## 

[**Motivation**](#_95v05ri21wy8) **2**

[**Entity Classes (@Entity)**](#_zi0b8a2rsoj6) **4**

[Primary Key](#_plc5tqwjod18) 5

[Indices & Uniqueness](#_5eu2c2omqh2s) 6

[Relationships](#_wywclvq0ouxz) 7

[@Embedded - Nested Objects](#_jupci7e0cboc) 8

[**Data Access Objects (@Dao)**](#_6lk39ihiit9d) **9**

[@Insert](#_p28iqumjpqj4) 9

[@Update](#_83yajan6eao) 9

[@Delete](#_9x380yst776) 10

[@Query](#_wew7u6stt1vg) 10

[Simple Queries](#_8dlbgft8d1h0) 10

[Passing Parameters into the Query](#_i2elrpnyzexi) 11

[Returning Subset of Columns](#_lj2e9y1v4my1) 11

[Passing a Collection of Arguments](#_8wsi24je3pf6) 12

[Observable Queries](#_tvi8bz11r776) 12

[RxJava](#_tjmaoa58xk24) 13

[Direct Cursor Access](#_kd0id3nq2gbq) 13

[Multiple Table Queries](#_f8phiy5fsjic) 13

[**Using TypeConverters for Custom Types**](#_ntat39ifbogi) **14**

[**Migrations**](#_axalj0o8fexm) **16**

[Testing Migrations](#_ab93tai3o3xt) 17

[Exporting Schema](#_3vt7jehn5now) 17

[**Testing**](#_wzm3z5vuzg8q) **18**

[JUnit Test on Your Host Machine](#_ynahqr3c5kbx) 19

[JUnit Test on an Android Device](#_17a840f7oj1p) 19

[**Best Practices**](#_xrojg5i3uf0e) **20**

[**Addendum**](#_37tk23wfep93) **20**

[What is Wrong With Object References Between Entities](#_dhqegwlinp1z) 20

## 

## Motivation

Apps that handle non-trivial amounts of structured data can benefit greatly from persisting that data locally. The most common use case is to cache relevant pieces of data. That way,when the device cannot access the network, the user can still browse that content while they are offline. Any user-initiated content changes are then synced to the server after the device is back online.

The core framework provides [built-in support](https://developer.android.com/training/basics/data-storage/databases.html) for working with raw SQL content. Although these APIs are powerful, they are fairly low-level and require a great deal of time and effort to use:

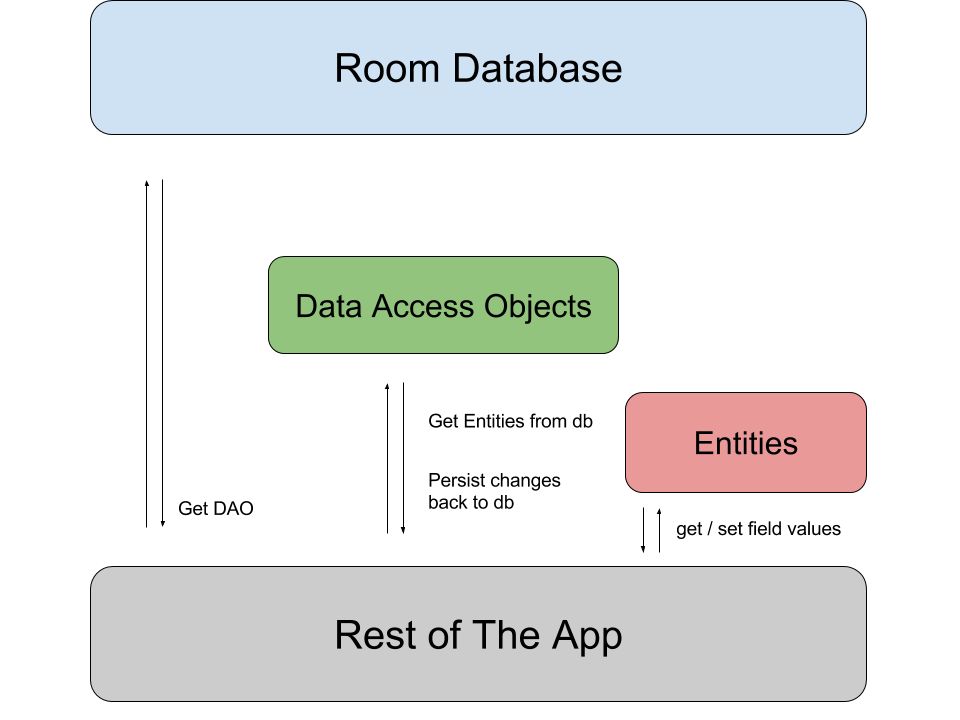
* There is no compile-time verification of raw SQL queries. As your data graph changes, you need to update the affected SQL queries manually. This process can be time consuming and error prone.
* You need to use lots of boilerplate code to convert between SQL queries and Java data objects.

