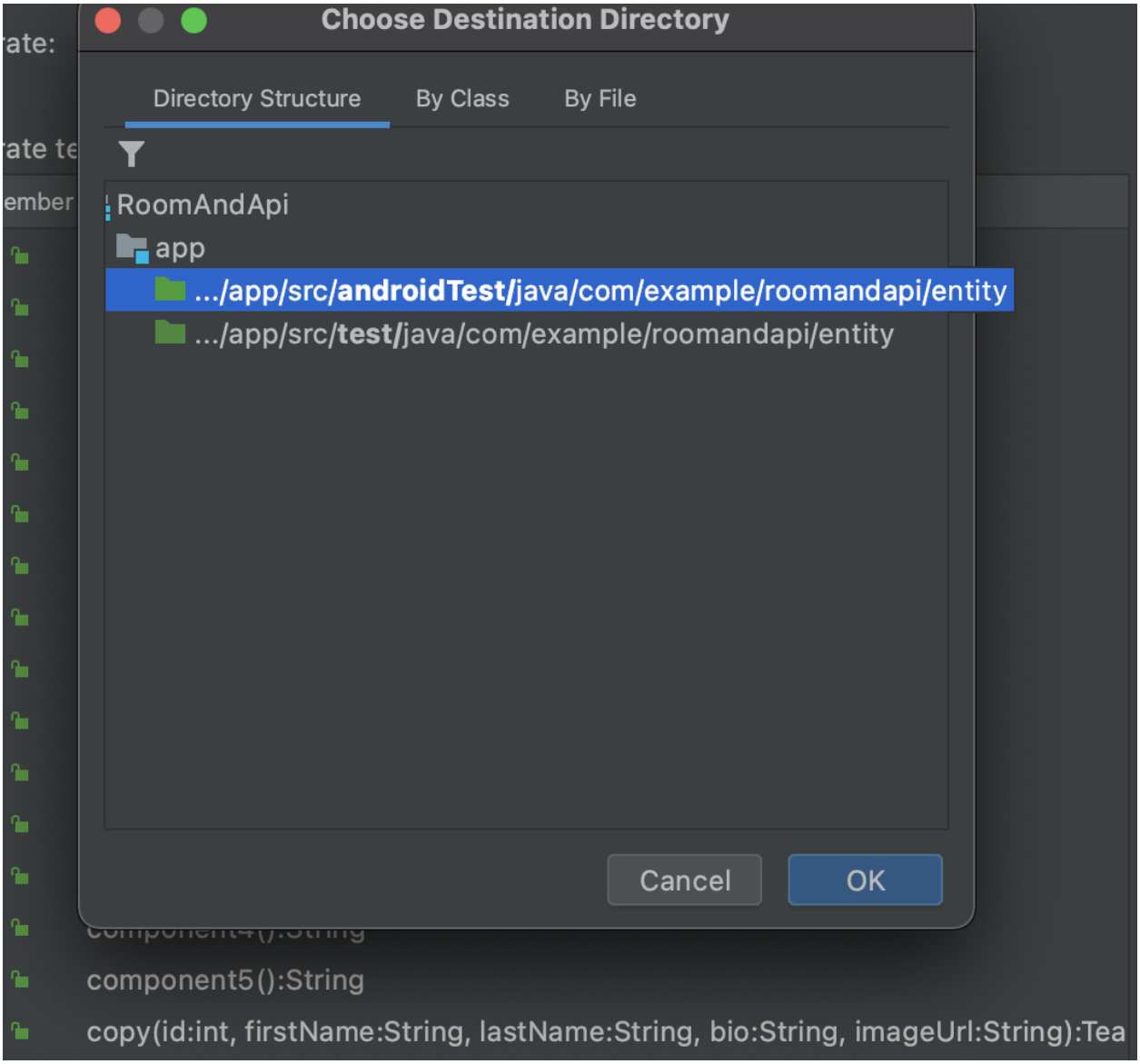

Figure 4: Step 4, Determine the type of test

Then the `OK` button can be clicked to move on to the `Choose Destination` dialog box shown in Figure 4. In this dialog box you can specify whether this will be an AndroidTest running within an emulator or a standard JUnit test by selecting the appropriate destination, AndroidTests or Tests. Since we are testing the RoomDatabase, the test has to be an Android Test and must stored in the AndroidTests directory

