

Read and update data with Room

1. Before you begin

You learned in the previous codelabs how to use a Room persistence library, an abstraction layer on top of a SQLite (<https://developer.android.com/training/data-storage/sqlite>) database, to store the app data. In this codelab, you'll add more features to the Inventory app and learn how to read, display, update, and delete data from the SQLite database using Room. You will use a `LazyColumn` to display the data from the database and automatically update the data when the underlying data in the database changes.

Prerequisites

- Ability to create and interact with the SQLite database using the Room library.
- Ability to create an entity, DAO, and database classes.
- Ability to use a data access object (DAO) to map Kotlin functions to SQL queries.
- Ability to display list items in a `LazyColumn` (<https://developer.android.com/jetpack/compose/lists>).
- Completion of the previous codelab in this unit, Persist data with Room.

What you'll learn

- How to read and display entities from a SQLite database.
- How to update and delete entities from a SQLite database using the Room library.

What you'll build

- An Inventory app that displays a list of inventory items and can update, edit, and delete items from the app database using Room.

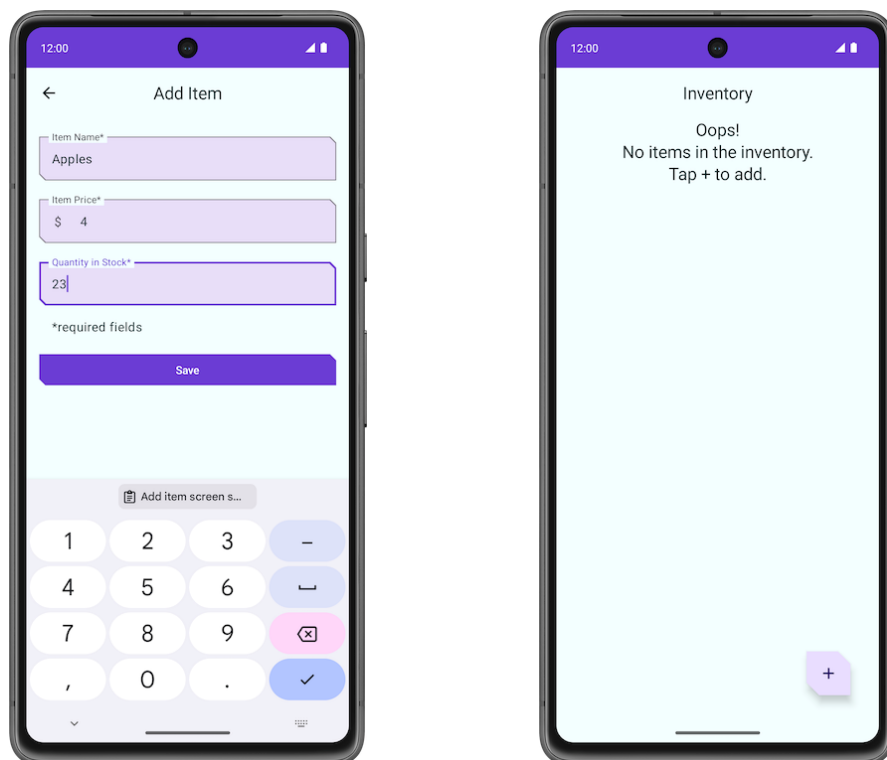
What you'll need

- A computer with Android Studio

2. Starter app overview

This codelab uses the Inventory app solution code from the previous codelab, Persist data with Room as the starter code. The starter app already saves data with the Room (<https://developer.android.com/reference/androidx/room/package-summary>) persistence library. The user can use the **Add Item** screen to add data to the app database.

Note: The current version of the starter app doesn't display the data stored in the database.



In this codelab, you extend the app to read and display the data, and update and delete entities on the database using a Room library.

Download the starter code for this codelab

To get started, download the starter 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 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>

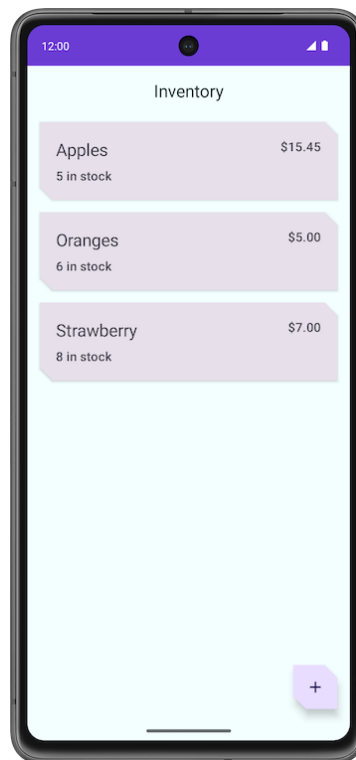
Note: The starter code is in the `room` branch of the downloaded repository.

