Android Template Code

Jacob Conner

September 27, 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      //API handling
86    //  implementation "org.jetbrains.anko:anko-common:0.10.8"
87      implementation 'com.google.code.gson:gson:2.8.7'
88      implementation 'com.squareup.retrofit2:retrofit:2.9.0'
89      implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
90
91  }
```

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 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 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 manifests 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 [**?**] .

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 [?] [?] [?].

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")
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     @Insert(onConflict = OnConflictStrategy.REPLACE)
16     suspend fun insert(item: List<TeamMember>)
17
18     @Update
19     suspend fun update(item: TeamMember)
20
21     @Delete
22     suspend fun delete(item: TeamMember)
23
24     @Query("DELETE FROM team")
25     suspend fun deleteAll()
26 }
```

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 [**?**].

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 com.example.roomandapi.dao.TeamMemberDao
import com.example.roomandapi.entity.TeamMember

@Database(entities = [
    TeamMember::class
                      ],
    version = 1,
    exportSchema = false)
@TypeConverters(DateTimeConverter::class)
abstract class WCDatabase: RoomDatabase() {
    abstract fun teamMemberDao(): TeamMemberDao

    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 [**?**].

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 {
    /**
     * DataTimeFormatter object with the pattern used to format
   LocalDateStrings
     */
    @RequiresApi(Build.VERSION_CODES.O)
    private val formatter = DateTimeFormatter.ISO_LOCAL_DATE

    /**
     * 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 [**?**]. 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 [**?**].