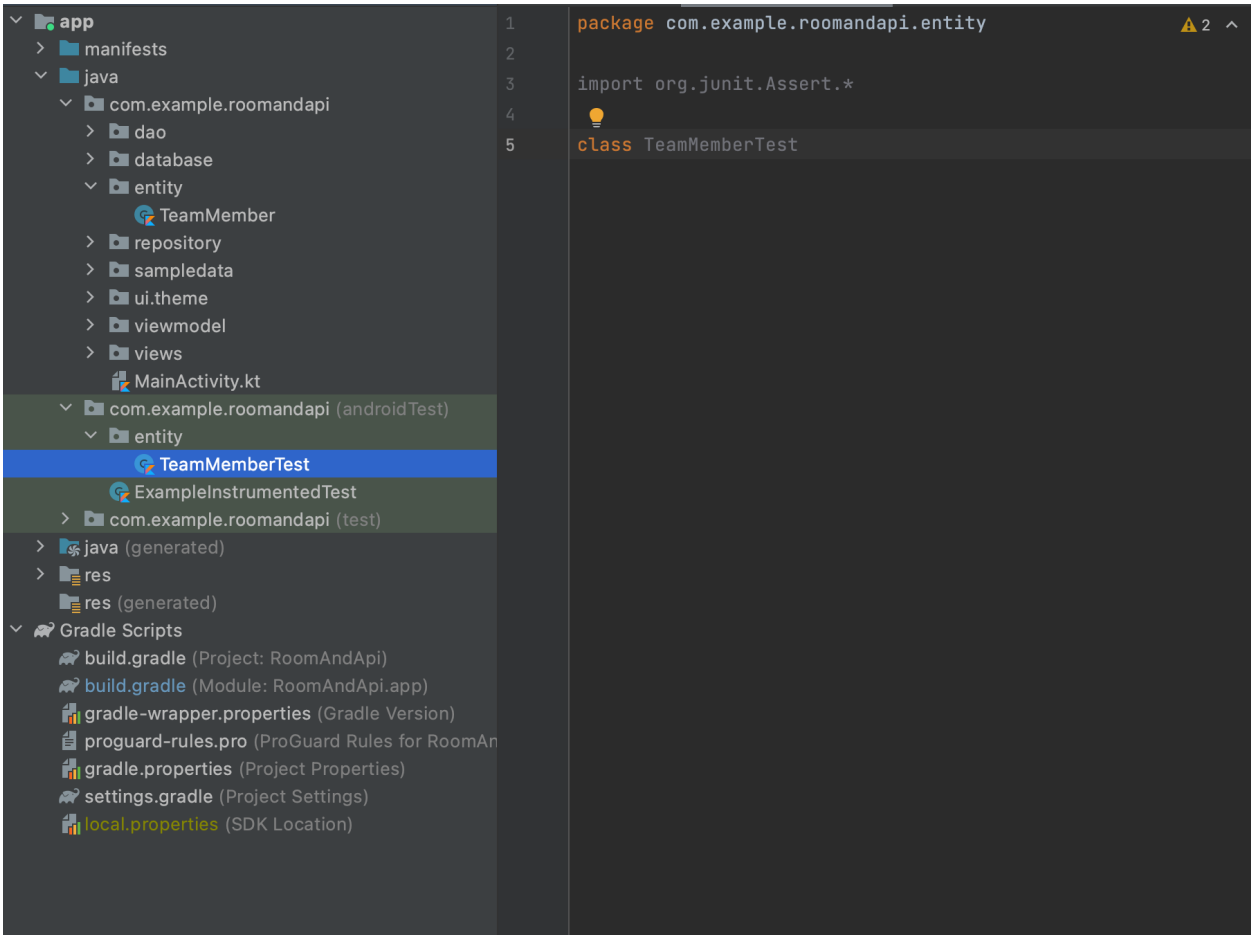

Figure 5: Auto-generated Test Class

After following those steps, a skeleton of an Android Test class will be created . In order to get the class to run properly, an annotation `@RunWith(AndroidJUnit4::class)` needs to be added at the front of the class. The class header will also need to extend the TestCase class using the heading `class nameOfClass: TestCase(){` After that has been created, common variables need to be created as members of the class. When testing the Room Database, the private members should include an instance of the database, the dao being tested and possibly the repository if the repository is being tested.

```
@Before
    public override fun setUp(){
        val context = ApplicationProvider.getApplicationContext<Context
    >()
        db = Room.inMemoryDatabaseBuilder(context, WCDatabase::class.
    java).build()
        dao = db.userDao()
    }
```

Listing 14: Code to Set up the Database in Test

Since the database will have to be created with every test, the `@Before` annotation is needed to define the code to generate the database. Listing 14 contains a `setUp()` function which is used to create a an `inMemoryDatabase` which is a database that is loaded into memory but does not persist after it is closed, which makes it ideal for testing. The inMemoryDatabase is created using Room's `inMemoryDatabseBuilder` class which takes the application context and the class of the database to instantiate with the build() method. Once the database is created, the dao is instantiated from the appropriate database method.

```
    @After
    fun closeDB(){
        db.close()
    }
```

Listing 15: Code to close the Database after the test

Since every test is going to have to close the databse, the `@After` annotation is used to create a function that is run after each test to close the database. An example of this function is shown in Listing 15. The function `closeDB()` simply abstracts the database's close method and calls it.

```
    @Test
    fun TestWriteAndReadUser() = runBlocking(){
        val members: List<TeamMember> =  listOf(
            TeamMember(1,
            "Jacob",
            "Conner",
            "Jacob Conner is a senior at Old Dominion University
   majoring in Computer Science. Currently, he lives in Blacksburg,
   Virginia, and works in a chat-based technical support role for Dish
   Network. Some of his hobbies include archaeology, learning Japanese
   and browsing Linkedin Learning.",
            "https://www.cs.odu.edu/~cpi/old/410/silvers21/bio/
   Jacob_Conner.jpg")
        )
        dao.insert(members)
        val byName: TeamMember = dao.getByName("Jacob", "Conner")
        MatcherAssert.assertThat(byName, CoreMatchers.equalTo(members
   [0]))
    }
```

Listing 16: Example Test for Room Database

Now that the initial setup has been created, tests for the database can be created. A test is annotated with `@Test` before the test function. Each function is set up with runBlocking to force the thread used to call the test to wait until the test finishes [5]. Then the test is setup with anything particular to the test. In Listing 16 the database's ability to insert and read data is tested by creating an instance of a team member. This team member is inserted into the database using the dao's insert method. The database is read to create a new instance of the team member which should be the same as the team member just inserted with the dao's getByName method. The