Room provides a nice abstraction over SQLite to allow fluent database access while harnessing the full power of SQLite.

There are 3 major components in Room:

* **@Database:** You can use this annotation to mark a class as a database holder. The annotation defines the list of entities, and the class's content defines the list of data access objects (DAOs) in the database. It is also the main access point for the underlying connection.  
  The annotated class should be an abstract class that extends RoomDatabase. At runtime, you can acquire an instance of it by calling Room.databaseBuilder() or Room.inMemoryDatabaseBuilder().
* **@Entity:** This annotation marks a class that holds a database row. For each entity, a database table is created to hold the items. You must reference the entity class through the Database#entities array. Each field of the entity is persisted in the database unless you annotate it with @Ignore.
* **Note:** Entities can have either a public constructor (if the DAO class can access each persisted field) or a set of arguments whose types and names match the fields. Room can also use full or partial constructors, such as a constructor which receives only some of the fields.
* **@Dao:** This annotation marks a class or interface as a [Data Access Object](https://en.wikipedia.org/wiki/Data_access_object) [(](https://en.wikipedia.org/wiki/Data_access_object)D[A](https://en.wikipedia.org/wiki/Data_access_object)O[)](https://en.wikipedia.org/wiki/Data_access_object). DAOs are the main component of Room and are responsible for defining the methods that access the database. The class that is annotated with @Database must contain an abstract method that has 0 arguments and returns the class that is annotated with @Dao. When generating the code at compile time, Room creates an implementation of this class.

**Important:** By accessing a databaseusing a DAO class instead of query builders or direct queries, you can separate different components of your database architecture. Furthermore, DAOs allow you to easily mock database access as you test your app.



The following code snippet contains a sample database configuration with 1 entity and 1 DAO.

|  |
| --- |
| // File: User.java @Entity public class User {  @PrimaryKey  private int uid;  @ColumnInfo(name = "first\_name")  private String firstName;  @ColumnInfo(name = "last\_name")  private String lastName;  // Getters and setters are ignored for brevity, but they're required for Room to work. }  // File: UserDao.java @Dao public interface UserDao {  @Query("SELECT \* FROM user")  List<User> getAll();  @Query("SELECT \* FROM user WHERE uid IN (:userIds)")  List<User> loadAllByIds(int[] userIds);  @Query("SELECT \* FROM user where first\_name LIKE :first AND last\_name LIKE :last LIMIT 1")  User findByName(String first, String last);  @Insert  void insertAll(User... users);  @Delete  void delete(User user); }  // File: AppDatabase.java @Database(entities = {User.class}, version = 1) public abstract class AppDatabase extends RoomDatabase {  public abstract UserDao userDao(); } |

After you create the files above, you get an instance of the created database using the following code:

|  |
| --- |
| AppDatabase db = Room  .databaseBuilder(getApplicationContext(), AppDatabase.class, "database-name")  .build(); |

**Note:** You should follow the singleton design pattern when instantiating an AppDatabase object, as each Database instance is fairly expensive, and you rarely need access to multiple instances.

## Entity Classes (@Entity)

When a class is annotated with *@Entity* and is referenced in the *entities* property of a *@Database* annotation, Room creates a database table for that *entitiy* in the database.

By default, Room creates a column for each field that's defined in the *entity.* If an *entity* has fields that you don't want to persist, you can annotate them using *@Ignore*, as shown in the following code snippet:

|  |
| --- |
| @Entity class User {  @PrimaryKey  public int id;  public String firstName;  public String lastName;  @Ignore  Bitmap picture; } |

To persist a field, Room must have access to it. You can make a field *public*, or you can provide a *setter* and *getter* for it. If you use setter and getter methods, keep in mind that they're based on Java Beans conventions in Room.

