Persist data with Room

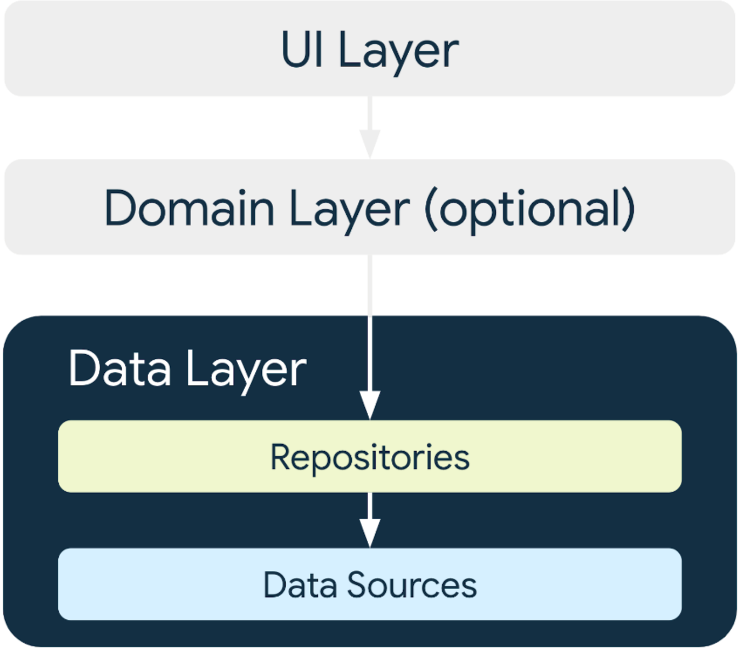
1. Before you begin

Most production-quality apps have data that the app needs to persist. For example, the app might store a playlist of songs, items on a to-do list, records of expenses and income, a catalog of constellations, or a history of personal data. For such use cases, you use a database to store this persistent data.

Room (https://developer.android.com/topic/libraries/architecture/room) is a persistence library that's part of Android Jetpack (https://developer.android.com/jetpack/androidx/explorer?case=data). Room is an abstraction layer on top of a SQLite (https://developer.android.com/training/data-storage/sqlite) database. SQLite uses a specialized language (SQL) to perform database operations. Instead of using SQLite directly, Room simplifies the chores of database setup, configuration, and interactions with the app. Room also provides compile-time checks of SQLite statements.

An *abstraction* layer is a set of functions that hide the underlying implementation/complexity. It provides an interface to an existing set of functionality, like SQLite in this case.

The image below shows how Room, as a data source, fits in with the overall architecture recommended in this course. Room is a Data Source.



Prerequisites

- Ability to build a basic user interface (UI) for an Android app using Jetpack Compose.
- Ability to use composables like `Text`, `Icon`, `IconButton`, and `LazyColumn`.
- Ability to use the `NavHost` composable to define routes and screens in your app.
- Ability to navigate between screens using a `NavHostController`.
- Familiarity with the Android architecture component `ViewModel`. Ability to use `ViewModelProvider.Factory` to instantiate the ViewModels.
- Familiarity with concurrency fundamentals.
- Ability to use coroutines for long-running tasks.
- Basic knowledge of SQLite databases and the SQL language.

What you'll learn

- How to create and interact with the SQLite database using the Room library.
- How to create an entity, a data access object (DAO), and database classes.
- How to use a DAO to map Kotlin functions to SQL queries.

What you'll build

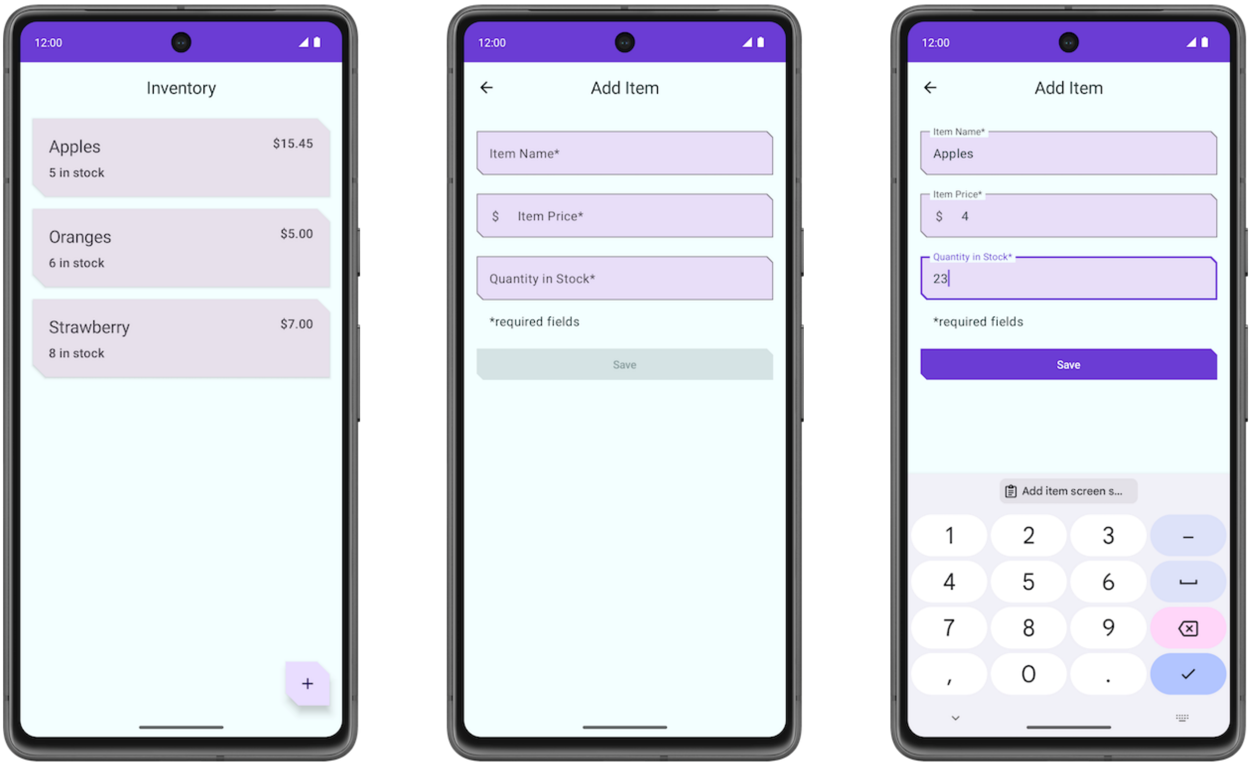
- You'll build an **Inventory** app that saves inventory items into the SQLite database.

What you need

- Starter code for the **Inventory** app
- A computer with Android Studio
- Device or an emulator with API level 26 or higher

2. App overview

In this codelab, you work with a starter code of the Inventory app and add the database layer to it using the Room library. The final version of the app displays a list of items from the inventory database. The user has options to add a new item, update an existing item, and delete an item from the inventory database. For this codelab, you save the item data to the Room database. You complete the rest of the app's functionality in the next codelab.



Inventory

Apples $15.45
5 in stock

Oranges $5.00
6 in stock

Strawberry $7.00
8 in stock

Add Item

Item Name*

$ Item Price*

Quantity in Stock*

*required fields

Add Item

Item Name*: Apples

Item Price*: $ 4

Quantity in Stock*: 23

*required fields

Note: The above screenshots are from the final version of the app at the end of the pathway, not the end of this codelab. These screenshots give you an idea of the final version of the app.