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 [**?**]. Once the formatting string has been defined, the converter class contains two methods for converting to and from the LocalDate class.

```
@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)
```

```
        }
    }

    fun updateTeamMember(item: TeamMember){
        viewModelScope.launch(Dispatchers.IO){
            repository.updateTeamMember(item = item)
        }
    }

    fun deleteTeamMember(item: TeamMember){
        viewModelScope.launch(Dispatchers.IO){
            repository.deleteTeamMember(item = item)
        }
    }

    fun deleteAllTeamMembers(){
        viewModelScope.launch(Dispatchers.IO){
            repository.deleteAll()
        }
    }
}

class TeamMemberViewModelFactory(
    private val application: Application
): ViewModelProvider.Factory{
    override fun <T: ViewModel?> create(modelClass: Class<T>): T{
        @Suppress("UNCHECKED_CAST")
        if(modelClass.isAssignableFrom(TeamMemberViewModel::class.java)
   ){
            return TeamMemberViewModel(application) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

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 [**?**]. Room Database is one of those examples.

4.1 Creating Tests

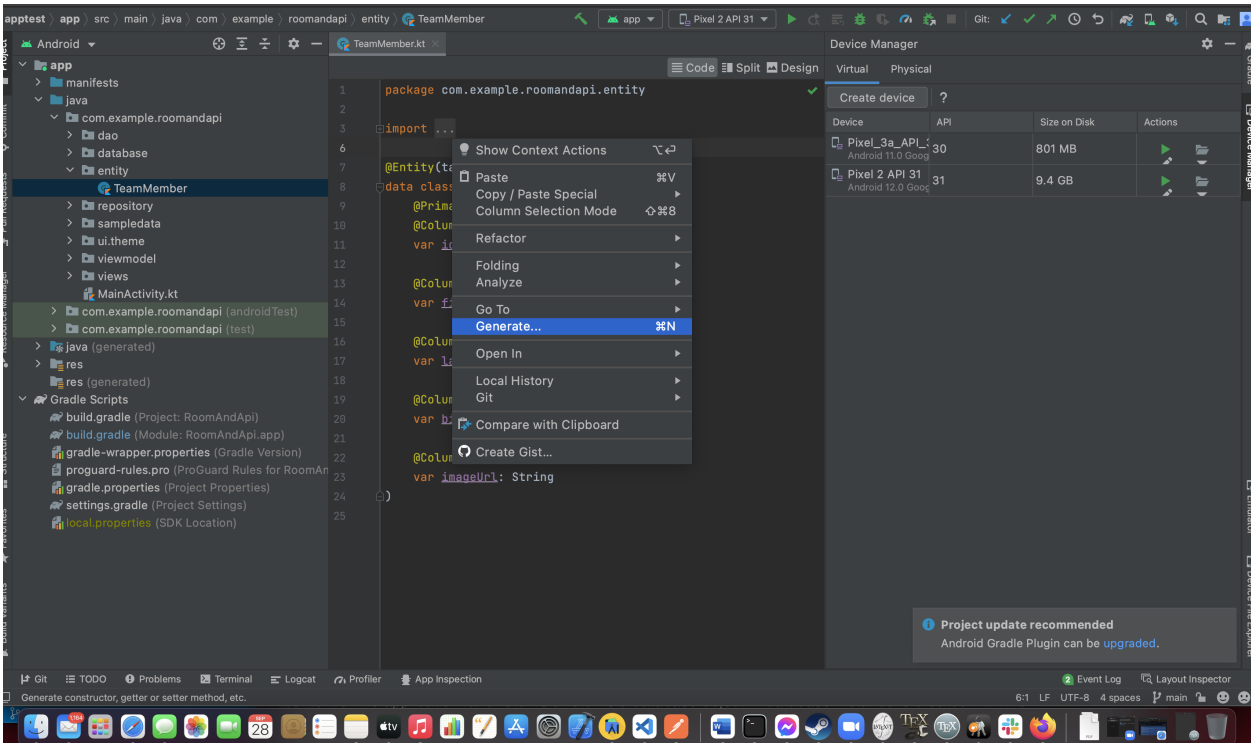

Figure 1: Step 1. Open the file you want to test and right click it.

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

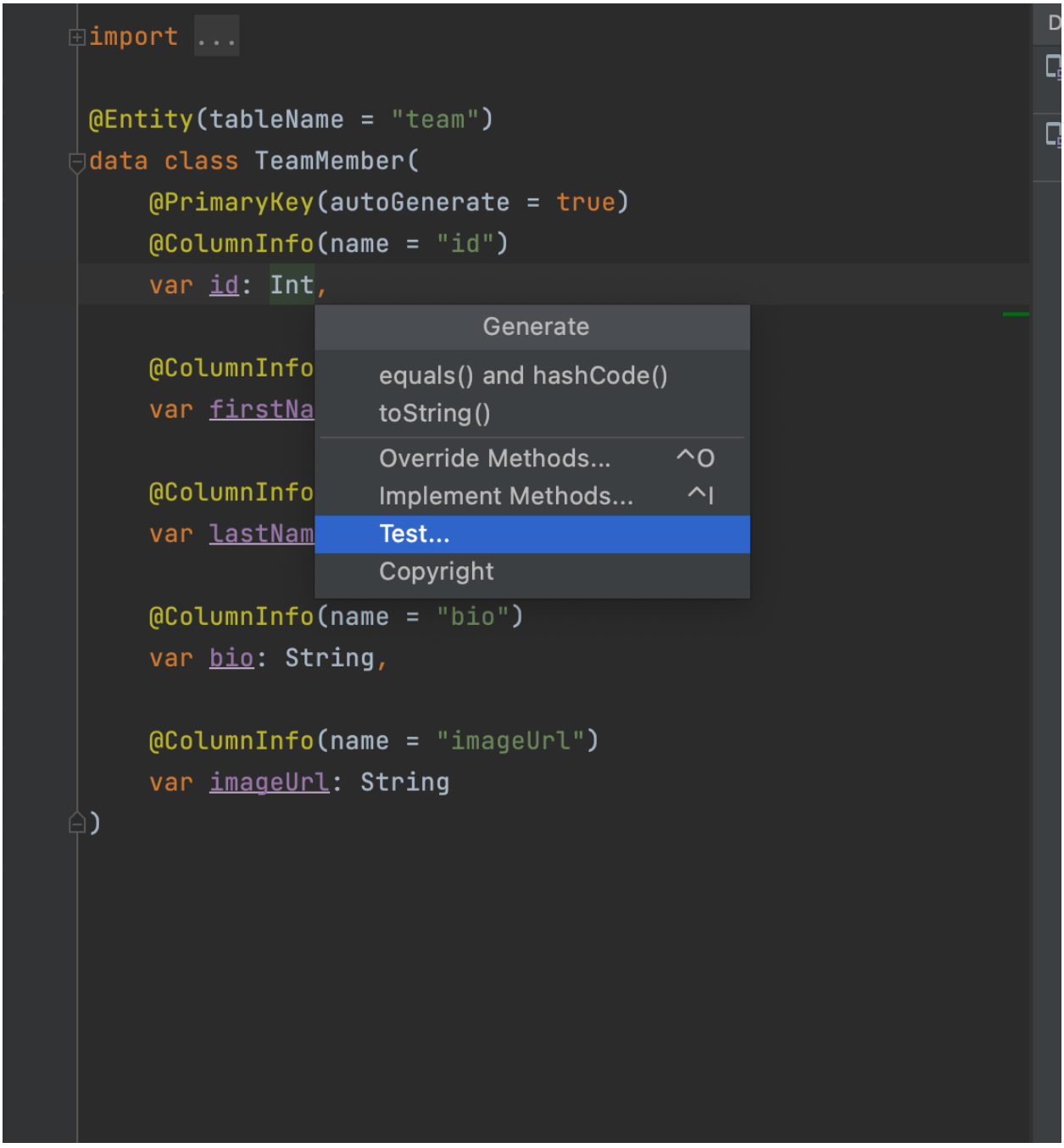

Figure 2: Step 2. Click Generate and then Click Test

