Room Database

This lesson will introduce how to use Room database to store and access blog articles.

WE'LL COVER THE FOLLOWING

Dependencies

To use the *Room* database, we need to add it to our dependencies list:

```
dependencies {
    implementation 'androidx.appcompat:appcompat:1.1.0'
    implementation 'androidx.constraintlayout:constraintlayout:1.1.3'
    implementation 'androidx.swiperefreshlayout:swiperefreshlayout:1.0.0'
    implementation 'com.google.android.material:material:1.1.0-alpha10'
    implementation 'com.github.bumptech.glide:glide:4.10.0'
    implementation 'com.squareup.okhttp3:okhttp:4.2.1'
    implementation 'com.google.code.gson:gson:2.8.6'

    def room_version = "2.2.3"

    implementation 'androidx.room:room-runtime:$room_version'
    annotationProcessor 'androidx.room:room-compiler:$room_version'
}
```

build.gradle

Because *Room* database heavily relies on custom annotations, we also added *Room* `annotationProcessor` dependency.

Entities

Now, we need to tell *Room* what entities we want to save to the database. Let's open the `Blog` class and add the `@Entity` annotation *(1)*. Doing so, we tell the

Room to create a table for the blog entity.

To define a primary key we can simply use `@PrimaryKey` annotation on the `id` field *(2)*. While the *Room* library can automatically persist all *Java* primitives, it can't persist custom objects, like `Author`. To save the `Author` object, we have two options:

- create a table for `Author` object and another table to link `Author` and `Blog` data
- embed all fields from the `Author` object into the `Blog` object on a database table level

We are going to proceed with the second option since the `Author` object doesn't have a lot of fields. To embed the `Author` object fields, we can simply use `@Embedded` annotation on the `author` field *(3)*.

Generally for *Room* to function properly all entity fields must be public, have setters or public constructors *(4)* so the *Room* can instantiate the object properly.

```
@Entity // 1
public class Blog implements Parcelable {

    @PrimaryKey // 2
    private int id;
    @Embedded // 3
    private Author author;
    private String title;
    private String date;
    private String image;
    private String description;
    private int views;
    private float rating;

    public Blog(int id, Author author, String title, String date, String image,
                String description, int views, float rating) { // 4
        this.id = id;
        this.author = author;
        this.title = title;
        this.date = date;
        this.image = image;
        this.description = description;
        this.views = views;
        this.rating = rating;
    }
    ...
}
```

DAO

DAO

Now that we have defined entities, it’s time to create a data access object (DAO).

Because we are using *Room* library, we don’t need to define the logic for our DAO object, instead, we will define the interface, and *Room* will generate the implementation for us.

Start by creating a new package `com.travelblog.database` and interface `BlogDAO`. To indicate that this is a *Room* data access object, add a `@Dao` annotation *(1)*.

```
@Dao
public interface BlogDAO
```

We are going to need 3 methods:

- get all blog articles *(1)*
- save blog articles *(2)*
- delete all articles *(3)*

```
@Dao
public interface BlogDAO {

    List<Blog> getAll(); // 1

    void insertAll(List<Blog> blogList); // 2

    void deleteAll(); // 3
}
```

The method signature is not enough for *Room* to generate the DAO implementation; we need to use appropriate annotation along with parameters:

- To execute an SQL *select* query, we can use `@Query` annotation with SQL query as a parameter `SELECT * FROM blog` which is going to select all data from the `blog` table *(1)*.
- To execute an SQL *insert* query, we can simply use `@Insert` annotation. *Room* is smart enough to infer everything else from the method parameters *(2)*.

- To execute SQL *delete* query, we can use `@Query` annotation with SQL query as a parameter `DELETE FROM blog` which is going to delete all data from the `blog` table *(3)*.

```
@Dao
public interface BlogDAO {

    @Query("SELECT * FROM blog") // 1
    List<Blog> getAll();

    @Insert // 2
    void insertAll(List<Blog> blogList);

    @Query("DELETE FROM blog") // 3
    void deleteAll();
}
```

