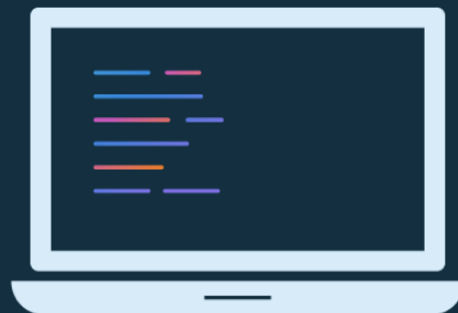




Lesson 9: App architecture (persistence)



About this lesson

Lesson 9: App architecture (persistence)

- [Storing data](#)
- [Room persistence library](#)
- [Asynchronous programming](#)
- [Coroutines](#)
- [Testing databases](#)
- [Summary](#)

Storing data

Ways to store data in an Android app

- App-specific storage
- Shared storage (files to be shared with other apps)
- Preferences
- Databases

What is a database?

Collection of structured data that can be easily accessed, searched, and organized, consisting of:

- Tables
- Rows
- Columns

Example Database

person
_id
name
age
email

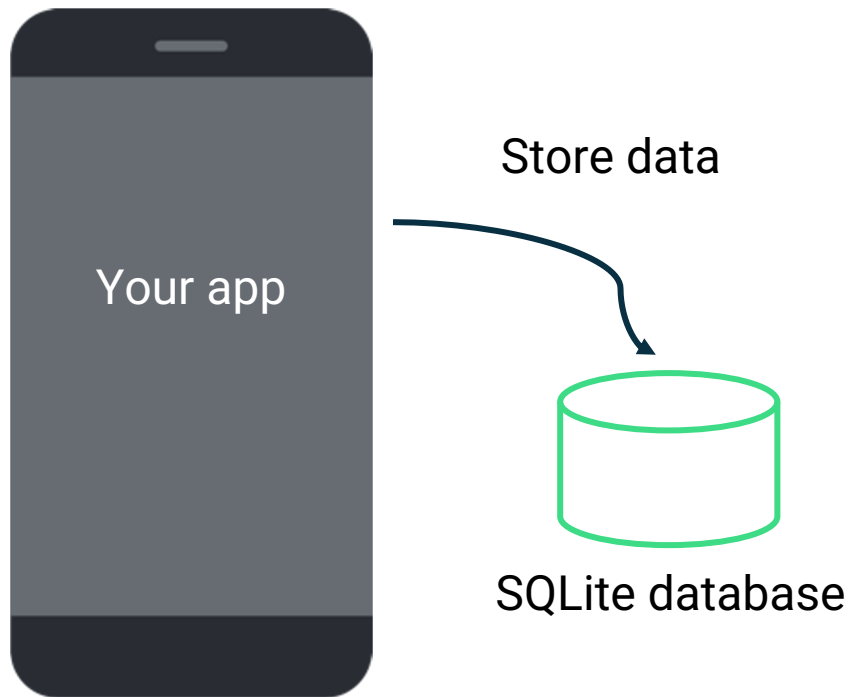
car
_id
make
model
year

Structured Query Language (SQL)

Use SQL to access and modify a relational database.

- Create new tables
- Query for data
- Insert new data
- Update data
- Delete data