3. Starter app overview

Download the starter code for this codelab

To get started, download the starter code:

https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app/archive/refs/heads/starter.zip

Alternatively, you can clone the GitHub repository for the code:

```
$ git clone https://github.com/google-developer-
training/basic-android-kotlin-compose-training-inventory-
app.git
$ cd basic-android-kotlin-compose-training-inventory-app
$ git checkout starter
```

Note: The starter code is in the `starter` branch of the downloaded repository.

You can browse the code in the `Inventory app` GitHub repository (https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app/tree/starter).

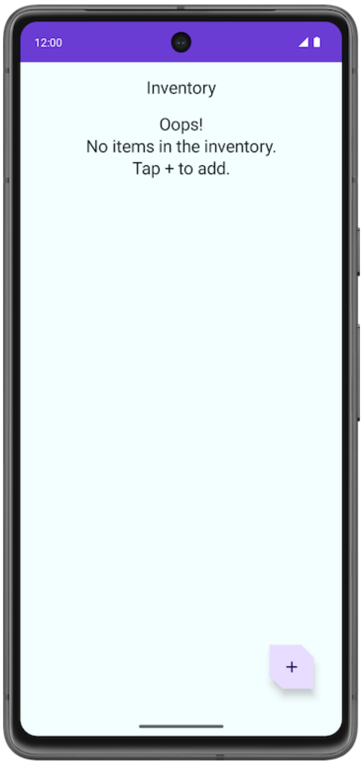
Starter code overview

1. Open the project with the starter code in Android Studio.
2. Run the app on an Android device or an emulator. Make sure the emulator or connected device runs with an API level 26 or higher. Database Inspector (https://developer.android.com/studio/inspect/database) works on emulators/devices that run API level 26 and higher.

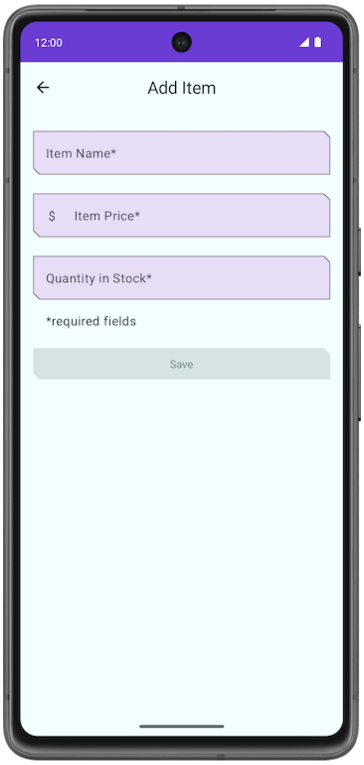
Note: The Database Inspector lets you inspect, query, and modify your app's databases while your app runs. The Database Inspector works with plain SQLite or with libraries built on top of SQLite, such as Room.

3. Notice that the app shows no inventory data.
4. Tap the floating action button (FAB), which lets you add new items to the database.

The app navigates to a new screen where you can enter details for the new item.



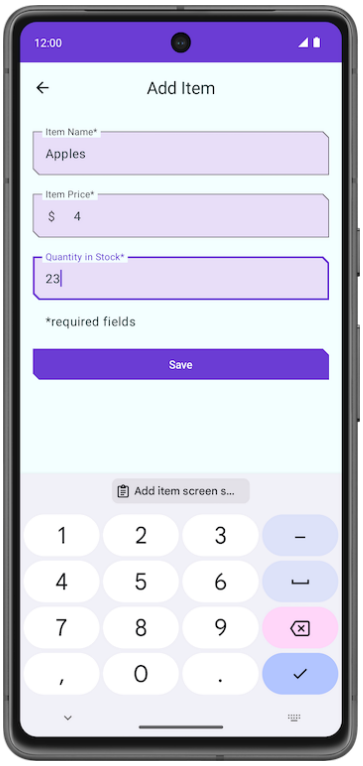
Oops!
No items in the inventory.
Tap + to add.



Problems with the starter code

1. In the **Add Item** screen, enter an item's details like name, price, and quantity of the Item.
2. Tap **Save**. The **Add Item** screen is not closed, but you can navigate back using the back key. The save functionality is not implemented, so the item details are not saved.

Notice that the app is incomplete and the **Save** button functionality is not implemented.



12:00

Add Item

Item Name*
Apples

Item Price*
$ 4

Quantity in Stock*
23

*required fields

Save

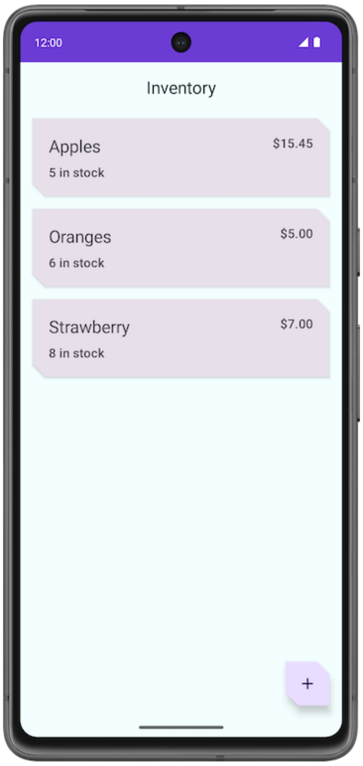
In this codelab, you add the code that uses Room to save the inventory details in the SQLite database. You use the Room persistence library to interact with the SQLite database.

Code walkthrough

The starter code you downloaded has pre-designed screen layouts for you. In this pathway, you focus on implementing the database logic. The following section is a brief walkthrough of some of the files to get you started.

ui/home/HomeScreen.kt

This file is the home screen, or the first screen in the app, which contains the composables to display the inventory list. It has a FAB + to add new items to the list. You display the items in the list later in the pathway.



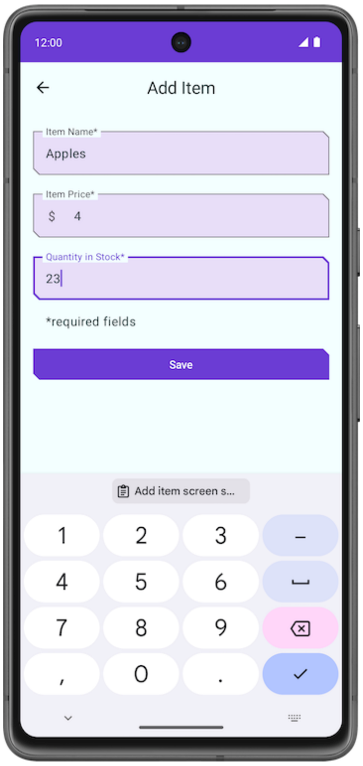
Apples $15.45
5 in stock

Oranges $5.00
6 in stock

Strawberry $7.00
8 in stock

ui/item/ItemEntryScreen.kt

This screen is similar to `ItemEditScreen.kt`. They both have text fields for the item details. This screen is displayed when the FAB is tapped in the home screen. The `ItemEntryViewModel.kt` is the corresponding `ViewModel` for this screen.



Item Name*: Apples

Item Price*: $ 4

Quantity in Stock*: 23

*required fields

ui/navigation/InventoryNavGraph.kt

This file is the navigation graph for the entire application.