MatcherAssert.assertThat method is used to test if the two TeamMember entities are equal. If they are equal, the test passes [14].

```
package com.example.roomandapi.entity

import android.content.Context
import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import androidx.test.ext.junit.runners.AndroidJUnit4
import com.example.roomandapi.dao.TeamMemberDao
import com.example.roomandapi.database.WCDatabase
import junit.framework.TestCase
import kotlinx.coroutines.runBlocking
import org.hamcrest.CoreMatchers
import org.hamcrest.MatcherAssert
import org.junit.After
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class TeamMemberTest : TestCase(){
    private lateinit var db: WCDatabase
    private lateinit var dao: TeamMemberDao

    @Before
    public override fun setUp(){
        val context = ApplicationProvider.getApplicationContext<Context
    >()
        db = Room.inMemoryDatabaseBuilder(context, WCDatabase::class.
    java).build()
        dao = db.teamMemberDao()
    }

    @After
    fun closeDB(){
        db.close()
    }

    @Test
    fun TestWriteAndReadUser() = runBlocking(){
        val members: List<TeamMember> =  listOf(
            TeamMember(1,
            "Jacob",
            "Conner",
            "Jacob Conner is a senior at Old Dominion University
    majoring in Computer Science. Currently, he lives in Blacksburg,
    Virginia, and works in a chat-based technical support role for Dish
```

```
    Network. Some of his hobbies include archaeology, learning Japanese
    and browsing Linkedin Learning.",
42              "https://www.cs.odu.edu/~cpi/old/410/silvers21/bio/
    Jacob_Conner.jpg")
43          )
44          dao.insert(members)
45          val byName: TeamMember = dao.getByName("Jacob", "Conner")
46          MatcherAssert.assertThat(byName, CoreMatchers.equalTo(members
    [0]))
47      }
48  }
```

Listing 17: Example entity test class

The example test file for the TeamMember class is shown in listing 17.

5 JetPack Compose

5.1 Navigation

Navigation in JetPack Compose consists of a NavController to provide the routes to the various views and buttons on the view to navigate to those routes [15] [16].

```
1  package com.example.roomandapi.views
2
3  import android.app.Application
4  import androidx.compose.foundation.background
5  import androidx.compose.foundation.layout.Arrangement
6  import androidx.compose.foundation.layout.Column
7  import androidx.compose.foundation.layout.Row
8  import androidx.compose.foundation.layout.fillMaxSize
9  import androidx.compose.material.Button
10 import androidx.compose.material.Text
11 import androidx.compose.runtime.Composable
12 import androidx.compose.runtime.livedata.observeAsState
13 import androidx.compose.ui.platform.LocalContext
14 import androidx.compose.ui.Alignment
15 import androidx.compose.ui.Modifier
16 import androidx.compose.ui.graphics.Color
17 import androidx.compose.ui.text.font.FontFamily
18 import androidx.compose.ui.text.font.FontWeight
19 import androidx.compose.ui.text.style.TextDecoration
20 import androidx.compose.ui.unit.sp
21 import androidx.lifecycle.viewmodel.compose.viewModel
22 import androidx.navigation.NavController
```

```
23 import androidx.navigation.compose.NavHost
24 import androidx.navigation.compose.composable
25 import androidx.navigation.compose.rememberNavController
26
27
28 import com.example.roomandapi.entity.TeamMember
29
30 @Composable
31 fun appNavController() {
32
33     val navController = rememberNavController()
34     NavHost(navController, startDestination = "main") {
35         composable(route = "main") {
36             MainView(navController)
37         }
38         composable(route = "about") {
39             AboutView(navController)
40         }
41         composable(route = "addProject") {
42             AddProjectView(navController)
43         }
44     }
45 }
46
47
48 @Composable
49 fun MainView(navController: NavController) {
50         Column(){
51             Row(){
52                 Button(onClick = { navController.navigate("main") }) {
53                     Text("Main")
54                 }
55                 Button(onClick = { navController.navigate("about") }) {
56                     Text("About Us")
57                 }
58                 Button(onClick = { navController.navigate("addProject")
    }) {
59                     Text("Add a Project")
60                 }
61             }
62             Column(
63                 modifier = Modifier
64                     .fillMaxSize()
65                     .background(Color.White),
66                 horizontalAlignment = Alignment.CenterHorizontally,
67                 verticalArrangement = Arrangement.Center
68             ) {
```

```
69                  Text(
70                      text = "Workout Companion",
71                      fontSize = 30.sp,
72                      fontWeight = FontWeight.Bold,
73                      textDecoration = TextDecoration.Underline,
74                      fontFamily = FontFamily.Serif
75                  )
76              }
77          }
78      }
```

Listing 18: Example Activity View with a NavController

```
@Composable
fun appNavController() {

    val navController = rememberNavController()
    //default destination
    NavHost(navController, startDestination = "nameOfDefaultRoute") {
      //list routes
        composable(route = "<nameOfFirstRoute>") {
              //viewToLoad
      firstView(navController)
        }

        composable(route = "<nameOfSecondRoute>") {
      secondView(navController)
        }
    }
}
```

Listing 19: Function to create a simple Nav Controller

In the navController function shown in listing 19, the navController begins like all composable functions with the `@Composable` annotation. Then the name of the function for the navController is defined. Since the nav controller manages routes, a state variable needs to be handled to keep track of the current route. This state is defined as navController in the example code and is created using the `rememberNavController()` method. Next the NavHost is created which takes the state, and the default or `startDestination`. Within the NavHost function, the `composable` method is called for each route. This method simply takes the route which has a string for the name of the route. Inside of this function, the view that should be loaded with that particular route. Since we need to keep track of the current route, each view used in a navController should require the remembered navController..

```
Button(onClick = { navController.navigate("<routeToNavigateTo") }) {
                //Label for button
                Text("Name of Button")
            }
```

```
}
```

Listing 20: Button to navigate to a view

In order to move between routes, a Button component can be created like the one shown in Listing 20. To make the button switch to a different route, the `onClick` argument needs to be set to `navController.navigate("nameOfRoute")`. Then normal styling can be used inside of the button. For example, a label can be set with the Text component with the Text property set to the name of the button. The text property's `text=` can be omitted when the text is the only property set [15].