### Primary Key

Each entity must define at least 1 field as a primary key. Even when there is only 1 field, you still need to annotate the field with *@Primary* key annotation. Also, if you want Room to assign automatic IDs to entities, you can set the @PrimaryKey’s *autoGenerate* property. If the Entity has a composite primary key, you can use the *primaryKeys* property of the *@Entity* annotation.

|  |
| --- |
| @Entity(primaryKeys = {"firstName", "lastName"}) class User {  public String firstName;  public String lastName;  @Ignore  Bitmap picture; } |

By default, Room uses the class name as the database table name. If you want the table to have a different name, set the *tableName* property of the *@Entity* annotation, as shown in the following code snippet:

|  |
| --- |
| @Entity(tableName = "users") class User {  ... } |

**Caution:** Table names in SQLite are case insensitive.

Similar to the *tablename* property, Room uses the field names as the column names in the database. If you want a column to have a different name, add the *@ColumnInfo* annotation to a field, as shown in the following code snippet:

|  |
| --- |
| @Entity(tableName = "users") class User {  @PrimaryKey  public int id;  @ColumnInfo(name = "first\_name")  public String firstName;  @ColumnInfo(name = "last\_name")  public String lastName;  @Ignore  Bitmap picture; } |

### Indices & Uniqueness

Depending on how you access the data, you might want to index certain fields in the database to speed up your queries. To add indices to an *entity*, include the *indices* property within the *@Entity* annotation, listing the names of the columns that you want to include in the index or composite index. The following code snippet demonstrates this annotation process:

|  |
| --- |
| @Entity(indices = {@Index("name"), @Index("last\_name", “address”)}) class User {  @PrimaryKey  public int id;  public String firstName;  public String address;  @ColumnInfo(name = "last\_name")  public String lastName;  @Ignore  Bitmap picture; } |

Sometimes, certain fields or groups of fields in a database must be unique . You can enforce this uniqueness property by setting the unique property of an @Index annotation to true. The following code sample prevents a table from having two rows that contain the same set of values for the *firstName* and *lastName* columns:

|  |
| --- |
| @Entity(indices = {@Index(value = {"first\_name", "last\_name"}, unique = true)}) class User {  @PrimaryKey  public int id;  @ColumnInfo(name = "first\_name")  public String firstName;  @ColumnInfo(name = "last\_name")  public String lastName;  @Ignore  Bitmap picture; } |

#### 

### Relationships

Because SQLite is a relational database, you can specify relationships between objects. Even though most ORM libraries allow entity objects to reference each other, Room explicitly forbids this. For more details, see the [Removing Object References between Entities](#_dhqegwlinp1z) section.

Even though you cannot use direct relationships, Room still allows you to define [Foreign Key](https://sqlite.org/foreignkeys.html) constraints between entities.

For example, if there's another entity called Book, you can define its relationship to the User entity using the @ForeignKey annotation, as shown in the following code snippet:

|  |
| --- |
| @Entity(foreignKeys = @ForeignKey(  entity = User.class,  parentColumns = "id",  childColumns = "user\_id")) class Book {  @PrimaryKey  public int bookId;  public String title;  @ColumnInfo(name = "user\_id")  public int userId; } |

Foreign Keys are very powerful, as they allow you to specify what occurs when the referenced entity is updated. For instance, you can tell SQLite to delete all books for a user if the User is deleted by including *onDelete = CASCADE* in the *@ForeignKey* annotation.

**Note:** SQLite handles @Insert(onConflict=REPLACE) as a set of REMOVE and REPLACE operations instead of a single UPDATE operation. This method of replacing conflicting values could affect your foreign key constraints. For more details, see the [SQLite documentation](https://sqlite.org/lang_conflict.html) for the ON\_CONFLICT clause.

#### @Embedded - Nested Objects

Sometimes, you'd like to express an entity or plain old Java object (POJOs) as a cohesive whole in your database logic, even if the object contains several fields. In these situations, you can use the *@Embedded* annotation to represent an object that you'd like to decompose into its subfields within a table. You can then query the embedded fields just as you would for other individual columns.