The generated implementation will be automatically added to the sources during the build time. Here is a sneak peek to the generated class; imagine how much time it would take to write all this boilerplate code.

```
public final class BlogDAO_Impl implements BlogDAO {
  private final RoomDatabase __db;

  private final EntityInsertionAdapter<Blog> __insertionAdapterOfBlog;

  private final SharedSQLiteStatement __preparedStmtOfDeleteAll;

  public BlogDAO_Impl(RoomDatabase __db) {
    this.__db = __db;
    this.__insertionAdapterOfBlog = new EntityInsertionAdapter<Blog>(__db) {
      @Override
      public String createQuery() {
        return "INSERT OR ABORT INTO `Blog` (`id`,`title`,`date`,`image`,`description`,`views
      }

      @Override
      public void bind(SupportSQLiteStatement stmt, Blog value) {
        stmt.bindLong(1, value.getId());
        if (value.getTitle() == null) {
          stmt.bindNull(2);
        } else {
          stmt.bindString(2, value.getTitle());
        }
        if (value.getDate() == null) {
          stmt.bindNull(3);
        } else {
          stmt.bindString(3, value.getDate());
        }
        if (value.getImage() == null) {
          stmt.bindNull(4);
        } else {
          stmt.bindString(4, value.getImage());
        }

        if (value.getDescription() == null) {
          stmt.bindNull(5);
        } else {
          stmt.bindString(5, value.getDescription());
        }
        stmt.bindLong(6, value.getViews());
        stmt.bindDouble(7, value.getRating());
        final Author _tmpAuthor = value.getAuthor();
        if(_tmpAuthor != null) {
          if (_tmpAuthor.getName() == null) {
            stmt.bindNull(8);
          } else {
            stmt.bindString(8, _tmpAuthor.getName());
          }
          if (_tmpAuthor.getAvatar() == null) {
            stmt.bindNull(9);
          } else {
            stmt.bindString(9, _tmpAuthor.getAvatar());
          }
        } else {
          stmt.bindNull(8);
          stmt.bindNull(9);
        }
      }
    };
    this.__preparedStmtOfDeleteAll = new SharedSQLiteStatement(__db) {
      @Override
      public String createQuery() {
        final String _query = "DELETE FROM blog";
        return _query;
      }
    };
  }

  @Override
  public void insertAll(final List<Blog> blogList) {
    __db.assertNotSuspendingTransaction();
    __db.beginTransaction();
    try {
      __insertionAdapterOfBlog.insert(blogList);
      __db.setTransactionSuccessful();
    } finally {
      __db.endTransaction();
    }
  }

  @Override
  public void deleteAll() {
    __db.assertNotSuspendingTransaction();
    final SupportSQLiteStatement _stmt = __preparedStmtOfDeleteAll.acquire();
    __db.beginTransaction();
    try {
      _stmt.executeUpdateDelete();
      __db.setTransactionSuccessful();
    } finally {
      __db.endTransaction();
      __preparedStmtOfDeleteAll.release(_stmt);
    }
  }
  }

  @Override
  public List<Blog> getAll() {
    final String _sql = "SELECT * FROM blog";
    final RoomSQLiteQuery _statement = RoomSQLiteQuery.acquire(_sql, 0);
    __db.assertNotSuspendingTransaction();
    final Cursor _cursor = DBUtil.query(__db, _statement, false, null);
    try {
      final int _cursorIndexOfId = CursorUtil.getColumnIndexOrThrow(_cursor, "id");
      final int _cursorIndexOfTitle = CursorUtil.getColumnIndexOrThrow(_cursor, "title");
      final int _cursorIndexOfDate = CursorUtil.getColumnIndexOrThrow(_cursor, "date");
      final int _cursorIndexOfImage = CursorUtil.getColumnIndexOrThrow(_cursor, "image");
      final int _cursorIndexOfDescription = CursorUtil.getColumnIndexOrThrow(_cursor, "descri
      final int _cursorIndexOfViews = CursorUtil.getColumnIndexOrThrow(_cursor, "views");
      final int _cursorIndexOfRating = CursorUtil.getColumnIndexOrThrow(_cursor, "rating");
      final int _cursorIndexOfName = CursorUtil.getColumnIndexOrThrow(_cursor, "name");
      final int _cursorIndexOfAvatar = CursorUtil.getColumnIndexOrThrow(_cursor, "avatar");
      final List<Blog> _result = new ArrayList<Blog>(_cursor.getCount());
      while(_cursor.moveToNext()) {
        final Blog _item;
        final int _tmpId;
        _tmpId = _cursor.getInt(_cursorIndexOfId);
        final String _tmpTitle;
        _tmpTitle = _cursor.getString(_cursorIndexOfTitle);
        final String _tmpDate;
        _tmpDate = _cursor.getString(_cursorIndexOfDate);
        final String _tmpImage;
        _tmpImage = _cursor.getString(_cursorIndexOfImage);
        final String _tmpDescription;
        _tmpDescription = _cursor.getString(_cursorIndexOfDescription);
        final int _tmpViews;
        _tmpViews = _cursor.getInt(_cursorIndexOfViews);
        final float _tmpRating;
        _tmpRating = _cursor.getFloat(_cursorIndexOfRating);
        final Author _tmpAuthor;
        if (! (_cursor.isNull(_cursorIndexOfName) && _cursor.isNull(_cursorIndexOfAvatar))) {
          final String _tmpName;
          _tmpName = _cursor.getString(_cursorIndexOfName);
          final String _tmpAvatar;
          _tmpAvatar = _cursor.getString(_cursorIndexOfAvatar);
          _tmpAuthor = new Author(_tmpName,_tmpAvatar);
        }  else  {
          _tmpAuthor = null;
        }
        _item = new Blog(_tmpId,_tmpAuthor,_tmpTitle,_tmpDate,_tmpImage,_tmpDescription,_tmpV
        _result.add(_item);
      }
      return _result;
    } finally {
      _cursor.close();
      _statement.release();
    }
  }
}
```