SQLite in Android



Example SQLite commands

Create

```
INSERT INTO colors VALUES ("red", "#FF0000");
```

Read

```
SELECT * from colors;
```

Update

```
UPDATE colors SET hex="#DD0000" WHERE name="red";
```

Delete

```
DELETE FROM colors WHERE name = "red";
```


Interacting directly with a database

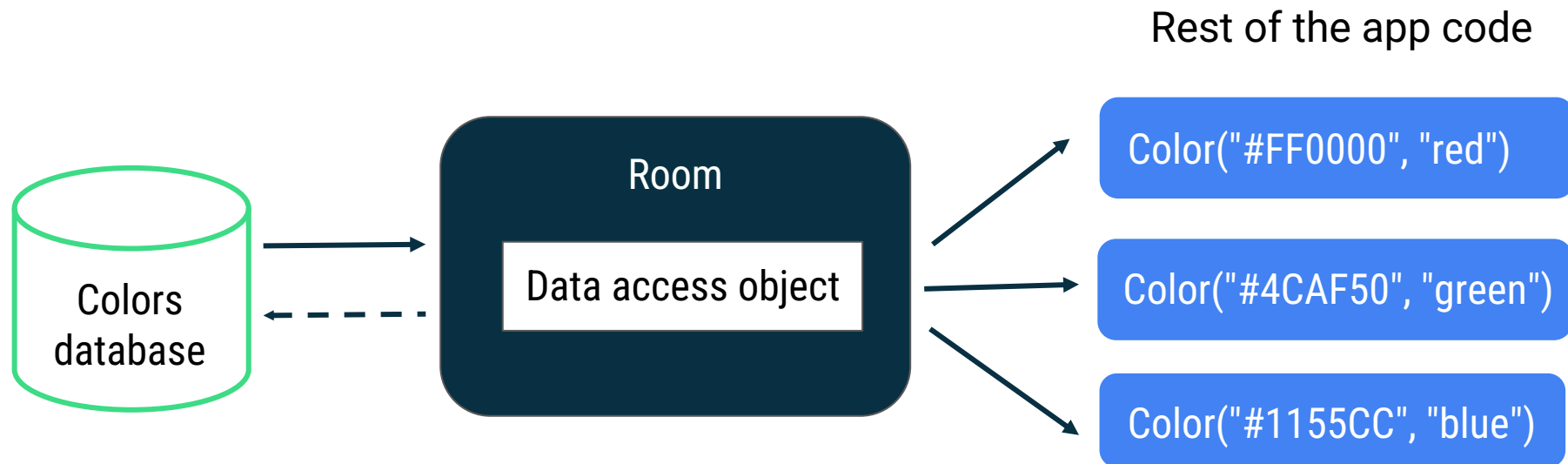
- No compile-time verification of raw SQL queries
- Need lots of boilerplate code to convert between SQL queries ↔ data objects

Room persistence library

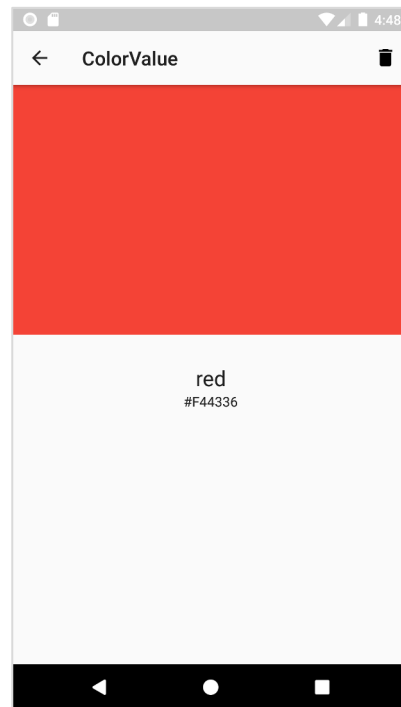
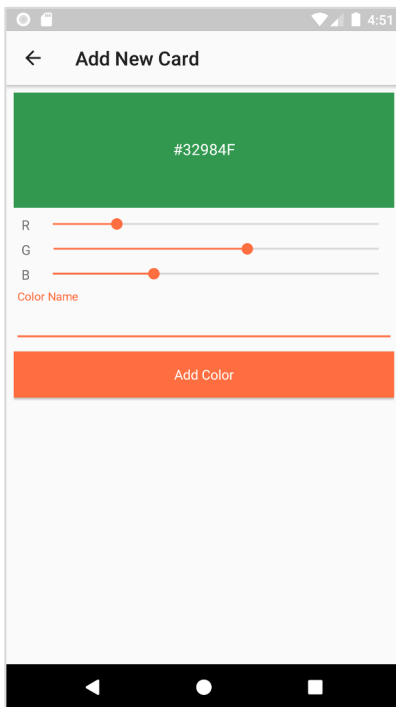
Add Gradle dependencies

```
dependencies {  
    implementation "androidx.room:room-runtime:$room_version"  
    kapt "androidx.room:room-compiler:$room_version"  
  
    // Kotlin Extensions and Coroutines support for Room  
    implementation "androidx.room:room-ktx:$room_version"  
  
    // Test helpers  
    testImplementation "androidx.room:room-testing:$room_version"  
}
```

Room



ColorValue app



Room

- Entity `Color`
- DAO `ColorDao`
- Database `ColorDatabase`

Color class

```
data class Color {  
    val hex: String,  
    val name: String  
}
```

Annotations

- Provide extra information to the compiler

`@Entity` marks entity class, `@Dao` for DAO, `@Database` for database

- Can take parameters

```
@Entity(tableName = "colors")
```

- Can autogenerate code for you

Entity

Class that maps to a SQLite database table

- `@Entity`
- `@PrimaryKey`
- `@ColumnInfo`

Example entity

```
@Entity(tableName = "colors")

data class Color {

    @PrimaryKey(autoGenerate = true) val _id: Int,

    @ColumnInfo(name = "hex_color") val hex: String,

    val name: String

}
```

colors

_id
hex_color
name

Data access object (DAO)

Work with DAO classes instead of accessing database directly:

- Define database interactions in the DAO.
- Declare DAO as an interface or abstract class.
- Room creates DAO implementation at compile time.
- Room verifies all of your DAO queries at compile-time.