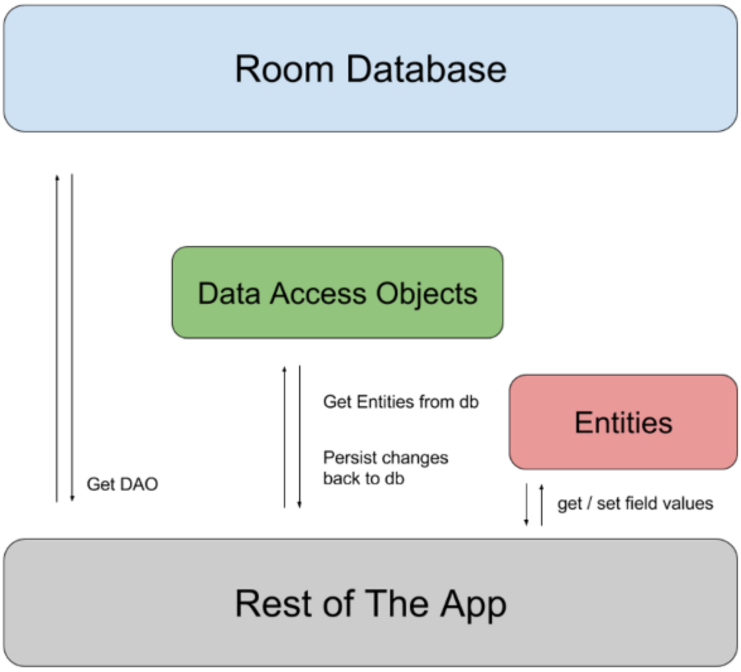
4. Main components of Room

Kotlin provides an easy way to work with data through data classes. While it is easy to work with in-memory data using data classes, when it comes to persisting data, you need to convert this data into a format compatible with database storage. To do so, you need tables to store the data and queries to access and modify the data.

The following three components of Room (https://developer.android.com/topic/libraries/architecture/room) make these workflows seamless.

- Room entities (https://developer.android.com/training/data-storage/room/defining-data) represent tables in your app's database. You use them to update the data stored in rows in tables and to create new rows for insertion.
- Room DAOs (https://developer.android.com/training/data-storage/room/accessing-data) provide methods that your app uses to retrieve, update, insert, and delete data in the database.
- Room Database class (https://developer.android.com/reference/kotlin/androidx/room/Database) is the database class that provides your app with instances of the DAOs associated with that database.

You implement and learn more about these components later in the codelab. The following diagram demonstrates how the components of Room work together to interact with the database.



Add Room dependencies

In this task, you add the required Room component libraries to your Gradle files.

1. Open the module-level gradle file `build.gradle.kts (Module: InventoryApp.app)`.
2. In the `dependencies` block, add the dependencies for the Room library shown in the following code.

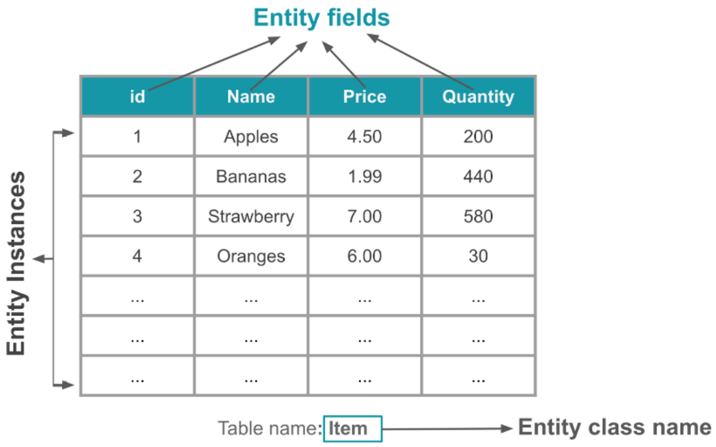
```
//Room
implementation("androidx.room:room-
runtime:${rootProject.extra["room_version"]}")
ksp("androidx.room:room-
compiler:${rootProject.extra["room_version"]}")
implementation("androidx.room:room-
ktx:${rootProject.extra["room_version"]}")
```

KSP is a powerful and yet simple API for parsing Kotlin annotations.

Note: For the library dependencies in your Gradle file, always use the most current stable release version numbers from the AndroidX releases (https://developer.android.com/jetpack/androidx/versions) page.

5. Create an item Entity

An Entity (https://developer.android.com/reference/androidx/room/Entity) class defines a table, and each instance of this class represents a row in the database table. The entity class has mappings to tell Room how it intends to present and interact with the information in the database. In your app, the entity holds information about inventory items, such as item name, item price, and quantity of items available.



id	Name	Price	Quantity
1	Apples	4.50	200
2	Bananas	1.99	440
3	Strawberry	7.00	580
4	Oranges	6.00	30
...	...	...	...
...	...	...	...
...	...	...	...

The `@Entity` annotation marks a class as a database Entity class. For each Entity class, the app creates a database table to hold the items. Each field of the Entity is represented as a column in the database, unless denoted otherwise (see Entity docs for details). Every entity instance stored in the database must have a primary key.
The primary key (https://developer.android.com/reference/androidx/room/PrimaryKey) is used to uniquely identify every record/entry in your database tables. After the app assigns a primary key, it cannot be modified; it represents the entity object as long as it exists in the database.

In this task, you create an Entity class and define fields to store the following inventory information for each item: an `Int` to store the primary key, a `String` to store the item name, a `double` to store the item price, and an `Int` to store the quantity in stock.

1. Open the starter code in the Android Studio.
2. Open the `data` package under the `com.example.inventory` base package.
3. Inside the `data` package, open the `Item` Kotlin class, which represents a database entity in your app.

```
// No need to copy over, this is part of the starter code
class Item(
    val id: Int,
    val name: String,
    val price: Double,
    val quantity: Int
)
```

Note: As a reminder, the primary constructor is part of the class header in a Kotlin class. It goes after the class name (and optional type parameters).

Data classes

Data classes are primarily used to hold data in Kotlin. They are defined with the keyword `data`. Kotlin data class objects have some extra benefits. For example, the compiler automatically generates utilities to compare, print, and copy such as `toString()`, `copy()`, and `equals()`.

Example:

```
// Example data class with 2 properties.
data class User(val firstName: String, val lastName: String){
}
```

To ensure consistency and meaningful behavior of the generated code, data classes must fulfill the following requirements:

- The primary constructor must have at least one parameter.
- All primary constructor parameters must be `val` or `var`.
- Data classes cannot be `abstract`, `open`, or `sealed`.

Warning: The compiler only uses the properties defined inside the primary constructor for the automatically generated functions. The compiler excludes properties declared inside the class body from the generated implementations.

To learn more about Data classes, check out the Data classes (https://kotlinlang.org/docs/data-classes.html) documentation.

1. Prefix the definition of the `Item` class with the `data` keyword to convert it to a data class.

```
data class Item(
    val id: Int,
    val name: String,
    val price: Double,
    val quantity: Int
)
```

2. Above the `Item` class declaration, annotate the data class with `@Entity`. Use the `tableName` argument to set the `items` as the SQLite table name.

```
import androidx.room.Entity

@Entity(tableName = "items")
data class Item(
   ...
)
```

Note: The `@Entity` annotation has several possible arguments. By default (no arguments to `@Entity`), the table name is the same as the class name. Use the `tableName` argument to customize the table name. For simplicity, you use an `item`. There are several other arguments for `@Entity` you can investigate in the Entity documentation (https://developer.android.com/reference/androidx/room/Entity).