5.2 Handling Remote Resources - Images

```
implementation("io.coil-kt:coil-compose:1.3.1")
```

Listing 21: Coil Dependency

Some APIs may make use of Image files referred to remotely and there are instances when we would like the application to download an image from the web and use that image in the web. In order to support this, the `Coil` library can be imported by adding the dependency in Listing 21 into the application's *build.gradle* file.

```
Image(
                        painter = rememberImagePainter("https://www.cs.
   odu.edu/~cpi/old/410/silvers21/bio/Jacob\_Conner.jpg"),
                        contentDescription = null,
                        modifier = Modifier.size(128.dp)
                    )
```

Listing 22: Example Image using Coil

Once the dependency has been imported, the image urls can be loaded in the app by creating an `Image` composable function and setting the painter property with the `rememberImagePainter` function that takes the url as an argument [17]. An example image composable to load a remote resource is shown in Listing 22.

5.3 Creating a form

A common task in most mobile applications is retrieving input from the user and this is most commonly handled through forms. Forms are a collection of various input fields and at least one button, which is used to trigger the submission process where the inputs are retrieved from the form, processed and then sent to a database or the relevant processing tools.

```
1 package com.example.roomandapi.views
2
```

```
3
4  import android.app.Application
5  import androidx.compose.foundation.background
6  import androidx.compose.foundation.clickable
7  import androidx.compose.foundation.layout.*
8  import androidx.compose.material.*
9  import androidx.compose.runtime.*
10 import androidx.compose.runtime.livedata.observeAsState
11 import androidx.compose.ui.Alignment
12 import androidx.compose.ui.Modifier
13 import androidx.compose.ui.draw.alpha
14 import androidx.compose.ui.graphics.Color
15 import androidx.compose.ui.platform.LocalContext
16 import androidx.compose.ui.text.input.TextFieldValue
17 import androidx.compose.ui.unit.dp
18 import androidx.lifecycle.viewmodel.compose.viewModel
19 import androidx.navigation.NavController
20 import com.example.roomandapi.entity.ProjectEntity
21 import com.example.roomandapi.sampledata.teamMemberList
22 import com.example.roomandapi.viewmodel.ProjectViewModel
23 import com.example.roomandapi.viewmodel.ProjectViewModelFactory
24 import com.example.roomandapi.viewmodel.TeamMemberViewModel
25 import com.example.roomandapi.viewmodel.TeamMemberViewModelFactory
26 import com.example.workout_companion.utility.DateTimeConverter
27 import com.vanpra.composematerialdialogs.MaterialDialog
28 import com.vanpra.composematerialdialogs.datetime.date.datepicker
29 import com.vanpra.composematerialdialogs.rememberMaterialDialogState
30 import java.time.LocalDate
31 import java.time.format.DateTimeFormatter
32
33 @Composable
34 fun AddProjectView(navController: NavController) {
35     val context = LocalContext.current
36     val projectViewModel: ProjectViewModel = viewModel(
37         factory = ProjectViewModelFactory(context.applicationContext as
    Application)
38     )
39     //remember the values in the form
40     var projectName by remember { mutableStateOf("") }
41     val projectTypeState = remember { mutableStateOf("")}
42     val radioOptions = listOf("Easy", "Intermediate", "Difficult")
43     val difficulty = remember { mutableStateOf("") }
44     val dateValue = remember { mutableStateOf("") }
45     val UserID = remember { mutableStateOf(0)}
46     val UserIDString = remember { mutableStateOf("")}
47
48     Column(
```

```
        Modifier.fillMaxWidth()
            .padding(start = 30.dp, end = 30.dp)
    ) {
        //Store some navigation buttons
        Row(){
            Button(onClick = { navController.navigate("main") }) {
                Text("Main")
            }
            Button(onClick = { navController.navigate("about") }) {
                Text("About Us")
            }
            Button(onClick = { navController.navigate("addProject") }) {
                Text("Add a Project")
            }
        }
        //set a top margin
        Spacer(modifier = Modifier.padding(top = 30.dp))
        //Create a heading
        Row(
            Modifier
                .fillMaxWidth(),
            horizontalArrangement = Arrangement.Center

        ) {
            Text(text = "Add a Project")
        }

        Spacer(modifier = Modifier.padding(top = 20.dp))

        //Form
        //Project Name Row
        // row has padding to the left and right margins
        Row() {
            Text(text = "Name of Project:")
            Spacer(modifier = Modifier.padding(10.dp))
            TextField(value = projectName,
                onValueChange = { projectName = it }
            )
        }
        Spacer(modifier = Modifier.padding(top = 10.dp))

        //Project Difficulty Radiobuttons
        // row has padding to the left and right margins
        Text("Project Difficulty: ")
        Row(
            Modifier.fillMaxWidth()
```

```
95          ) {
96              GroupedRadioButton(radioOptions, difficulty)
97          }
98          Spacer(modifier = Modifier.padding(top = 10.dp))
99
100         //Project Type Dropdown Row
101         // row has padding to the left and right margins
102         Row() {
103             Text(text = "Project Type:")
104             Spacer(modifier = Modifier.padding(10.dp))
105             ProjectTypeDropdown(projectTypeState)
106         }
107         //Date Time Picker
108         Spacer(modifier = Modifier.padding(top = 10.dp))
109         Row() {
110             Text(text = "Creation Date:")
111             Spacer(modifier = Modifier.padding(10.dp))
112             showDatePicker(dateValue)
113         }
114         //User Dropdown
115         Spacer(modifier = Modifier.padding(top = 10.dp))
116         Row() {
117             Text(text = "User ID:")
118             Spacer(modifier = Modifier.padding(10.dp))
119             UserDropdown(UserID)
120         }
121         Spacer(modifier = Modifier.padding(top = 10.dp))
122
123         //Submit Button
124         Row(horizontalArrangement = Arrangement.Center,
125         modifier = Modifier.padding(start = 120.dp, end = 120.dp)) {
126             Button(onClick = { submit(projectName, difficulty.value,
127                                       projectTypeState.value, dateValue.
    value,
128                                       UserID.value, projectViewModel) })
    {
129                 Text("Submit")
130             }
131         }
132         Spacer(modifier = Modifier.padding(top = 10.dp))
133         // An approach for checking to see if state is being handled
    correctly
134         //just a bunch of text fields with the value set to a
    particular state and
135         // an onValueChange function that updates the value when the
    state changes
136 //        Row() {
```

```
137 //             Text("Current Project Name: ")
138 //             TextField(