For instance, our *User* class can include a field of type Address, which represents a composition of fields named street, city, state, and postCode. To store the composed columns separately in the table, include an Address field in the User class that is annotated with *@Embedded*, as shown in the following code snippet:

|  |
| --- |
| class Address {  public String street;  public String state;  public String city;  @ColumnInfo(name = "post\_code")  public int postCode; }  @Entity class User {  @PrimaryKey  public int id;  public String firstName;  @Embedded  public Address address; } |

The table representing a User object then contains the following columns: *id, firstName, street, state, city,* and *post\_code.*

**Note:** Embedded fields can also include other embedded fields.

If an entity has multiple embedded fields of the same type, you can keep each column unique by setting the *prefix* property. Room then adds the provided value to the beginning of each column name in the embedded object.

## Data Access Objects (@Dao)

The main component in Room is the *@Dao* class. DAOs abstract access to the database access in a clean way.

There are multiple queries you can represent using a DAO class This document includes several common examples.

### @Insert

When you create a DAO method and annotate it with *@Insert*, Room generates an implementation that inserts all parameters into the database in a single transaction.

The following code snippet shows several example queries:

|  |
| --- |
| @Dao public interface MyDao {  @Insert(onConflict = OnConflictStrategy.REPLACE)  public void insertUsers(User... users);  @Insert  public void insertBothUsers(User user1, User user2);  @Insert  public void insertUsersAndFriends(User user, List<User> friends); } |

If the @Insert method receives only 1 parameter, it can return a *long* which is the new [rowId](https://sqlite.org/lang_createtable.html#rowid) for the inserted item. If the parameter is an array or a collection, it can also return *long[]* or *List<Long>.*

For more details, see the reference documentation for the @Dao annotation.

### @Update

@Update is a convenience method that updates a set of entities, given as parameters, in the database. It uses a query that matches against the primary key of each entity.

|  |
| --- |
| @Dao public interface MyDao {  @Update  public void updateUsers(User... users); } |

Although usually not necessary, you can have this method return an *int* instead. The return value indicates the number of rows updated in the database.

### @Delete

A convenience API that deletes a set of entities, given as parameters, from the database. It uses the primary keys to find the entities to delete.

|  |
| --- |
| @Dao public interface MyDao {  @Delete  public void deleteUsers(User... users); } |

Although usually not necessary, you can have this method return an *int* instead. The return value indicates the number of rows removed from the database.

### @Query

This is the main annotation used in DAO classes. It allows you to perform read/write operations on a database. Each *@Query* method is verified at compile time, so if there is a problem with the query, a compilation error occurs instead of a runtime failure.

Room also verifies the return value of the query such that if the name of the field in the returned object doesn't match the corresponding column names in the query response, Room alerts you in one of the following two ways:

* It gives a warning if only some field names match.
* It gives an error if no field names match.

#### Simple Queries

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT \* FROM user")  public User[] loadAllUsers(); } |

This is a very simple query that loads all users. At compile time, Room knows that it is querying all columns in the **user** table. If the query contains a syntax error, or if the **user** table doesn't exist in the database, Room displays an error with the appropriate message as your app compiles.

#### Passing Parameters into the Query

Most of the time, you need to pass parameters into queries to perform filtering operations, such as displaying only users who are older than a certain age. To accomplish this task, use method parameters in your Room annotation, as shown in the following code snippet:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT \* FROM user WHERE age > :minAge")  public User[] loadAllUsersOlderThan(int minAge); } |

When this query is processed at compile time, Room matches the *:minAge* bind parameter with the *minAge* method parameter. Room performs the match using the parameter names. If there is a mismatch, an error occurs as your app compiles.

You can also pass multiple parameters or reference them multiple times in a query, as shown in the following code snippet:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT \* FROM user WHERE age BETWEEN :minAge AND :maxAge")  public User[] loadAllUsersBetweenAges(int minAge, int maxAge);   @Query("SELECT \* FROM user WHERE first\_name LIKE :search OR last\_name LIKE :search")  public List<User> findUserWithName(String search); } |

#### Returning Subset of Columns

Most of the time, you need to get only a few fields of an entity. For example, your UI might display just a user's first name and last name, rather than every detail about the user. By fetching only the columns that appear in your app's UI, you save valuable resources, and your query completes more quickly.