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

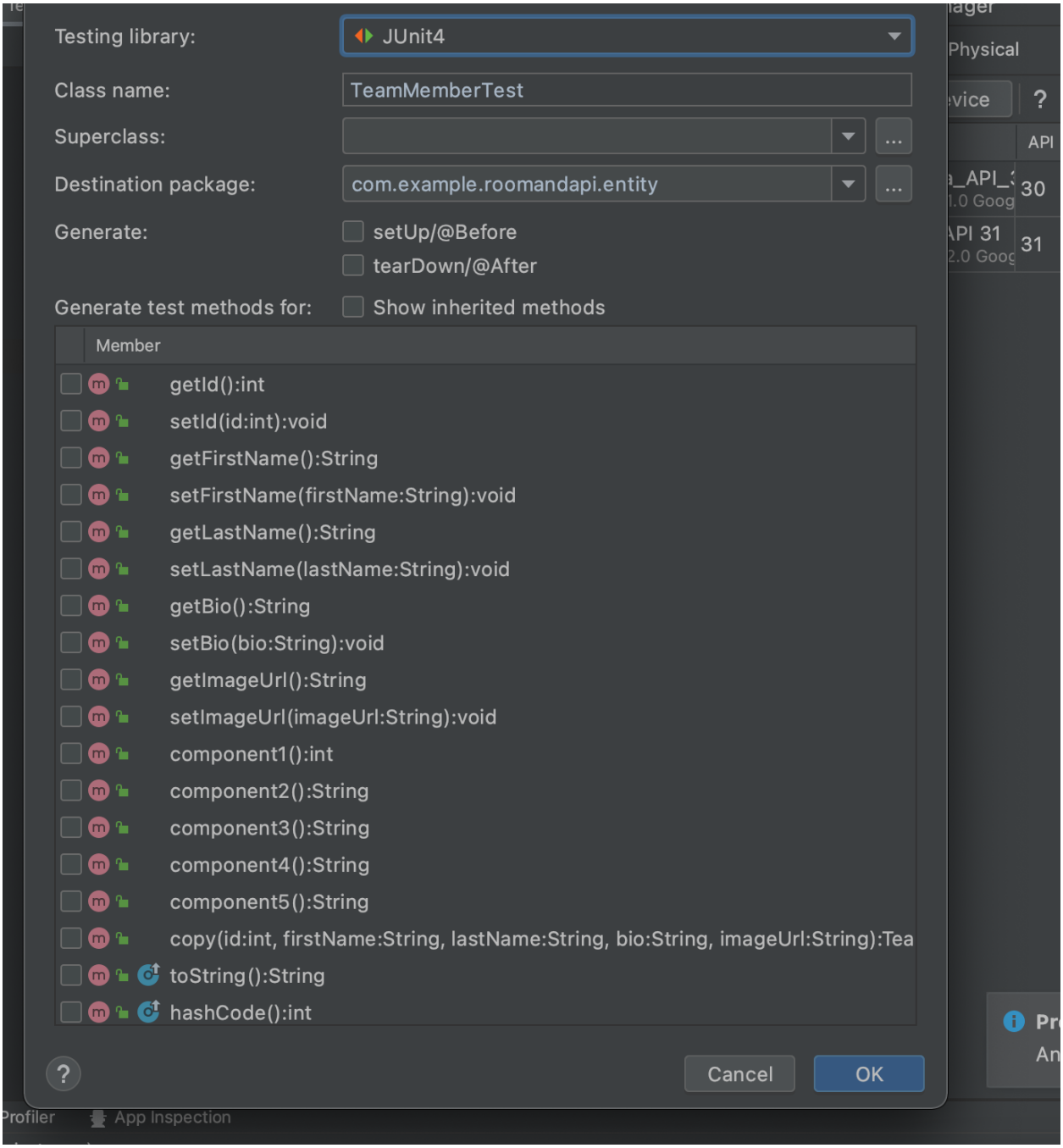

Figure 3: Step 3, Select the version of JUnit to use

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.

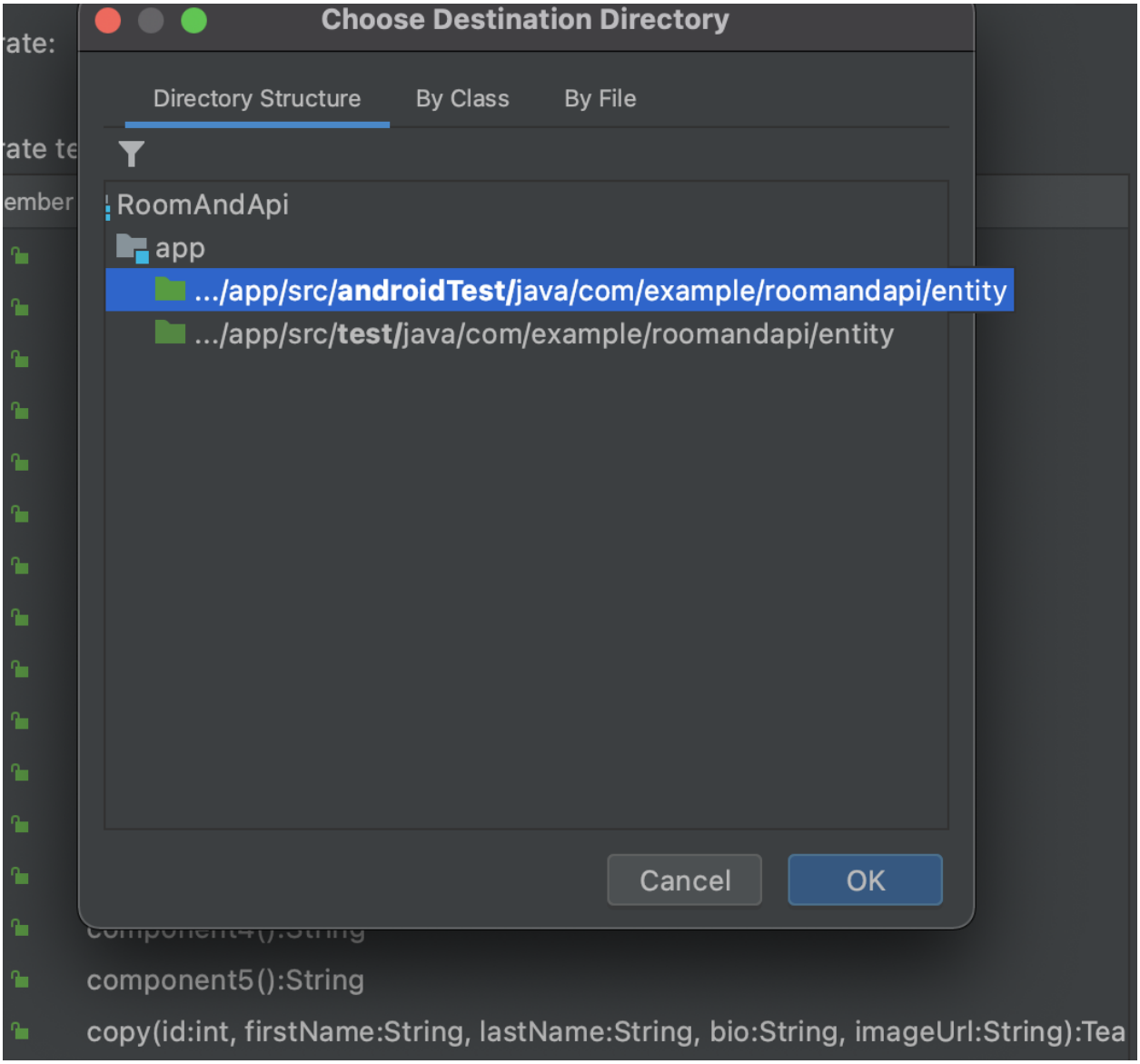

Figure 4: Step 4, Determine the type of test

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

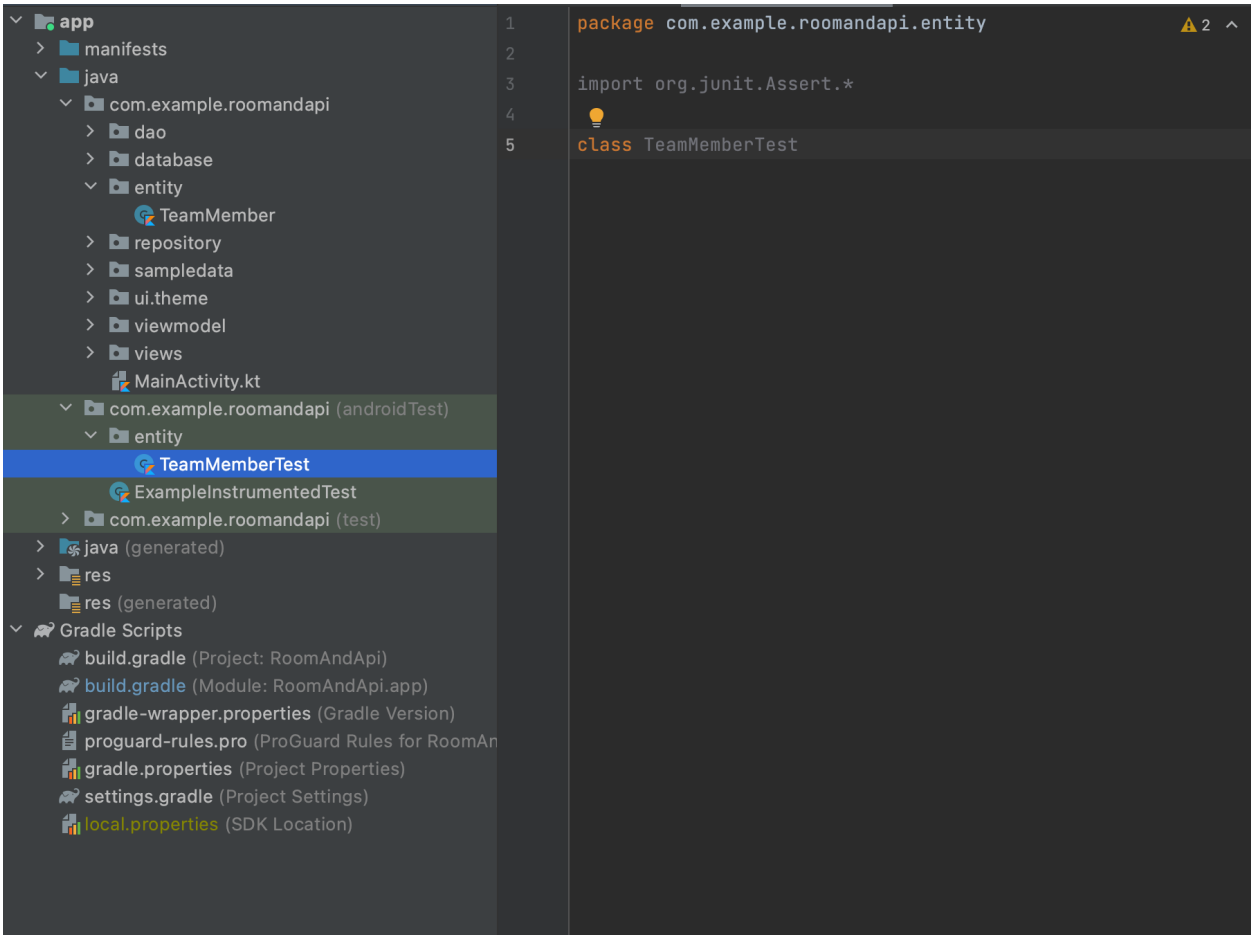

Figure 5: Auto-generated Test Class

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 abstractions 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.")
        )
        dao.insert()
        val byName = 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 [**?**]. 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 [**?**].