139 //                 value = projectName,
140 //                 onValueChange = { projectName = it },)
141 //             Spacer(modifier = Modifier.padding(10.dp))
142 //         }
143 //         Row() {
144 //             Text("Current Project Difficulty: ")
145 //             TextField(
146 //                 value = difficulty.value,
147 //                 onValueChange = { difficulty.value = it },)
148 //             Spacer(modifier = Modifier.padding(10.dp))
149 //         }
150 //         Row() {
151 //             Text("Current Project Type: ")
152 //             TextField(
153 //                 value = projectTypeState.value,
154 //                 onValueChange = { projectTypeState.value = it },)
155 //             Spacer(modifier = Modifier.padding(10.dp))
156 //         }
157 //         Row() {
158 //             Text("Current Date: ")
159 //             TextField(
160 //                 value = dateValue.value,
161 //                 onValueChange = { dateValue.value = it },)
162 //             Spacer(modifier = Modifier.padding(10.dp))
163 //         }
164     }
165 }
166
167 fun submit(projectName: String, difficulty: String,
168            projectType:String, dateValue:String, UserID: Int,
    projectViewModel: ProjectViewModel ){
169
170     val formatter = DateTimeFormatter.ISO_LOCAL_DATE
171     val date: LocalDate = LocalDate.parse(dateValue, formatter)
172     var project = ProjectEntity(1, projectName, difficulty, projectType
    , date, UserID)
173     projectViewModel.addProject(project)
174 }
175
176 @Composable
177 fun ProjectTypeDropdown(projectTypeState: MutableState<String>) {
178     var expanded by remember { mutableStateOf(false) }
179     val items = listOf("Personal Project", "School Project", "Work
    Project")
180     var selectedIndex by remember { mutableStateOf(0) }
```

```
181     Box(modifier = Modifier.wrapContentSize(Alignment.TopStart)) {
182         Text(items[selectedIndex],modifier = Modifier
183             .fillMaxWidth()
184             .clickable(onClick = { expanded = true })
185             .background(
186                 Color.LightGray
187             )
188         )
189         DropdownMenu(
190             expanded = expanded,
191             onDismissRequest = { expanded = false },
192             modifier = Modifier
193                 .fillMaxWidth()
194                 .background(
195                     Color.LightGray
196                 )
197         ) {
198             items.forEachIndexed { index, s ->
199                 DropdownMenuItem(onClick = {
200                     selectedIndex = index
201                     expanded = false
202                     projectTypeState.value = items[selectedIndex]
203                 }) {
204                     Text(text = s)
205                 }
206             }
207         }
208     }
209 }
210
211
212 @Composable
213 fun GroupedRadioButton(mItems: List<String>, radioState: MutableState<
    String>) {
214
215     Row(Modifier
216         .fillMaxWidth()) {
217         mItems.forEach { mItem ->
218                 RadioButton(
219                     selected = radioState.value == mItem,
220                     onClick = { radioState.value = mItem },
221                     enabled = true,
222                     colors = RadioButtonDefaults.colors(selectedColor =
    Color.Magenta)
223                 )
224             Text(text = mItem, modifier = Modifier.padding(start = 8.dp
    ))
```

```
        }
    }
}

@Composable
fun ReadonlyTextField(
    value: TextFieldValue,
    onValueChange: (TextFieldValue) -> Unit,
    modifier: Modifier = Modifier,
    onClick: () -> Unit,
    label: @Composable () -> Unit
) {
    Box {
        TextField(
            value = value,
            onValueChange = onValueChange,
            modifier = modifier,
            label = label
        )
        Box(
            modifier = Modifier
                .matchParentSize()
                .alpha(0f)
                .clickable(onClick = onClick),
        )
    }
}


@Composable
fun showDatePicker(DateValue: MutableState<String>){

    val dialogState = rememberMaterialDialogState()
    val textState = remember { mutableStateOf(TextFieldValue()) }
    val dialog = MaterialDialog(
        dialogState = dialogState,
        buttons = {
            positiveButton("Ok")
            negativeButton("Cancel")
        }){
        datepicker { date ->
            val formatter = DateTimeFormatter.ISO_LOCAL_DATE
            val formattedDate = date.format(formatter)
            textState.value = TextFieldValue(formattedDate)
            DateValue.value = formattedDate
        }
    }
```

```
272
273     Column(modifier = Modifier.padding(16.dp)) {
274         ReadonlyTextField(
275             value = textState.value,
276             onValueChange = { textState.value = it },
277             onClick = {
278                 dialogState.show()
279             },
280
281             label = {
282                 Text(text = "Date")
283             }
284         )
285     }
286 }
287
288 @Composable
289 fun UserDropdown(UserID: MutableState<Int>) {
290     var expanded by remember { mutableStateOf(false) }
291     val context = LocalContext.current
292     val teamMemberViewModel: TeamMemberViewModel = viewModel(
293         factory = TeamMemberViewModelFactory(context.applicationContext
    as Application)
294     )
295     teamMemberViewModel.addTeamMember(teamMemberList)
296     val teamList = teamMemberViewModel.readAllTeamMembers.
   observeAsState(listOf()).value
297     val items = listOf("Personal Project", "School Project", "Work
   Project")
298     var selectedIndex by remember { mutableStateOf(0) }
299     Box(modifier = Modifier.wrapContentSize(Alignment.TopStart)) {
300         if(teamList.isNotEmpty()) {
301             Text(
302                 "${teamList[selectedIndex].id}",
303                 modifier = Modifier
304                     .fillMaxWidth()
305                     .clickable(onClick = { expanded = true })
306                     .background(
307                         Color.LightGray
308                     )
309             )
310         }
311         else{
312             Text(
313                 " ",
314                 modifier = Modifier
315                     .fillMaxWidth()
```

```
                    .clickable(onClick = { expanded = true })
                    .background(
                        Color.LightGray
                    )
            )
        }
        DropdownMenu(
            expanded = expanded,
            onDismissRequest = { expanded = false },
            modifier = Modifier
                .fillMaxWidth()
                .background(
                    Color.LightGray
                )
        ) {
            teamList.forEachIndexed { index, s ->
                DropdownMenuItem(onClick = {
                    selectedIndex = index
                    UserID.value = teamList[selectedIndex].id
                    expanded = false
                }) {
                    Text(text = "${s.firstName} ${s.lastName}")
                }
            }
        }
    }
}
```