3. Annotate the `id` property with `@PrimaryKey` to make the `id` the primary key. A primary key is an ID to uniquely identify every record/entry in your `Item` table

```
import androidx.room.PrimaryKey

@Entity(tableName = "items")
data class Item(
    @PrimaryKey
    val id: Int,
    ...
)
```

4. Assign the `id` a default value of `0`, which is necessary for the `id` to auto generate `id` values.
5. Add the `autoGenerate` parameter to the `@PrimaryKey` annotation to specify whether the primary key column should be auto-generated. If `autogenerate` is set to `true`, Room will automatically generate a unique value for the primary key column when a new entity instance is inserted into the database. This ensures that each entity instance has a unique identifier, without having to manually assign values to the primary key column

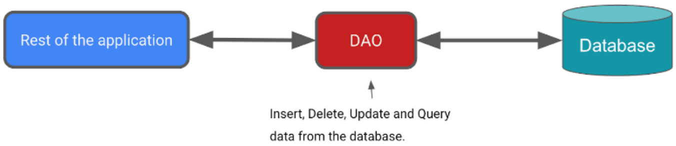
```
data class Item(
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0,
    // ...
)
```

Great! Now that you have created an Entity class, you can create a Data Access Object (DAO) to access the database.

6. Create the item DAO

The Data Access Object (https://developer.android.com/reference/androidx/room/Dao) (DAO) is a pattern you can use to separate the persistence layer from the rest of the application by providing an abstract interface. This isolation follows the single-responsibility principle (https://en.wikipedia.org/wiki/Single-responsibility_principle), which you have seen in previous codelabs.

The functionality of the DAO is to hide all the complexities involved in performing database operations in the underlying persistence layer, separate from the rest of the application. This lets you change the data layer independently of the code that uses the data.



In this task, you define a DAO for Room. DAOs are the main components of Room that are responsible for defining the interface that accesses the database.

The DAO you create is a custom interface that provides convenience methods for querying/retrieving, inserting, deleting, and updating the database. Room generates an implementation of this class at compile time.

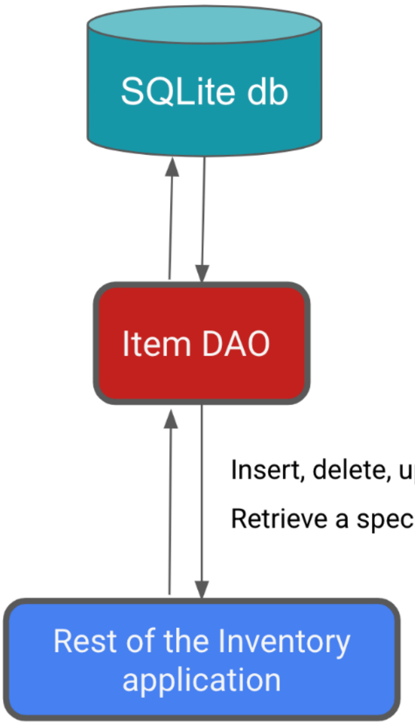
The `Room` library provides convenience annotations, such as `@Insert`, `@Delete`, and `@Update`, for defining methods that perform simple inserts, deletes, and updates without requiring you to write a SQL statement.

If you need to define more complex operations for insert, delete, update, or if you need to query the data in the database, use a `@Query` annotation instead.

As an added bonus, as you write your queries in Android Studio, the compiler checks your SQL queries for syntax errors.

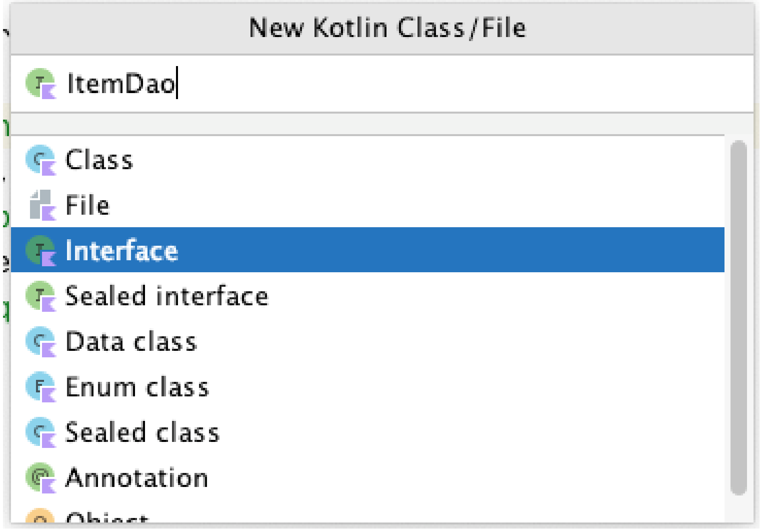
For the Inventory app, you need the ability to do the following:

- **Insert** or add a new item.
- **Update** an existing item to update the name, price, and quantity.
- **Get** a specific item based on its primary key, `id`.
- **Get all items** so you can display them.
- **Delete** an entry in the database.


SQLite db
Item DAO
Insert, delete, update an item.
Retrieve a specific item or all items
Rest of the Inventory application

Complete the following steps to implement the item DAO in your app:

1. In the `data` package, create the Kotlin interface `ItemDao.kt`.



2. Annotate the `ItemDao` interface with `@Dao`.

```
import androidx.room.Dao

@Dao
interface ItemDao {
}
```

3. Inside the body of the interface, add an `@Insert` annotation.

4. Below the `@Insert`, add an `insert()` function that takes an instance of the `Entity` class `item` as its argument.
5. Mark the function with the `suspend` keyword to let it run on a separate thread.

The database operations can take a long time to execute, so they need to run on a separate thread. Room doesn't allow database access on the main thread.

```
import androidx.room.Insert

@Insert
suspend fun insert(item: Item)
```

When inserting items into the database, conflicts can happen. For example, multiple places in the code tries to update the entity with different, conflicting, values such as the same primary key. An entity is a row in DB. In the Inventory app, we only insert the entity from one place that is the **Add Item** screen so we are not expecting any conflicts and can set the conflict strategy to *Ignore*.

6. Add an argument `onConflict` and assign it a value of `OnConflictStrategy.IGNORE`.

The argument `onConflict` tells the Room what to do in case of a conflict. The `OnConflictStrategy.IGNORE` strategy ignores a new item.

To know more about the available conflict strategies, check out the `OnConflictStrategy` (https://developer.android.com/reference/androidx/room/OnConflictStrategy.html) documentation.

```
import androidx.room.OnConflictStrategy

@Insert(onConflict = OnConflictStrategy.IGNORE)
suspend fun insert(item: Item)
```

Now `Room` generates all the necessary code to insert the `item` into the database. When you call any of the DAO functions that are marked with Room annotations, Room executes the corresponding SQL query on the database. For example, when you call the above method, `insert()` from your Kotlin code, `Room` executes a SQL query to insert the entity into the database.

7. Add a new function with `@Update` annotation that takes an `Item` as parameter.

The entity that's updated has the same primary key as the entity that's passed in. You can update some or all of the entity's other properties.

8. Similar to the `insert()` method, mark this function with the `suspend` keyword.

```
import androidx.room.Update

@Update
suspend fun update(item: Item)
```

Add another function with the `@Delete` annotation to delete item(s), and make it a suspending function.

Note: The `@Delete` annotation deletes an item or a list of items. You need to pass the entities you want to delete. If you don't have the entity, you might have to fetch it before calling the `delete()` function.

```
import androidx.room.Delete

@Delete
suspend fun delete(item: Item)
```

There is no convenience annotation for the remaining functionality, so you have to use the `@Query` annotation and supply SQLite queries.

9. Write a SQLite query to retrieve a particular item from the item table based on the given `id`. The following code provides a sample query that selects all columns from the `items`, where the `id` matches a specific value and `id` is a unique identifier.

Example:

```
// Example, no need to copy over
SELECT * from items WHERE id = 1
```

10. Add a `@Query` annotation.
11. Use the SQLite query from the previous step as a string parameter to the `@Query` annotation.
12. Add a `String` parameter to the `@Query` that is a SQLite query to retrieve an item from the item table.