Database

Finally, when entities and DAO are defined, we can create a database object. Create an abstract `AppDatabase` class in the `com.travelblog.database` and make it extend `RoomDatabase`.

```
public abstract class AppDatabase extends RoomDatabase
```

The *Room* library will generate the implementation for us, similarly to the DAO interface.

Now, add a `@Database` annotation with parameters to specify database entities and version.

```
@Database(entities = {Blog.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase
```

Next, define the abstract method, which is going to return the `BlogDAO` object.

```
@Database(entities = {Blog.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract BlogDAO blogDao();
}
```

Here is what the generated class looks like.

```
public final class AppDatabase_Impl extends AppDatabase {
  private volatile BlogDAO _blogDAO;

  @Override
  protected SupportSQLiteOpenHelper createOpenHelper(DatabaseConfiguration configuration) {
    final SupportSQLiteOpenHelper.Callback _openCallback = new RoomOpenHelper(configuration,
      @Override
      public void createAllTables(SupportSQLiteDatabase _db) {
        _db.execSQL("CREATE TABLE IF NOT EXISTS `Blog` (`id` INTEGER NOT NULL, `title` TEXT,
        _db.execSQL("CREATE TABLE IF NOT EXISTS room_master_table (id INTEGER PRIMARY KEY,ide
        _db.execSQL("INSERT OR REPLACE INTO room_master_table (id,identity_hash) VALUES(42, '
      }

      @Override
      public void dropAllTables(SupportSQLiteDatabase _db) {
        _db.execSQL("DROP TABLE IF EXISTS `Blog`");
        if (mCallbacks != null) {
          for (int _i = 0, _size = mCallbacks.size(); _i < _size; _i++) {
            mCallbacks.get(_i).onDestructiveMigration(_db);
          }
        }
      }

      @Override
      protected void onCreate(SupportSQLiteDatabase _db) {
        if (mCallbacks != null) {
          for (int _i = 0, _size = mCallbacks.size(); _i < _size; _i++) {
```

```
          mCallbacks.get(_i).onCreate(_db);
        }
      }

    }

    @Override
    public void onOpen(SupportSQLiteDatabase _db) {
      mDatabase = _db;
      internalInitInvalidationTracker(_db);
      if (mCallbacks != null) {
        for (int _i = 0, _size = mCallbacks.size(); _i < _size; _i++) {
          mCallbacks.get(_i).onOpen(_db);
        }
      }
    }

    @Override
    public void onPreMigrate(SupportSQLiteDatabase _db) {
      DBUtil.dropFtsSyncTriggers(_db);
    }

    @Override
    public void onPostMigrate(SupportSQLiteDatabase _db) {
    }

    @Override
    protected RoomOpenHelper.ValidationResult onValidateSchema(SupportSQLiteDatabase _db) {
      final HashMap<String, TableInfo.Column> _columnsBlog = new HashMap<String, TableInfo.
      _columnsBlog.put("id", new TableInfo.Column("id", "INTEGER", true, 1, null, TableInfo
      _columnsBlog.put("title", new TableInfo.Column("title", "TEXT", false, 0, null, Table
      _columnsBlog.put("date", new TableInfo.Column("date", "TEXT", false, 0, null, TableIn
      _columnsBlog.put("image", new TableInfo.Column("image", "TEXT", false, 0, null, Table
      _columnsBlog.put("description", new TableInfo.Column("description", "TEXT", false, 0,
      _columnsBlog.put("views", new TableInfo.Column("views", "INTEGER", true, 0, null, Tab
      _columnsBlog.put("rating", new TableInfo.Column("rating", "REAL", true, 0, null, Tabl
      _columnsBlog.put("name", new TableInfo.Column("name", "TEXT", false, 0, null, TableIn