Example DAO

@Dao

interface ColorDao {

@Query("SELECT * FROM colors")

fun getAll(): Array<Color>

@Insert

fun insert(vararg color: Color)

@Update

fun update(color: Color)

@Delete

fun delete(color: Color)

Query

@Dao

```
interface ColorDao {
```

```
    @Query("SELECT * FROM colors")
```

```
    fun getAll(): Array<Color>
```

```
    @Query("SELECT * FROM colors WHERE name = :name")
```

```
    fun getColorByName(name: String): LiveData<Color>
```

```
    @Query("SELECT * FROM colors WHERE hex_color = :hex")
```

```
    fun getColorByHex(hex: String): LiveData<Color>
```

Insert

```
@Dao
interface ColorDao {
    ...

    @Insert
    fun insert(vararg color: Color)

    ...
}
```

Update

```
@Dao
interface ColorDao {
    ...

    @Update
    fun update(color: Color)

    ...
}
```

Delete

```
@Dao
interface ColorDao {
    ...

    @Delete
    fun delete(color: Color)

    ...
}
```


Create a Room database

- Annotate class with `@Database` and include list of entities:

```
@Database(entities = [Color::class], version = 1)
```

- Declare abstract class that extends `RoomDatabase`:

```
abstract class ColorDatabase : RoomDatabase() {
```

- Declare abstract method with no args that returns the DAO:

```
abstract fun colorDao(): ColorDao
```

Example Room database

```
@Database(entities = [Color::class], version = 1)
abstract class ColorDatabase : RoomDatabase() {
    abstract fun colorDao(): ColorDao
    companion object {
        @Volatile
        private var INSTANCE: ColorDatabase? = null
        fun getInstance(context: Context): ColorDatabase {
            ...
        }
    }
    ...
}
```

Create database instance

```
fun getInstance(context: Context): ColorDatabase {  
    return INSTANCE ?: synchronized(this) {  
        INSTANCE ?: Room.databaseBuilder(  
            context.applicationContext,  
            ColorDatabase::class.java, "color_database"  
        )  
        .fallbackToDestructiveMigration()  
        .build()  
        .also { INSTANCE = it }  
    }  
}
```

Get and use a DAO

Get the DAO from the database:

```
val colorDao = ColorDatabase.getInstance(application).colorDao()
```

Create new Color and use DAO to insert it into database:

```
val newColor = Color(hex = "#6200EE", name = "purple")  
colorDao.insert(newColor)
```

Asynchronous programming



Long-running tasks

- Download information
- Sync with a server
- Write to a file
- Heavy computation
- Read from, or write to, a database

Need for async programming

- Limited time to do tasks and remain responsive
- Balanced with the need to execute long-running tasks
- Control over how and where tasks are executed

Async programming on Android

- Threading
- Callbacks
- Plus many other options

What is the recommended way?

Coroutines

Coroutines


- Keep your app responsive while managing long-running tasks.
- Simplify asynchronous code in your Android app.
- Write code in sequential way
- Handle exceptions with `try/catch` block

Benefits of coroutines

- Lightweight
- Fewer memory leaks
- Built-in cancellation support
- Jetpack integration

Suspend functions

- Add `suspend` modifier
- Must be called by other `suspend` functions or coroutines



The diagram shows a code snippet with annotations. A green arrow points from the left margin to the `suspend` keyword. A vertical line connects the `suspend` keyword to the `wordDao.insert(word)` call. A small box with a minus sign is at the top of this line, and another small box with a minus sign is at the bottom, indicating a scope or execution flow.