The query now says to select all columns from the `items`, where the `id` matches the `:id` argument. Notice the `:id` uses the colon notation in the query to reference arguments in the function.

```
@Query("SELECT * from items WHERE id = :id")
```

13. After the `@Query` annotation, add a `getItem()` function that takes an `Int` argument and returns a `Flow<Item>`.

```
import androidx.room.Query
import kotlinx.coroutines.flow.Flow

@Query("SELECT * from items WHERE id = :id")
fun getItem(id: Int): Flow<Item>
```

It is recommended to use `Flow` in the persistence layer. With `Flow` as the return type, you receive notification whenever the data in the database changes. The `Room` keeps this `Flow` updated for you, which means you only need to explicitly get the data once. This setup is helpful to update the inventory list, which you implement in the next codelab. Because of the `Flow` return type, Room also runs the query on the background thread. You don't need to explicitly make it a `suspend` function and call it inside a coroutine scope.

Note: Flow in Room database can keep the data *up-to-date* by emitting a notification whenever the data in the database changes. This allows you to observe the data and update your UI accordingly.

14. Add a `@Query` with a `getAllItems()` function.
15. Have the SQLite query return all columns from the `item` table, ordered in ascending order.
16. Have `getAllItems()` return a list of `Item` entities as `Flow`. `Room` keeps this `Flow` updated for you, which means you only need to explicitly get the data once.

```
@Query("SELECT * from items ORDER BY name ASC")
fun getAllItems(): Flow<List<Item>>
```

Completed `ItemDao`:

```
import androidx.room.Dao
import androidx.room.Delete
import androidx.room.Insert
import androidx.room.OnConflictStrategy
import androidx.room.Query
import androidx.room.Update
import kotlinx.coroutines.flow.Flow

@Dao
interface ItemDao {
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun insert(item: Item)

    @Update
    suspend fun update(item: Item)

    @Delete
    suspend fun delete(item: Item)

    @Query("SELECT * from items WHERE id = :id")
    fun getItem(id: Int): Flow<Item>

    @Query("SELECT * from items ORDER BY name ASC")
    fun getAllItems(): Flow<List<Item>>
}
```

17. Though you won't see any visible changes, build your app to make sure it has no errors.

7. Create a Database instance

In this task, you create a RoomDatabase that uses your `Entity` and DAO from the previous tasks. The database class defines the list of entities and DAOs.

The Database class provides your app with instances of the DAOs you define. In turn, the app can use the DAOs to retrieve data from the database as instances of the associated data entity objects. The app can also use the defined data entities to update rows from the corresponding tables or to create new rows for insertion.

You need to create an abstract `RoomDatabase` class and annotate it with `@Database`. This class has one method that returns the existing instance of the `RoomDatabase` if the database doesn't exist.

Here's the general process for getting the `RoomDatabase` instance:

- Create a `public abstract` class that extends `RoomDatabase`. The new abstract class you define acts as a database holder. The class you define is abstract because `Room` creates the implementation for you.
- Annotate the class with `@Database`. In the arguments, list the entities for the database and set the version number.
- Define an abstract method or property that returns an `ItemDao` instance, and the `Room` generates the implementation for you.
- You only need one instance of the `RoomDatabase` for the whole app, so make the `RoomDatabase` a singleton.
- Use `Room`'s `Room.databaseBuilder` to create your (`item_database`) database only if it doesn't exist. Otherwise, return the existing database.

Create the Database

1. In the `data` package, create a Kotlin class `InventoryDatabase.kt`.
2. In the `InventoryDatabase.kt` file, make `InventoryDatabase` class an `abstract` class that extends RoomDatabase.
3. Annotate the class with `@Database`. Disregard the missing parameters error, which you fix in the next step.

```
import androidx.room.Database
import androidx.room.RoomDatabase

@Database
abstract class InventoryDatabase : RoomDatabase() {}
```

The `@Database` annotation requires several arguments so that `Room` can build the database.

4. Specify the `Item` as the only class with the list of `entities`.
5. Set the `version` as `1`. Whenever you change the schema of the database table, you have to increase the version number.
6. Set `exportSchema` to `false` so as not to keep schema version history backups.

```
@Database(entities = [Item::class], version = 1, exportSchema
= false)
```

7. Inside the body of the class, declare an abstract function that returns the `ItemDao` so that the database knows about the DAO.

```
abstract fun itemDao(): ItemDao
```

8. Below the abstract function, define a `companion object`, which allows access to the methods to create or get the database and uses the class name as the qualifier.

```
companion object {}
```

9. Inside the `companion` object, declare a private nullable variable `Instance` for the database and initialize it to `null`.

The `Instance` variable keeps a reference to the database, when one has been created. This helps maintain a single instance of the database opened at a given time, which is an expensive resource to create and maintain.

10. Annotate `Instance` with `@Volatile`.

The value of a volatile variable is never cached, and all reads and writes are to and from the main memory. These features help ensure the value of `Instance` is always up to date and is the same for all execution threads. It means that changes made by one thread to `Instance` are immediately visible to all other threads.

```
@Volatile
private var Instance: InventoryDatabase? = null
```

11. Below `Instance`, while still inside the `companion` object, define a `getDatabase()` method with a `Context` parameter that the database builder needs.
12. Return a type `InventoryDatabase`. An error message appears because `getDatabase()` isn't returning anything yet.

```
import android.content.Context

fun getDatabase(context: Context): InventoryDatabase {}
```

Multiple threads can potentially ask for a database instance at the same time, which results in two databases instead of one. This issue is known as a race condition (https://en.wikipedia.org/wiki/Race_condition). Wrapping the code to get the database inside a `synchronized` block means that only one thread of execution at a time can enter this block of code, which makes sure the database only gets initialized once. Use `synchronized{}` block to avoid the race condition.

13. Inside `getDatabase()`, return the `Instance` variable—or, if `Instance` is null, initialize it inside a `synchronized{}` block. Use the elvis operator(`?:`) to do this.
14. Pass in `this`, the companion object. You fix the error in later steps.

```
return Instance ?: synchronized(this) { }
```

15. Inside the synchronized block, use the database builder to get the database. Continue to ignore the errors, which you fix in the next steps.

```
import androidx.room.Room

Room.databaseBuilder()
```

16. Inside the `synchronized` block, use the database builder to get a database. Pass in the application context, the database class, and a name for the database- `item_database` to the `Room.databaseBuilder()`.

```
Room.databaseBuilder(context, InventoryDatabase::class.java,
"item_database")
```

Android Studio generates a Type Mismatch error. To remove this error, you have to add a `build()` in the following steps.

17. Add the required migration strategy to the builder. Use `.fallbackToDestructiveMigration()`.

```
.fallbackToDestructiveMigration()
```

Note: Normally, you would provide a migration object with a migration strategy for when the schema changes. A *migration object* is an object that defines how you take all rows with the old schema and convert them to rows in the new schema, so that no data is lost. Migration (https://medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929) is beyond the scope of this codelab, but the term refers to when the schema is changed and you need to move your date without losing the data. Since this is a sample app, a simple alternative is to destroy and rebuild the database, which means that the inventory data is lost. For example, if you change something in the entity class, like adding a new parameter, you can allow the app to delete and re-initialize the database.