Room allows you to return any Java object from your queries as long as the set of result columns can be mapped into the returned object. For example, you can create the following POJO to fetch the user's first name and last name:

|  |
| --- |
| public class NameTuple {  @ColumnInfo(name="first\_name")  public String firstName;  @ColumnInfo(name="last\_name")  public String lastName; } |

Now, you can use this POJO in your query method:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT first\_name, last\_name FROM user")  public List<NameTuple> loadFullName(); } |

Room understands that the query returns *first\_name* and *last\_name* columns and that these values can be mapped into the fields of the *NameTuple* class. Therefore, Room can generate the proper code. If the query returns too many columns, or a column that doesn't exist in *NameTuple* class, Room displays a warning.

**Note:** These POJOs can also use the *@Embedded* annotation.

#### Passing a Collection of Arguments

Some of your queries might require you to pass in a variable number of parameters, with the exact number of parameters not known until runtime. For example, you might want to retrieve information about all users from a subset of regions. Room understands when a parameter represents a collection and automatically expands it at runtime based on the number of parameters provided.

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT first\_name, last\_name FROM user WHERE region IN (:regions)")  public List<NameTuple> loadUsersFromRegions(List<String> regions); } |

#### Observable Queries

When performing queries, you'll often want your app's UI to update automatically when the data changes. To achieve this, use a return value of type *LiveData* in your query method description.Room generates all necessary code to update the *LiveData* when the database is updated.

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT first\_name, last\_name FROM user WHERE region IN (:regions)")  public LiveData<List<User>> loadUsersFromRegionsSync(List<String> regions); } |

**Note:** As of version 1.0, Room uses the list of tables accessed in the query to decide whether to update *LiveData* objects.

##### RxJava

Room can also return RxJava2 Publisher and Flowable objects from the queries you define. To use this functionality, add the **android.arch.persistence.room:rxjava2** artifact from the Room group into your build Gradle dependencies, as shown in the following code snippet:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT \* from user where id = :id LIMIT 1")  public Flowable<User> loadUserById(int id); } |

#### Direct Cursor Access

If your app's logic requires direct access to the return rows, you can return a Cursor object from your queries, as shown in the following code snippet: if you need.

|  |
| --- |
| @Dao public interface MyDao {  @Query("select \* from user where age > :minAge LIMIT 5")  public Cursor loadRawUsersOlderThan(int minAge); } |

**Caution:** It's highly discouraged to work with the Cursor API because it doesn't guarantee whether the rows exist or what values the rows contain. Use this functionality only if you already have code that expects a cursor and that you can't refactor easily.

#### Multiple Table Queries

Some of your queries might require access to multiple tables to calculate the result. Room allows you to write any query, so you can also join tables. Furthermore, if the response is an observable data type, such as Flowable or LiveData, Room watches all tables referenced in the query for invalidation.

The following code snippet shows how to perform a table join to consolidate information between a table containing users who are borrowing books and a table containing data about books currently on loan:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT \* FROM book " +  "INNER JOIN loan ON loan.book\_id = book.id " +  "INNER JOIN user ON user.id = loan.user\_id " +  "WHERE user.name LIKE :userName")  public List<Book> findBooksBorrowedByNameSync(String userName); } |

You can also return POJOs from these queries. For example, you can write a query that loads the User and their pet’s name as follows:

|  |
| --- |
| @Dao public interface MyDao {  @Query("SELECT user.name AS userName, pet.name AS petName " +  "FROM user, pet " +  "WHERE user.id = pet.user\_id")  public LiveData<List<UserPet>> loadUserAndPetNames();  // You can also define this class in a separate file, as long as you add the  // "public" access modifier.  static class UserPet {  public String userName;  public String petName;  } } |

## Using TypeConverters for Custom Types

Room provides built-in support for primitives and their boxed alternatives. However, you sometimes use custom data types that you would like to each store in the database in a single column. To add this kind of support for custom types, you provide a *TypeConverter*, which converts a custom class to/from a known type that Room can persist.

For example, if we want to persist instances of *java.util.Date*, we can write the following *TypeConverter* to store the equivalent Unix timestamp in the database:

|  |
| --- |
| public class Converters {  @TypeConverter  public static Date fromTimestamp(Long value) {  return value == null ? null : new Date(value);  }   @TypeConverter  public static Long dateToTimestamp(Date date) {  return date == null ? null : date.getTime();  } } |

The preceding example defines 2 functions, one that converts a *java.util.Date* object to a *Long* object and another that performs the inverse conversion, from *Long* to *java.util.Date*. Since Room already knows how to persist *Long* objects, it can use this converter to persist values of type *java.util.Date*.