Listing 23: Example View with a form to collect projects that a team member may work on

An example of a form created using JetPack Compose is shown in Listing 23. This form is a bit contrived to providet a variety of the more common input, but it is used to collect information about projects that a team member may work on. The form first consists of a text field to get the name of the project, a set of radio buttons that a user can use to select the perceived difficulty of the project, a dropdown box to set the category of project, a date for when the. project was created and lastly a drop down box where the user can specify the team member assigned to it, and it will return the user id in the team table for that team member.

The code used to create the form is shown in the AddProjectView function. This function is a composable function that is used in other composable functions and makes use of other other composable functions, so the `@Composable` annotation is used. so it makes use of the @Composable which is accessible by a route in the NavController, so the NavController has to be passed to the function.

```
    val context = LocalContext.current
    val projectViewModel: ProjectViewModel = viewModel(
        factory = ProjectViewModelFactory(context.applicationContext as
    Application)
```

```
    )
```

Listing 24: Creating a ViewModel in a View

Since the form will have to insert the user's input into a database, the ProjectViewModel is created using the code shown in 24. ViewModels require the context of the application, so the context variable is created using the LocalContext.current to get the context of the current view. A `ProjectViewModel` defined in the `viewmodel/ProjectViewModel.kt` class is created using the ProjectViewModelFactory method with the context variable. The resulting projectViewModel value can now be used in other parts of the view to make use of the various CRUD operations in that view model.

```
//remember the values in the form
   var projectName by remember { mutableStateOf("") }
   val projectTypeState = remember { mutableStateOf("")}
   val difficulty = remember { mutableStateOf("") }
   val dateValue = remember { mutableStateOf("") }
   val UserID = remember { mutableStateOf(0)}
   val UserIDString = remember { mutableStateOf("")}
```

Listing 25: Tracking State Changes

Since JetPack Compose makes use of states to track changes in values of the various fields in our form, a number of mutable states need to be created using the `remember{ mutableStateOf("") }` function. These can then be passed to the values of textFields, dropDowns and radio buttons and are frequently manipulated using the `onValueChange` property of those fields using code like `onValueChange = { myState.value = it }`, which can be used to tell the field to automatically update the value property of the state when the user enters input in that particular field. There should be at least one state per input field in the form and certain inputs may make use of more if the input field has to trigger a dialog box or a dropdown menu. The states used to track the values in the AddProjectForm are shown in Listing 25

5.3.1 Rows and Columns

Now that the initial discussion on state and using viewmodels in a view has been provided, the process of creating the form and the elements in it will be discussed. In JetPack Compose there are a variety of containers to organize elements on a screen, but the two most common are `Column()` and `Row()`. Columns arrange elements vertically and rows arrange elements horizontally. Both Column and Row take a variety of optional arguments that can be used to further customize the layout of the composable function. In this example form, I have stored all of the elements in a Column that fills up the width of the screen using the `Modifier.fillMaxWidth()` property with the exception of a left and right margin set to 30.dp using the `Modifier.padding(start=30.dp, end=30.d` property. At the top of the form, which is the first element in the Column composable, the navigation elements are stored in a Row() where buttons to go to the main screen, about us screen and the add a project screen. These buttons are placed immediately after each other, so they are placed touching each other from right to left.

5.3.2 Spacers

```
Spacer(modifier = Modifier.padding(top = 30.dp))
```

Listing 26: Using Spacers

In order to create white space between various elements a spacer can be used. A spacer takes a modifier argument which can be set to the Modifier object with a property. In Listing 26 the spacer has a top margin of 30.dp which is used to provide 30.dp units of white space between the navigation and the next element. Text A title can be created for the form using a `Text()` element. There are several properties that can be set for the text property which can used to style the text, but at a minimum it requires an argument of text which is the string displayed for the text.

5.3.3 Centering

```
//Create a heading
      Row(
          Modifier
              .fillMaxWidth(),
          horizontalArrangement = Arrangement.Center

      ) {
          Text(text = "Add a Project")
      }
```

Listing 27: Centered Text

If we simply place the text element in the column, we will find the text element is left aligned with a padding of 30.dp. Generally, a centered title is more visually appealing, so we can create a Row container and make use of the Modifier.fillMaxWidth() property and the horizontalArrangement = Arrangement.Center property to create a row where all items in that row are center aligned. Then the text for our title can be stored inside this Row container as shown in Listing 27.