18. To create the database instance, call `.build()`. This call removes the Android Studio errors.

```
.build()
```

19. After `build()`, add an `also` block and assign `Instance = it` to keep a reference to the recently created db instance.

```
.also { Instance = it }
```

20. At the end of the `synchronized` block, return `instance`. Your final code looks like the following code:

```
import android.content.Context
import androidx.room.Database
import androidx.room.Room
import androidx.room.RoomDatabase

/**
* Database class with a singleton Instance object.
*/
@Database(entities = [Item::class], version = 1, exportSchema
= false)
abstract class InventoryDatabase : RoomDatabase() {

    abstract fun itemDao(): ItemDao

    companion object {
        @Volatile
        private var Instance: InventoryDatabase? = null

        fun getDatabase(context: Context): InventoryDatabase {
            // if the Instance is not null, return it,
otherwise create a new database instance.
            return Instance ?: synchronized(this) {
                Room.databaseBuilder(context,
InventoryDatabase::class.java, "item_database")
                    .build()
                    .also { Instance = it }
            }
        }
    }
}
```

Tip: You can use this code as a template for your future projects. The way you create the `RoomDatabase` instance is similar to the process in the previous steps. You might have to replace the entities and DAOs specific to your app.

21. Build your code to make sure there are no errors.

8. Implement the Repository

In this task, you implement the `ItemsRepository` interface and `OfflineItemsRepository` class to provide `get`, `insert`, `delete`, and `update` entities from the database.

1. Open the `ItemsRepository.kt` file under the `data` package.
2. Add the following functions to the interface, which map to the DAO implementation.

```
import kotlinx.coroutines.flow.Flow

/**
* Repository that provides insert, update, delete, and
retrieve of [Item] from a given data source.
*/
interface ItemsRepository {
    /**
     * Retrieve all the items from the the given data source.
     */
    fun getAllItemsStream(): Flow<List<Item>>

    /**
     * Retrieve an item from the given data source that
matches with the [id].
```

```
     */
    fun getItemStream(id: Int): Flow<Item?>

    /**
     * Insert item in the data source
     */
    suspend fun insertItem(item: Item)

    /**
     * Delete item from the data source
     */
    suspend fun deleteItem(item: Item)

    /**
     * Update item in the data source
     */
    suspend fun updateItem(item: Item)
}
```

3. Open the `OfflineItemsRepository.kt` file under the `data` package.
4. Pass in a constructor parameter of the type `ItemDao`.

```
class OfflineItemsRepository(private val itemDao: ItemDao) :
ItemsRepository
```

5. In the `OfflineItemsRepository` class, override the functions defined in the `ItemsRepository` interface and call the corresponding functions from the `ItemDao`.

```
import kotlinx.coroutines.flow.Flow

class OfflineItemsRepository(private val itemDao: ItemDao) :
ItemsRepository {
    override fun getAllItemsStream(): Flow<List<Item>> =
itemDao.getAllItems()

    override fun getItemStream(id: Int): Flow<Item?> =
itemDao.getItem(id)

    override suspend fun insertItem(item: Item) =
itemDao.insert(item)

    override suspend fun deleteItem(item: Item) =
itemDao.delete(item)

    override suspend fun updateItem(item: Item) =
itemDao.update(item)
}
```

Implement AppContainer class

In this task, you instantiate the database and pass in the DAO instance to the `OfflineItemsRepository` class.

1. Open the `AppContainer.kt` file under the `data` package.
2. Pass in the `ItemDao()` instance to the `OfflineItemsRepository` constructor.
3. Instantiate the database instance by calling `getDatabase()` on the `InventoryDatabase` class passing in the context and call `.itemDao()` to create the instance of `Dao`.

```
override val itemsRepository: ItemsRepository by lazy {
    OfflineItemsRepository(InventoryDatabase.getDatabase(context).itemDao())
}
```

You now have all the building blocks to work with your Room. This code compiles and runs, but you have no way to tell if it actually works. So, this moment is a good time to test your database. To complete the test, you need the `ViewModel` to talk to the database.

9. Add the save functionality

You have thus far created a database, and the UI classes were part of the starter code. To save the app's transient data and to also access the database, you need to update the `ViewModels`. Your `ViewModels` interact with the database via the DAO and provide data to the UI. All database operations need to be run away from the main UI thread; you do so with coroutines and `viewModelScope` (https://developer.android.com/topic/libraries/architecture/coroutines#viewmodelscope).

UI state class walkthrough

Open the `ui/item/ItemEntryViewModel.kt` file. The `ItemUiState` data class represents the UI state of an Item. The `ItemDetails` data class represents a single item.

The starter code provides you with three extension functions:

- The `ItemDetails.toItem()` extension function converts the `ItemUiState` UI state object to the `Item` entity type.
- The `Item.toItemUiState()` extension function converts the `Item` Room entity object to the `ItemUiState` UI state type.

- The `Item.toItemDetails()` extension function converts the `Item` Room entity object to the `ItemDetails`.

```
// No need to copy, this is part of starter code
/**
* Represents Ui State for an Item.
*/
data class ItemUiState(
    val itemDetails: ItemDetails = ItemDetails(),
    val isEntryValid: Boolean = false
)

data class ItemDetails(
    val id: Int = 0,
    val name: String = "",
    val price: String = "",
    val quantity: String = "",
)

/**
* Extension function to convert [ItemDetails] to [Item]. If
the value of [ItemDetails.price] is
* not a valid [Double], then the price will be set to 0.0.
Similarly if the value of
* [ItemDetails.quantity] is not a valid [Int], then the
quantity will be set to 0
*/
fun ItemDetails.toItem(): Item = Item(
    id = id,
    name = name,
    price = price.toDoubleOrNull() ?: 0.0,
    quantity = quantity.toIntOrNull() ?: 0
)

fun Item.formatedPrice(): String {
    return NumberFormat.getCurrencyInstance().format(price)
}

/**
* Extension function to convert [Item] to [ItemUiState]
*/
fun Item.toItemUiState(isEntryValid: Boolean = false):
ItemUiState = ItemUiState(
    itemDetails = this.toItemDetails(),
    isEntryValid = isEntryValid
)

/**
* Extension function to convert [Item] to [ItemDetails]
```

```
*/
fun Item.toItemDetails(): ItemDetails = ItemDetails(
    id = id,
    name = name,
    price = price.toString(),
    quantity = quantity.toString()
)
```

You use the above class in the view models to read and update the UI.

Update ItemEntry ViewModel

In this task, you pass in the repository to the `ItemEntryViewModel.kt` file. You also save the item details entered in the Add Item screen into the database.

1. Notice the `validateInput()` private function in the `ItemEntryViewModel` class.

```
// No need to copy over, this is part of starter code
private fun validateInput(uiState: ItemDetails =
itemUiState.itemDetails): Boolean {
    return with(uiState) {
        name.isNotBlank() && price.isNotBlank() &&
quantity.isNotBlank()
    }
}
```

The above function checks if the `name`, `price`, and `quantity` are empty. You use this function to verify user input before adding or updating the entity in the database.

2. Open the `ItemEntryViewModel` class and add a `private` default constructor parameter of the type `ItemsRepository`.

```
import com.example.inventory.data.ItemsRepository

class ItemEntryViewModel(private val itemsRepository:
ItemsRepository) : ViewModel() {
}
```

3. Update the `initializer` for the item entry view model in the `ui/AppViewModelProvider.kt` and pass in the repository instance as a parameter.

```
object AppViewModelProvider {
    val Factory = viewModelFactory {
        // Other Initializers
        // Initializer for ItemEntryViewModel
        initializer {
            ItemEntryViewModel(inventoryApplication().containe
r.itemsRepository)
        }
        //...
    }
}
```

4. Go to the `ItemEntryViewModel.kt` file and at the end of the `ItemEntryViewModel` class and add a suspend function called `saveItem()` to insert an item into the Room database. This function adds the data to the database in a non-blocking way.

```
suspend fun saveItem() {
}
```

5. Inside the function, check if `itemUiState` is valid and convert it to `Item` type so Room can understand the data.
6. Call `insertItem()` on `itemsRepository` and pass in the data. The UI calls this function to add Item details to the database.

```
suspend fun saveItem() {
    if (validateInput()) {
        itemsRepository.insertItem(itemUiState.itemDetails.toI
tem())
    }
}
```

You have now added all the required functions to add entities to the database. In the next task, you update the UI to use the above functions.