Next, you adding the *@TypeConverters* annotation to *AppDatabase* so that Room can use the *Converter* that you've defined for each Entity and Dao in the *AppDatabase*.

|  |
| --- |
| // File: AppDatabase.java @Database(entities = {User.java}, version = 1) @TypeConverters({Converter.class}) public abstract class AppDatabase extends RoomDatabase {  public abstract UserDao userDao(); } |

Using these converters, you can then use your custom types in other queries, just as you would use primitive types, as shown in the following code snippet:

|  |
| --- |
| // File: User.java  @Entity public class User {  ...  private Date birthday; }  // File: UserDao.java @Dao public interface UserDao {  ...  @Query("SELECT \* FROM user WHERE birthday BETWEEN :from AND :to")  List<User> findUsersBornBetweenDates(Date from, Date to); } |

You can also limit the *@TypeConverters* to different scopes, including individual entities, DAOs, and DAO methods. For more details, see the reference documentation for the *@TypeConverters* annotation.

## Migrations

As you add and change features in your app, you need to modify your entity classes to reflect these changes. When a user updates to the latest version of your app, you don't want them to lose all of their existing data, especially if you can't recover the data from a remote server.

Room allows you to write Migration classes to preserve user data in this manner. Each Migration class specifies *from* and *to* versions. At runtime, Room runs each Migration class's migrate() method, using the correct order to migrate the database to a later version.

**Caution:** If you don't provide the necessary migrations, Room rebuilds the database instead (which means you’ll lose all of your data in the database).

|  |
| --- |
| Room.databaseBuilder(getApplicationContext(), MyDb.class, "database-name")  .addMigrations(MIGRATION\_1\_2, MIGRATION\_2\_3).build();  static final Migration MIGRATION\_1\_2 = new Migration(1, 2) {  @Override  public void migrate(SupportSQLiteDatabase database) {  database.execSQL("CREATE TABLE `Fruit` (`id` INTEGER,"  + " `name` TEXT, PRIMARY KEY(`id`))");  } };  static final Migration MIGRATION\_2\_3 = new Migration(2, 3) {  @Override  public void migrate(SupportSQLiteDatabase database) {  database.execSQL("ALTER TABLE Book "  + " ADD COLUMN pub\_year INTEGER");  } }; |

**Caution:** To keep your migration logic functioning as expected, use full queries instead of referencing constants that represent the queries.

After the migration process finishes, Room validates the schema to ensure that the migration occurred correctly. If Room finds a problem, it throws an exception that contains the mismatched information.

### Testing Migrations

Migrations aren't trivial to write, and failure to write them properly could cause a crash loop in your application. To preserve your app's stability, you should test your migrations beforehand. Room provides a **testing** maven artifact to assist with this testing process. However, for this artifact to work, you need to export your database's schema.

#### Exporting Schema

Upon compilation, Room exports your database's schema information into a JSON file. You should store your database's schema history in your version control system, as it allows Room to create older versions of your database for testing purposes.

To export the schema, set the *room.schemaLocation* annotation processor property in your build.gradle file, as shown in the following code snippet:

|  |
| --- |
| // build.gradle android {  ...  defaultConfig {  ...  javaCompileOptions {  annotationProcessorOptions {  arguments = ["room.schemaLocation": "$projectDir/schemas".toString()]  }  }  } } |

You should commit the exported JSON files into your version control system.

To test these migrations, add the **android.arch.persistence.room:testing** maven artifact from **room** into your test dependencies, and add the schema location as an asset folder, as shown in the following code snippet:

|  |
| --- |
| // build.gradle android {  ...  sourceSets {  androidTest.assets.srcDirs += files("$projectDir/schemas".toString())  } } |

The testing package provides a *MigrationTestHelper* class, which can read these schema files. It is also a Junit4 *TestRule* so it can manage created databases.