```
1  package com.example.roomandapi.entity
2
3  import android.content.Context
4  import androidx.room.Room
5  import androidx.test.core.app.ApplicationProvider
6  import com.example.roomandapi.dao.TeamMemberDao
7  import com.example.roomandapi.database.WCDatabase
8  import junit.framework.TestCase
9  import kotlinx.coroutines.runBlocking
10 import org.hamcrest.CoreMatchers
11 import org.hamcrest.MatcherAssert
12 import org.junit.After
13 import org.junit.Assert.*
14 import org.junit.Before
15 import org.junit.Test
16
17 class TeamMemberTest : TestCase(){
18     private lateinit var db: WCDatabase
19     private lateinit var dao: TeamMemberDao
20
21     @Before
22     public override fun setUp(){
23         val context = ApplicationProvider.getApplicationContext<Context
       >()
24         db = Room.inMemoryDatabaseBuilder(context, WCDatabase::class.
       java).build()
25         dao = db.teamMemberDao()
26     }
27
28     @After
29     fun closeDB(){
30         db.close()
31     }
32
33     @Test
34     fun TestWriteAndReadUser() = runBlocking(){
35         val members: List<TeamMember> =  listOf(
36             TeamMember(1,
37             "Jacob",
38             "Conner",
39             "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.")
40         )
41         dao.insert()
```