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

Suspend and resume

- `suspend`

Pauses execution of current coroutine and saves local variables

- `resume`

Automatically loads saved state and continues execution from the point the code was suspended

Example

```
suspend fun fetchDocs() {  
    val docs = get("...")  
    show(docs)  
}
```

Main Thread
[stack]

suspend

resume



Add suspend modifier to DAO methods

```
@Dao
interface ColorDao {

    @Query("SELECT * FROM colors")
    suspend fun getAll(): Array<Color>

    @Insert
    suspend fun insert(vararg color: Color)

    @Update
    suspend fun update(color: Color)

    @Delete
    suspend fun delete(color: Color)
```

Control where coroutines run

Dispatcher	Description of work	Examples of work
<code>Dispatchers.Main</code>	UI and nonblocking (short) tasks	Updating LiveData, calling suspend functions
<code>Dispatchers.IO</code>	Network and disk tasks	Database, file IO
<code>Dispatchers.Default</code>	CPU intensive	Parsing JSON

withContext

```
suspend fun get(url: String) {  
  
    // Start on Dispatchers.Main  
  
    withContext(Dispatchers.IO) {  
        // Switches to Dispatchers.IO  
        // Perform blocking network IO here  
    }  
  
    // Returns to Dispatchers.Main  
}
```

CoroutineScope

Coroutines must run in a `CoroutineScope`:

- Keeps track of all coroutines started in it (even suspended ones)
- Provides a way to cancel coroutines in a scope
- Provides a bridge between regular functions and coroutines

Examples: `GlobalScope`

`ViewModel` **has** `viewModelScope`

`Lifecycle` **has** `lifecycleScope`

Start new coroutines

- `launch` - no result needed

```
fun loadUI() {  
    launch {  
        fetchDocs()  
    }  
}
```

- `async` - can return a result

ViewModelScope

```
class MyViewModel: ViewModel() {  
  
    init {  
        viewModelScope.launch {  
            // Coroutine that will be canceled  
            // when the ViewModel is cleared  
        }  
    }  
    ...  
}
```

Example viewModelScope

```
class ColorViewModel(val dao: ColorDao, application: Application)
    : AndroidViewModel(application) {

    fun save(color: Color) {
        viewModelScope.launch {
            dao.insert(color)
        }
    }

    ...
}
```

Testing databases

Add Gradle dependencies

```
android {  
    defaultConfig {  
        ...  
        testInstrumentationRunner "androidx.test.runner  
            .AndroidJUnitRunner"  
        testInstrumentationRunnerArguments clearPackageData: 'true'  
    }  
}  
  
dependencies {  
    testImplementation 'junit:junit:4.12'  
    androidTestImplementation 'androidx.test.ext:junit:1.1.0'  
    androidTestImplementation 'androidx.test.espresso:espresso-core:3.1.1'  
}
```

Testing Android code

- `@RunWith (AndroidJUnit4::class)`
- `@Before`
- `@After`
- `@Test`

Create test class

```
@RunWith(AndroidJUnit4::class)
```

```
class DatabaseTest {
```

```
    private lateinit val colorDao: ColorDao
```

```
    private lateinit val db: ColorDatabase
```

```
    private val red = Color(hex = "#FF0000", name = "red")
```

```
    private val green = Color(hex = "#00FF00", name = "green")
```

```
    private val blue = Color(hex = "#0000FF", name = "blue")
```

```
    ...
```



Create and close database for each test

In `DatabaseTest.kt`:

`@Before`

```
fun createDb() {  
    val context: Context = ApplicationProvider.getApplicationContext()  
    db = Room.inMemoryDatabaseBuilder(context, ColorDatabase::class.java)  
        .allowMainThreadQueries()  
        .build()  
    colorDao = db.colorDao()  
}
```

`@After`

```
@Throws(IOException::class)  
fun closeDb() = db.close()
```

Test insert and retrieve from a database

In DatabaseTest.kt:

```
@Test
@Throws(Exception::class)
fun insertAndRetrieve() {
    colorDao.insert(red, green, blue)
    val colors = colorDao.getAll()
    assert(colors.size == 3)
}
```

Summary

Summary

In Lesson 9, you learned how to:

- Set up and configure a database using the Room library
- Use coroutines for asynchronous programming
- Use coroutines with Room
- Test a database

Learn more

- [7 Pro-tips for Room](#)
- [Room Persistence Library](#)
- [SQLite Home Page](#)
- [Save data using SQLite](#)
- [Coroutines Guide](#)
- [Dispatchers - kotlinx-coroutines-core](#)
- [Coroutines on Android \(part I\): Getting the background](#)
- [Coroutines on Android \(part II\): Getting started](#)
- [Easy Coroutines in Android: viewModelScope](#)
- [Kotlin Coroutines 101](#)

Pathway

Practice what you've learned by completing the pathway:

[Lesson 9: App architecture \(persistence\)](#)