A sample migration test appears in the following code snippet:

|  |
| --- |
| @RunWith(AndroidJUnit4.class) public class MigrationTest {  private static final String TEST\_DB = "migration-test";  @Rule  public MigrationTestHelper helper;   public MigrationTest() {  helper = new MigrationTestHelper(InstrumentationRegistry.getContext(),  MigrationDb.class.getCanonicalName(),  new FrameworkSQLiteOpenHelperFactory());  }   @Test  public void migrate1To2() throws IOException {  SupportSQLiteDatabase db = helper.createDatabase(TEST\_DB, 1);  // db has schema version 1. insert some data using SQL queries.  // You cannot use DAO classes because they expect the latest schema.  db.execSQL(...);  // Prepare for the next version.  db.close();  // Re-open the database with version 2 and provide  // MIGRATION\_1\_2 as the migration process.  db = helper.runMigrationsAndValidate(TEST\_DB, 2, true, MIGRATION\_1\_2);   // MigrationTestHelper automatically verifies the schema changes,  // but you need to validate that the data was migrated properly.  }  } |

## Testing

When running tests for your app, you shouldn't need to create a full database if you are not testing the database itself. Room allows you to easily mock the data access layer in your tests. This process is possible because your DAOs don't leak any details of your database.

When testing the rest of your application, you should create mock or fake instances of your DAO classes.

There are 2 ways to test your database, on your host development machine and on an Android device..

### JUnit Test on Your Host Machine

Room uses SQLite Support Library, which provides interfaces that match those in the Android Framework classes. This support allows you to pass custom implementations of the support library to test your database queries.

Even though this setup allows your tests to run very quickly, it **isn't recommended** because the version of SQLite running on your device—and your users' devices—might not match the version on your host machine.

### JUnit Test on an Android Device

The recommended approach for testing your database implementation is writing a JUnit test that runs on an Android device. These tests don’t require creating an Activity, so they still should be faster than to your UI tests.

When setting up your tests, you should create an in-memory version of your database to make your tests more hermetic as shown in the following example:

|  |
| --- |
| @RunWith(AndroidJUnit4.class) public class SimpleEntityReadWriteTest {  private UserDao mUserDao;  private TestDatabase mDb;   @Before  public void createDb() {  Context context = InstrumentationRegistry.getTargetContext();  mDb = Room.inMemoryDatabaseBuilder(context, TestDatabase.class).build();  mUserDao = mDb.getUserDao();  }   @After  public void closeDb() throws IOException {  SupportSQLiteDatabase connection = mDb.getDatabase();  if (connection != null) {  connection.close();  }  }   @Test  public void writeUserAndReadInList() throws Exception {  User user = TestUtil.createUser(3);  user.setName("george");  mUserDao.insert(user);  List<User> byName = mUserDao.findUsersByName("george");  assertThat(byName.get(0), equalTo(user));  } } |

For more information about testing your database migrations, see [Migration Testing](#_ab93tai3o3xt).

## Best Practices

## Addendum

### What is Wrong With Object References Between Entities

Mapping relationships from a database to the respective object model is a common practice and works very well on the **server side** where it's performant to lazily load fields as they're accessed.

However, on the client side, lazy loading is not feasible because it's likely to happen on the UI thread, and querying information on disk in the UI thread creates significant performance problems. The UI Thread has about 16 ms to calculate and draw an activity's s updated layout, so even if a query takes only 5 ms, it's still likely that your app will run out of time to draw the frame, causing noticeable jank. Worse still, the query could take more time to complete if there's a separate transaction running in parallel, or if the device is busy with other disk-heavy tasks.

If you don't use lazy loading, however, the app fetches more data than it needs, creating memory consumption problems.

ORMs usually leave this decision to developers so that they can do whatever is best for their app's use cases. Unfortunately, developers usually end up sharing the model between their app and UI. As the UI changes over time, problems occur that are difficult to anticipate and debug.

For example, take a UI that loads a list of *Books*, with each book having an *Author*. You might initially design your queries to use lazy loading, such that *Book* objects use a *getAuthor()* method to return the *Author*. The first invocation of the getAuthor() call queries the database. Some time later, you realize that you need to display the author name in your app's UI as well. You can add the method call easily enough, as shown in the following code snippet::

|  |
| --- |
| authorNameTextView.setText(user.getAuthor().getName()); |

However, this seemingly innocent change causes the *Authors* table to be queried on the main thread.

If you query author information **eagerly,** it becomes difficult to change how data is loaded if you no longer need that data, such as the case where your app's UI no longer needs to display information about the *Author*. Therefore, your app continues to load data that it no longer displays. This situation is even worse if the *Author* class references another table, such as with a *getBooks()* method.

For these reasons, Room disallows object references between entity classes. Instead, you must explicitly request the data that your app needs.