ItemEntryBody() composable walkthrough

1. In the `ui/item/ItemEntryScreen.kt` file, the `ItemEntryBody()` composable is partially implemented for you as part of the stater code. Look at the `ItemEntryBody()` composable in the `ItemEntryScreen()` function call.

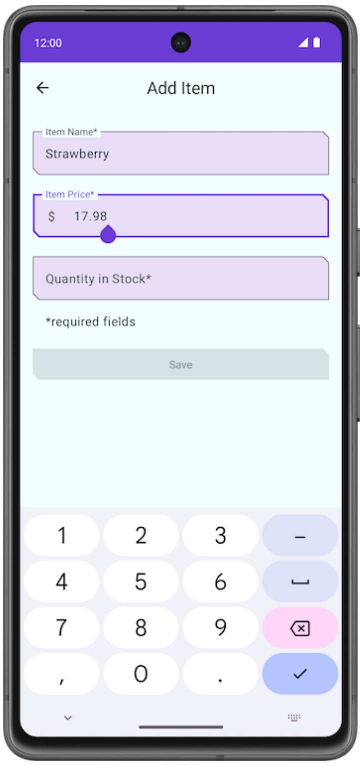
```
// No need to copy over, part of the starter code
ItemEntryBody(
    itemUiState = viewModel.itemUiState,
    onItemValueChange = viewModel::updateUiState,
    onSaveClick = { },
    modifier = Modifier
```

```
        .padding(innerPadding)
        .verticalScroll(rememberScrollState())
        .fillMaxWidth()
)
```

2. Note that the UI state and the `updateUiState` lambda are being passed as function parameters. Look at the function definition to see how the UI state is being updated.

```
// No need to copy over, part of the starter code
@Composable
fun ItemEntryBody(
    itemUiState: ItemUiState,
    onItemValueChange: (ItemUiState) -> Unit,
    onSaveClick: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        // ...
    ) {
        ItemInputForm(
             itemDetails = itemUiState.itemDetails,
             onValueChange = onItemValueChange,
             modifier = Modifier.fillMaxWidth()
         )
        Button(
             onClick = onSaveClick,
             enabled = itemUiState.isEntryValid,
             shape = MaterialTheme.shapes.small,
             modifier = Modifier.fillMaxWidth()
         ) {
             Text(text = stringResource(R.string.save_action))
         }
    }
}
```

You are displaying `ItemInputForm` and a **Save** button in this composable. In the `ItemInputForm()` composable, you are displaying three text fields. The **Save** is only enabled if text is entered in the text fields. The `isEntryValid` value is true if the text in all the text fields is valid (not empty).



Add Item

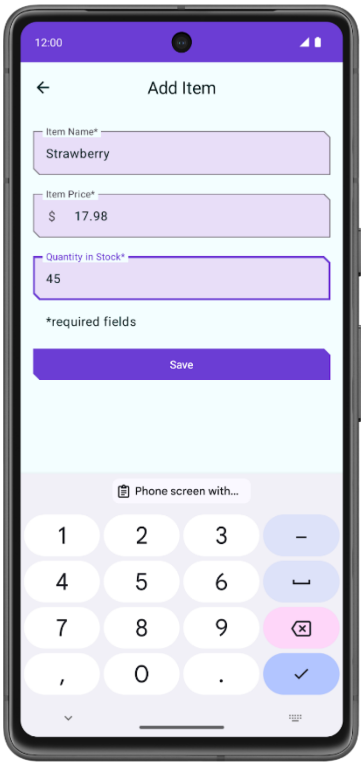
Item Name*
Strawberry

Item Price*
$ 17.98

Quantity in Stock*

*required fields

Save



Add Item

Item Name*
Strawberry

Item Price*
$ 17.98

Quantity in Stock*
45

*required fields

Save

3. Take a look at the `ItemInputForm()` composable function implementation and notice the `onValueChange` function parameter. You are updating the `itemDetails` value with the value entered by the user in the text fields. By the time the **Save** button is enabled, `itemUiState.itemDetails` has the values that need to be saved.

```
// No need to copy over, part of the starter code
@Composable
fun ItemEntryBody(
    //...
) {
    Column(
        // ...
    ) {
        ItemInputForm(
             itemDetails = itemUiState.itemDetails,
             //...
         )
        //...
    }
}
```

```
// No need to copy over, part of the starter code
@Composable
fun ItemInputForm(
    itemDetails: ItemDetails,
    modifier: Modifier = Modifier,
    onValueChange: (ItemUiState) -> Unit = {},
    enabled: Boolean = true
) {
    Column(modifier = modifier.fillMaxWidth(),
verticalArrangement = Arrangement.spacedBy(16.dp)) {
        OutlinedTextField(
            value = itemUiState.name,
            onValueChange = {
onValueChange(itemDetails.copy(name = it)) },
            //...
        )
        OutlinedTextField(
            value = itemUiState.price,
            onValueChange = {
onValueChange(itemDetails.copy(price = it)) },
            //...
        )
        OutlinedTextField(
            value = itemUiState.quantity,
            onValueChange = {
onValueChange(itemDetails.copy(quantity = it)) },
            //...
        )
    }
}
```

Add click listener to the Save button

To tie everything together, add a click handler to the **Save** button. Inside the click handler, you launch a coroutine and call `saveItem()` to save the data in the Room database.

1. In the `ItemEntryScreen.kt`, inside the `ItemEntryScreen` composable function, create a `val` named `coroutineScope` with the `rememberCoroutineScope()` composable function.

Note: The `rememberCoroutineScope()` is a composable function that returns a `CoroutineScope` bound to the composition where it's called. You can use the `rememberCoroutineScope()` composable function when you want to launch a coroutine outside of a composable and ensure the coroutine is canceled after the scope leaves the composition. You can use this function when you need to control the lifecycle of coroutines manually, for example, to cancel an animation whenever a user event happens.

```
import androidx.compose.runtime.rememberCoroutineScope

val coroutineScope = rememberCoroutineScope()
```

2. Update the `ItemEntryBody()` function call and launch a coroutine inside `onSaveClick` lambda.

```
ItemEntryBody(
   // ...
    onSaveClick = {
        coroutineScope.launch {
        }
    },
    modifier = modifier.padding(innerPadding)
)
```

3. Look at the `saveItem()` function implementation in the `ItemEntryViewModel.kt` file to check if `itemUiState` is valid, converting `itemUiState` to `Item` type, and inserting it in the database using `itemsRepository.insertItem()`.

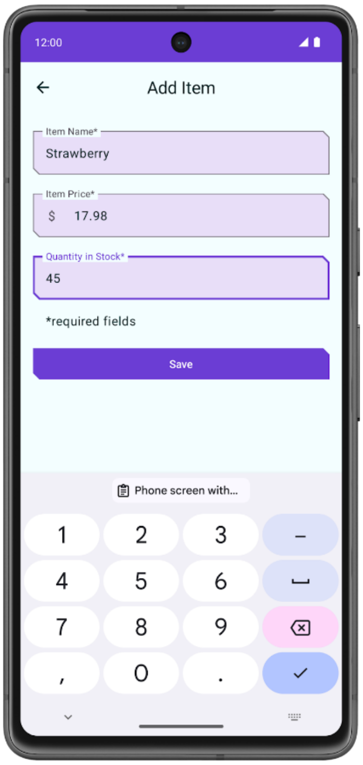
```
// No need to copy over, you have already implemented this as
part of the Room implementation

suspend fun saveItem() {
    if (validateInput()) {
        itemsRepository.insertItem(itemUiState.itemDetails.toI
tem())
    }
}
```

4. In the `ItemEntryScreen.kt`, inside the `ItemEntryScreen` composable function, inside the coroutine, call `viewModel.saveItem()` to save the item in the database.

```
ItemEntryBody(
    // ...
    onSaveClick = {
        coroutineScope.launch {
            viewModel.saveItem()
        }
    },
    //...
)
```

Notice that you did not use `viewModelScope.launch()` for `saveItem()` in the `ItemEntryViewModel.kt` file, but it is necessary for `ItemEntryBody()` when you call a repository method. You can only call suspend functions from a coroutine or another suspend function. The function `viewModel.saveItem()` is a suspend function.