5.3.4 TextField Inputs

```
TextField(value = projectName,
              onValueChange = { projectName = it }
          )
```

Listing 28: TextField inputs

Now that a title has been created, we can start creating the various input types needed for our form. The first input in this form is the text input field used to get the name of the project. Generally forms make use of a label before the input field, so I opted to create a Row container for each input that consisted of a `Text` field with the name of the field, a spacer with a padding of 10.dp and then

the input type of interest. Since project name is a text field, the TextField composable is used as shown in Listing 28. TextFields have at least two arguments, a value argument which points to the state used to track this field and the onValueChange which is used to update the value of that state.

5.3.5 RadioButtons

```
@Composable
fun GroupedRadioButton(mItems: List<String>, radioState: MutableState<
   String>) {

    Row(Modifier
        .fillMaxWidth()) {
        mItems.forEach { mItem ->
                RadioButton(
                    selected = radioState.value == mItem,
                    onClick = { radioState.value = mItem },
                    enabled = true,
                    colors = RadioButtonDefaults.colors(selectedColor =
    Color.Magenta)
                )
            Text(text = mItem, modifier = Modifier.padding(start = 8.dp
   ))
        }
    }
}
            )
```

Listing 29: Radiobutton inputs

The second input type used in the form is the radio button which is used to specify the perceived difficulty of the project. Since there were more than a few lines of code to create a radio button, a GroupedRadioButton function was created to store a list of strings for the various options to display and the mutableState used to track the value of the radiobutton after any changes. The GroupedRadioButton composable function is shown in Listing 29

```
@Composable
fun ProjectTypeDropdown(projectTypeState: MutableState<String>) {
    var expanded by remember { mutableStateOf(false) }
    val items = listOf("Personal Project", "School Project", "Work
   Project")
    var selectedIndex by remember { mutableStateOf(0) }
    Box(modifier = Modifier.wrapContentSize(Alignment.TopStart)) {
        Text(items[selectedIndex],modifier = Modifier
            .fillMaxWidth()
            .clickable(onClick = { expanded = true })
            .background(
                Color.LightGray
```

```
            )
        )
        DropdownMenu(
            expanded = expanded,
            onDismissRequest = { expanded = false },
            modifier = Modifier
                .fillMaxWidth()
                .background(
                    Color.LightGray
                )
        ) {
            items.forEachIndexed { index, s ->
                DropdownMenuItem(onClick = {
                    selectedIndex = index
                    expanded = false
                    projectTypeState.value = items[selectedIndex]
                }) {
                    Text(text = s)
                }
            }
        }
    }
}
```

Listing 30: DropDown box

5.3.6 Dropdown Boxes

A dropdown or select box can be used to select one option from a list of options, much like radio buttons. The advantage to dropdown boxes is that they make better use of space and can store more options on a screen through the ability to scroll down a list of different options, whereas all radio buttons have to be displayed at once which limits their use to cases where only a few options are available. In this form there are two dropdown boxes, one is used to select the type of project, personal, school or work and the other is to select the name of a user and it returns the id fo the user. The biggest difference in the implementation of these two drop downs is that the UserDropDown box creates a TeamMemberViewModel and uses it to populate a list of TeamMember objects to get the names and ids of all team members. Since a query to the database is used and there could be some delay, conditional logic has to be used for the currently selected value to be an empty string if the list is empty. Otherwise the application will crash due to an invalid index in the list. In Listing 30, the code to create the project dropdown box is shown. In this function, three mutable states are used. First the expanded mutable state tracks a boolean to see if the dropdown is open or not and is set to true when clicked. The second mutable state, selectedIndex, is used to track an integer equal to the index of the currently selected list item in the dropdown box. The last mutable state is projectTypeState which is the state passed to the function and is used to track the current string value of the item at the selectedIndex. A box container containing a single text element is used to

define the appearance of the dropdown field when it is closed. The `DropdownMenu` container is used to define the appearance of the dropdown menu when it is clicked and the list of all dropdown items is shown. Inside of this function, the list items are looped and `DropdownMenuItems` are created from that list each with an `onClick` method which is used to set the values of the various states needed in the Dropdown function. A text element inside of the DropdownMenuItem is used to set the text displayed for that item in the dropdown menu.

5.3.7 DatePicker

The last type of input field in the AddProjectView is the DatePicker input. JetPack Compose does not natively support a DatePicker, but it can be implemented using a third party library such as Compose Material Dialogs [18].

```
  implementation "io.github.vanpra.compose-material-dialogs:datetime
   :0.6.0"
  implementation "io.github.vanpra.compose-material-dialogs:color
   :0.6.0"
```

Listing 31: Compose Material Dialogs Dependencies

To use this library the dependencies shown in Listing 31 need to be added to the `app/build.gradle` file.

```
  @Composable
fun showDatePicker(DateValue: MutableState<String>){

    val dialogState = rememberMaterialDialogState()
    val textState = remember { mutableStateOf(TextFieldValue()) }
    val dialog = MaterialDialog(
        dialogState = dialogState,
        buttons = {
            positiveButton("Ok")
            negativeButton("Cancel")
        }){
        datepicker { date ->
            val formatter = DateTimeFormatter.ISO_LOCAL_DATE
            val formattedDate = date.format(formatter)
            textState.value = TextFieldValue(formattedDate)
            DateValue.value = formattedDate
        }
    }
        Column(modifier = Modifier.padding(16.dp)) {
        ReadonlyTextField(
            value = textState.value,
            onValueChange = { textState.value = it },
            onClick = {
                dialogState.show()
```

```
            },

            label = {
                Text(text = "Date")
            }
        )
    }
```