      _columnsBlog.put("avatar", new TableInfo.Column("avatar", "TEXT", false, 0, null, Tab
      final HashSet<TableInfo.ForeignKey> _foreignKeysBlog = new HashSet<TableInfo.ForeignK
      final HashSet<TableInfo.Index> _indicesBlog = new HashSet<TableInfo.Index>(0);
      final TableInfo _infoBlog = new TableInfo("Blog", _columnsBlog, _foreignKeysBlog, _in
      final TableInfo _existingBlog = TableInfo.read(_db, "Blog");
      if (! _infoBlog.equals(_existingBlog)) {
        return new RoomOpenHelper.ValidationResult(false, "Blog(com.travelblog.http.Blog).\
                + " Expected:\n" + _infoBlog + "\n"
                + " Found:\n" + _existingBlog);
      }
      return new RoomOpenHelper.ValidationResult(true, null);
    }
  }, "a9fcc6cf6ae770a49c551e62f7bc543a", "756582c5cdabe52608640da2e036e488");
  final SupportSQLiteOpenHelper.Configuration _sqliteConfig = SupportSQLiteOpenHelper.Confi
      .name(configuration.name)
      .callback(_openCallback)
      .build();
  final SupportSQLiteOpenHelper _helper = configuration.sqliteOpenHelperFactory.create(_sql
  return _helper;
}

@Override
protected InvalidationTracker createInvalidationTracker() {
  final HashMap<String, String> _shadowTablesMap = new HashMap<String, String>(0);
  HashMap<String, Set<String>> _viewTables = new HashMap<String, Set<String>>(0);
  return new InvalidationTracker(this, _shadowTablesMap, _viewTables, "Blog");
  }

  @Override
  public void clearAllTables() {
    super.assertNotMainThread();
    final SupportSQLiteDatabase _db = super.getOpenHelper().getWritableDatabase();
    try {
      super.beginTransaction();
      _db.execSQL("DELETE FROM `Blog`");
      super.setTransactionSuccessful();
    } finally {
      super.endTransaction();
      _db.query("PRAGMA wal_checkpoint(FULL)").close();
      if (!_db.inTransaction()) {
        _db.execSQL("VACUUM");
      }
    }
  }

  @Override
  public BlogDAO blogDao() {
    if (_blogDAO != null) {
      return _blogDAO;
    } else {
      synchronized(this) {
        if(_blogDAO == null) {
          _blogDAO = new BlogDAO_Impl(this);
        }
        return _blogDAO;
      }
    }
  }
}
```

Now our database can be used to store or retrieve the data:

- use the `Room.*databaseBuilder*` to create the `AppDatabase` object *(1)*
- use the `blogDao` method to access the `BlogDAO` object *(2)*
- use the `BlogDAO` methods to retrieve/store/remove the data *(3)*

```
AppDatabase database =
      Room.databaseBuilder(context, AppDatabase.class, "blog-database").build(); // 1
BlogDAO dao = database.blogDao(); // 2
List<Blog> blogList = dao.getAll(); // 3
```

Creating the database object is very expensive; so that’s why it is a common practice to make it a singleton.

```
public class DatabaseProvider {

    private static volatile AppDatabase instance;

    public static AppDatabase getInstance(Context context) {
```

```
    public static AppDatabase getInstance(Context context) {
        if (instance == null) {
            synchronized (DatabaseProvider.class) {
                if (instance == null) {
                    instance = Room
                            .databaseBuilder(context, AppDatabase.class, "blog-database")
                            .build();
                }
            }
        }
        return instance;
    }
}
```

The next lesson will show the power of the Repository pattern.