5. Build and run your app.
6. Tap the + FAB.
7. In the **Add Item** screen, add the item details and tap **Save**. Notice that tapping the **Save** button does not close the **Add Item** screen.



Item Name*: Strawberry

Item Price*: $ 17.98

Quantity in Stock*: 45

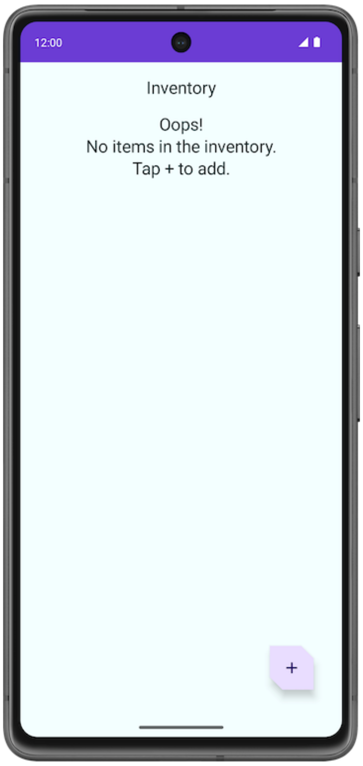
*required fields

8. In the `onSaveClick` lambda, add a call to `navigateBack()` after the call to `viewModel.saveItem()` to navigate back to the previous screen. Your `ItemEntryBody()` function looks like the following code:

```
ItemEntryBody(
    itemUiState = viewModel.itemUiState,
    onItemValueChange = viewModel::updateUiState,
    onSaveClick = {
        coroutineScope.launch {
            viewModel.saveItem()
            navigateBack()
        }
    },
    modifier = modifier.padding(innerPadding)
)
```

9. Run the app again and perform the same steps to enter and save the data. Notice that this time, the app navigates back to the **Inventory** screen.

This action saves the data, but you cannot see the inventory data in the app. In the next task, you use the Database Inspector to view the data you saved.



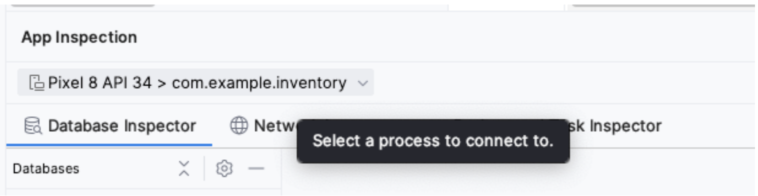
Oops!
No items in the inventory.
Tap + to add.

10. View the database content using Database Inspector

The Database Inspector lets you inspect, query, and modify your app's databases while your app runs. This feature is especially useful for database debugging. The Database Inspector works with plain SQLite and with libraries built on top of SQLite, such as Room. Database Inspector works best on emulators/devices running API level 26.

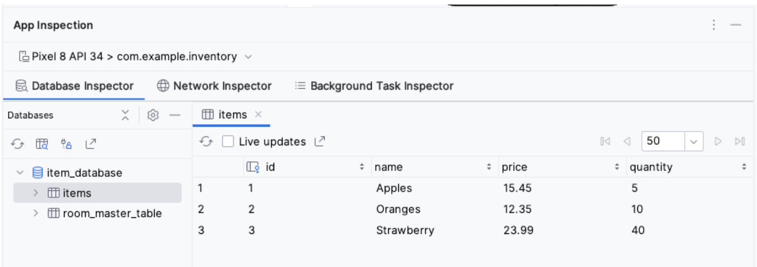
Note: The Database Inspector only works with the SQLite library included in the Android operating system on API level 26 and higher. It doesn't work with other SQLite libraries that you bundle with your app.

1. Run your app on an emulator or connected device running API level 26 or higher, if you have not done so already.
2. In Android Studio, select **View** > **Tool Windows** > **App Inspection** from the menu bar.
3. Select the **Database Inspector** tab.
4. In the **Database Inspector** pane, select the `com.example.inventory` from the dropdown menu if it is not already selected. The **item_database** in the Inventory app appears in the **Databases** pane.



Select a process to connect to.

5. Expand the node for the **item_database** in the **Databases** pane and select **Item** to inspect. If your **Databases** pane is empty, use your emulator to add some items to the database using the **Add Item** screen.
6. Check the **Live updates** checkbox in the Database Inspector to automatically update the data it presents as you interact with your running app in the emulator or device.



	id	name	price	quantity
1	1	Apples	15.45	5
2	2	Oranges	12.35	10
3	3	Strawberry	23.99	40

Congratulations! You created an app that can persist data using Room. In the next codelab, you will add a `lazyColumn` to your app to display the items on the database, and add new features to the app, like the ability to delete and update the entities. See you there!

11. Get the solution code

The solution code for this codelab is in the GitHub repo. To download the code for the finished codelab, use the following git commands:

```
$ git clone https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app.git
$ cd basic-android-kotlin-compose-training-inventory-app
$ git checkout room
```

Alternatively, you can download the repository as a zip file, unzip it, and open it in Android Studio.

https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app/archive/refs/heads/room.zip

Note: The solution code is in the `room` branch of the downloaded repository.

If you want to see the solution code for this codelab, view it on GitHub.

https://github.com/google-developer-training/basic-android-kotlin-compose-training-inventory-app/tree/room

12. Summary

- Define your tables as data classes annotated with `@Entity`. Define properties annotated with `@ColumnInfo` as columns in the tables.
- Define a data access object (DAO) as an interface annotated with `@Dao`. The DAO maps Kotlin functions to database queries.
- Use annotations to define `@Insert`, `@Delete`, and `@Update` functions.
- Use the `@Query` annotation with an SQLite query string as a parameter for any other queries.
- Use Database Inspector to view the data saved in the Android SQLite database.

13. Learn more

Android Developer Documentation

Save data in a local database using Room	https://developer.android.com/training/data-storage/room
androidx.room	https://developer.android.com/reference/androidx/room/package-summary
Debug your database with the Database Inspector	https://developer.android.com/studio/inspect/database

Blog posts

7 Pro-tips for Room	https://medium.com/androiddevelopers/7-pro-tips-for-room-fbadea4bfbd1
The one and only object. Kotlin Vocabulary	https://medium.com/androiddevelopers/the-one-and-only-object-5dfd2cf7ab9b

Videos

Kotlin: Using Room Kotlin APIs	https://www.youtube.com/watch?v=vsDkhRTMdA0
Database Inspector	https://www.youtube.com/watch?v=UMc7Tu0nKYQ

Other documentation and articles

Singleton pattern	https://en.wikipedia.org/wiki/Singleton_pattern
Companion objects	https://kotlinlang.org/docs/object-declarations.html#companion-objects

SQLite Tutorial - An Easy Way to Master SQLite Fast	https://www.sqlitetutorial.net/