Listing 32: DatePicker

The DatePicker function is created using the code shown in Listing 32. [19]. This code basically creates a dialog box using the MaterialDialog function and then a datepicker function is used to take the date selected and convert it to a string. After the dialog box has been created, the dialog state's `show` method needs to be called with an onclick event to be able to show the dialog box.

```
@Composable
fun ReadonlyTextField(
    value: TextFieldValue,
    onValueChange: (TextFieldValue) -> Unit,
    modifier: Modifier = Modifier,
    onClick: () -> Unit,
    label: @Composable () -> Unit
) {
    Box {
        TextField(
            value = value,
            onValueChange = onValueChange,
            modifier = modifier,
            label = label
        )
        Box(
            modifier = Modifier
                .matchParentSize()
                .alpha(0f)
                .clickable(onClick = onClick),
        )
    }
}
```

Listing 33: ReadonlyTextField

In this case a custom textField class, called `ReadonlyTextField` shown in Listing 33 is used to create a clickable text element that can show the resulting date text, but does not trigger the keyboard. In this class a box element with a text field and a box element over top of the text field is used to prevent triggering the keyboard, so it has the ability to behave like a button, but also dynamically updates the text value when the datePicker is closed [19].

5.3.8 Submit Button

```
Row(horizontalArrangement = Arrangement.Center,
       modifier = Modifier.padding(start = 120.dp, end = 120.dp)) {
           Button(onClick = { submit(projectName, difficulty.value,
                                     projectTypeState.value, dateValue.
  value,
                                     UserID.value, projectViewModel) })
  {
               Text("Submit")
           }
       }
```

Listing 34: Submit Button

The last element of the AddProjectView is the submit button. This is created using a `Button` with an onClick argument set to the function used to submit the form data. The label for the button is set by storing a `Text` element inside of the button. The code used to create the submit button is shown in **??**

5.3.9 Submit Button

```
fun submit(projectName: String, difficulty: String,
           projectType:String, dateValue:String, UserID: Int,
  projectViewModel: ProjectViewModel ){

   val formatter = DateTimeFormatter.ISO_LOCAL_DATE
   val date: LocalDate = LocalDate.parse(dateValue, formatter)
   var project = ProjectEntity(1, projectName, difficulty, projectType
  , date, UserID)
   projectViewModel.addProject(project)
}
```

Listing 35: Submit Function

The submit function simply takes the value for each of the mutableStates that was tracking an input field and then viewModel used to insert the data into the database. The various values are used to create a ProjectEntity and then the ProjectViewModel's addProject method is used to add the entity to the database. The code used to create the submit function is shown in Listing 35

6 Documenting with Dokka

test

List of Figures

List of Tables

Listings

References

[1] Google, Save data in a local database using room, accessed on 2021-9-22.

URL https://web.archive.org/web/20210918002911/https://developer.android.com/training/data-storage/room

[2] MakeItEasy, How to create room database mvvm pattern in jetpack compose part - i, accessed on 2021-9-27.

URL https://www.youtube.com/watch?v=sbyiq0aM2iw&t=2s

[3] MakeItEasy, How to create room database mvvm pattern in jetpack compose part - 2, accessed on 2021-9-27.

URL https://www.youtube.com/watch?v=UxsXRFl-Ho0

[4] Google, Android room with a view - kotlin, accessed on 2021-9-27.

URL https://developer.android.com/codelabs/android-room-with-a-view-kotlin#0

[5] Kotlin, Coroutines basics, accessed on 2021-9-22.

URL https://web.archive.org/web/20210722140552/https://kotlinlang.org/docs/coroutines-basics.html

[6] Android, Roomdatabase.builder, accessed on 2021-9-22.

URL https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase.Builder

[7] SQLite, Datatypes in sqlite version 3, accessed on 2021-9-27.

URL https://web.archive.org/web/20210902014614/https://www.sqlite.org/datatype3.html

[8] Android, Referencing complex data using room referencing complex data using room, accessed on 2021-9-27.

URL https://web.archive.org/web/20210917234748/https://developer.android.com/training/data-storage/room/referencing-data

[9] LuaSoftwareCode, Java.time.localdatetime converter for room (kotlin), accessed on 2021-9-28.

URL https://code.luasoftware.com/tutorials/android/localdatetime-converter-for-room/

[10] Oracle, Class datetimeformatter, accessed on 2021-9-27.

URL https://web.archive.org/web/20210813002023/https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html

[11] Android, Datetimeformatter, accessed on 2021-9-27.

URL https://web.archive.org/web/20200820054416/https://developer.android.com/reference/kotlin/java/time/format/DateTimeFormatter

[12] StackOverflow, How to format localdate to string?, accessed on 2021-9-28.

URL https://web.archive.org/web/20161130210232/http://stackoverflow.com/questions/28177370/how-to-format-localdate-to-string

[13] Android, Fundamentals of testing, accessed on 2021-9-27.

URL https://web.archive.org/web/20210912110612/https://developer.android.com/training/testing/fundamentals

[14] SimplifiedCoding, Android unit test room database, accessed on 2021-9-27.

URL https://www.youtube.com/watch?v=Hb0UV0f6kmQ

[15] J. Wolf, Navigating in jetpack compose, accessed on 2021-9-27.

URL https://web.archive.org/web/20210723153430/https://medium.com/google-developer-experts/navigating-in-jetpack-compose-78c78d365c6a

[16] Google, Jetpack compose navigation, accessed on 2021-9-27.

URL https://developer.android.com/codelabs/jetpack-compose-navigation

[17] StackOverflow, How to load image from url in jetpack compose?, accessed on 2021-9-28.

URL https://stackoverflow.com/questions/66957702/how-to-load-image-from-url-in-jetpack-compose

[18] P. Maganti, Compose material dialogs, accessed on 2021-10-7.

URL https://github.com/vanpra/compose-material-dialogs

[19] M. Grobmann, Jetpack compose: Datepicker-textfield, accessed on 2021-10-7.

URL https://caelis.medium.com/jetpack-compose-datepicker-textfield-39808e42646a