```
42          val byName = dao.getByName("Jacob", "Conner")
43          MatcherAssert.assertThat(byName, CoreMatchers.equalTo(members
    [0]))
44      }
45  }
```

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 [**?**] [**?**].

```
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.fillMaxSize
8  import androidx.compose.material.Button
9  import androidx.compose.material.Text
10 import androidx.compose.runtime.Composable
11 import androidx.compose.runtime.livedata.observeAsState
12 import androidx.compose.ui.platform.LocalContext
13 import androidx.compose.ui.Alignment
14 import androidx.compose.ui.Modifier
15 import androidx.compose.ui.graphics.Color
16 import androidx.compose.ui.text.font.FontFamily
17 import androidx.compose.ui.text.font.FontWeight
18 import androidx.compose.ui.text.style.TextDecoration
19 import androidx.compose.ui.unit.sp
20 import androidx.lifecycle.viewmodel.compose.viewModel
21 import androidx.navigation.NavController
22 import androidx.navigation.compose.NavHost
23 import androidx.navigation.compose.composable
24 import androidx.navigation.compose.rememberNavController
25
26
27 import com.example.roomandapi.entity.TeamMember
28
```

```
@Composable
fun appNavController() {

    val navController = rememberNavController()
    NavHost(navController, startDestination = "main") {
        composable(route = "main") {
            MainView(navController)
        }
        composable(route = "about") {
            AboutView(navController)
        }
    }
}


@Composable
fun MainView(navController: NavController) {
        Column(){
            Button(onClick = { navController.navigate("about") }) {
                Text("About Us")
            }
            Column(
                modifier = Modifier
                    .fillMaxSize()
                    .background(Color.White),
                horizontalAlignment = Alignment.CenterHorizontally,
                verticalArrangement = Arrangement.Center
            ) {
                Text(
                    text = "Workout Companion",
                    fontSize = 30.sp,
                    fontWeight = FontWeight.Bold,
                    textDecoration = TextDecoration.Underline,
                    fontFamily = FontFamily.Serif
                )
            }
        }
    }
```

Listing 18: Example Activty 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

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

Listing 20: Button to navigate to a view

6 Documenting with Dokka

test

List of Figures

List of Tables

Listings