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

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

3. Update UI state

In this task, you add a `LazyColumn` to the app to display the data stored in the database.



HomeScreen composable function walkthrough

- Open the `ui/home/HomeScreen.kt` file and look at the `HomeScreen()` composable.

```
@Composable
fun HomeScreen(
    navigateToItemEntry: () -> Unit,
```

```


        navigateToItemUpdate: (Int) -> Unit,
        modifier: Modifier = Modifier,
    ) {
        val scrollBehavior =
            TopAppBarDefaults.enterAlwaysScrollBehavior()

        Scaffold(
            topBar = {
                // Top app with app title
            },
            floatingActionButton = {
                FloatingActionButton(
                    // onClick details
                ) {
                    Icon(
                        // Icon details
                    )
                }
            },
        ) { innerPadding ->

            // Display List header and List of Items
            HomeBody(
                itemList = listOf(), // Empty list is being
                passed in for itemList
                onItemClick = navigateToItemUpdate,
                modifier = modifier.padding(innerPadding)
                    .fillMaxSize()
            )
        }
    }

```

This composable function displays the following items:

- The top app bar with the app title
- The floating action button (FAB) for the addition of new items to inventory 
- The `HomeBody()` composable function

The `HomeBody()` composable function displays inventory items based on the passed in list. As part of the starter code implementation, an empty list (`listOf()`) is passed to the `HomeBody()` composable function. To pass the inventory list to this composable, you must retrieve the inventory data from the repository and pass it into the `HomeViewModel`.

Emit UI state in the `HomeViewModel`

When you added methods to `ItemDao` to get items- `getItem()` and `getAllItems()` - you specified a `Flow` as the return type. Recall that a `Flow` represents a generic stream of data. By returning a `Flow`, you only need to explicitly call the methods from the DAO once for a given lifecycle. Room handles updates to the underlying data in an asynchronous manner.

Getting data from a flow is called *collecting from a flow*. When collecting from a flow in your UI layer, there are a few things to consider.

- Lifecycle events like configuration changes, for example rotating the device, causes the activity to be recreated. This causes recomposition and collecting from your `Flow` all over again.
- You want the values to be cached as state so that existing data isn't lost between lifecycle events.
- Flows should be canceled if there's no observers left, such as after a composable's lifecycle ends.

The recommended way to expose a `Flow` from a `ViewModel` is with a `StateFlow`. Using a `StateFlow` allows the data to be saved and observed, regardless of the UI lifecycle. To convert a `Flow` to a `StateFlow`, you use the `stateIn` operator.

The `stateIn` operator has three parameters which are explained below:

- `scope` - The `viewModelScope` defines the lifecycle of the `StateFlow`. When the `viewModelScope` is canceled, the `StateFlow` is also canceled.
- `started` - The pipeline should only be active when the UI is visible. The `SharingStarted.WhileSubscribed()` is used to accomplish this. To configure a delay (in milliseconds) between the disappearance of the last subscriber and the stopping of the sharing coroutine, pass in the `TIMEOUT_MILLIS` to the `SharingStarted.WhileSubscribed()` method.
- `initialValue` - Set the initial value of the state flow to `HomeUiState()`.

Once you've converted your `Flow` into a `StateFlow`, you can collect it using the `collectAsState()` method, converting its data into `State` of the same type.

In this step, you'll retrieve all items in the Room database as a `StateFlow` observable API for UI state. When the Room Inventory data changes, the UI updates automatically.

1. Open the `ui/home/HomeViewModel.kt` file, which contains a `TIMEOUT_MILLIS` constant and a `HomeUiState` data class with a list of items as a constructor parameter.

```
// No need to copy over, this code is part of starter code
class HomeViewModel : ViewModel() {
```

```

        companion object {
            private const val TIMEOUT_MILLIS = 5_000L
        }
    }

    data class HomeUiState(val itemList: List<Item> = listOf())

```

2. Inside the `HomeViewModel` class, declare a `val` called `homeUiState` of the type `StateFlow<HomeUiState>`. You will resolve the initialization error shortly.

```
val homeUiState: StateFlow<HomeUiState>
```

3. Call `getAllItemsStream()` on `itemsRepository` and assign it to `homeUiState` you just declared.

```
val homeUiState: StateFlow<HomeUiState> =
    itemsRepository.getAllItemsStream()
```

You now get an error - Unresolved reference: `itemsRepository`. To resolve the Unresolved reference error, you need to pass in the `ItemsRepository` object to the `HomeViewModel`.

4. Add a constructor parameter of the type `ItemsRepository` to the `HomeViewModel` class.

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

class HomeViewModel(itemsRepository: ItemsRepository):
    ViewModel() {

```

5. In the `ui/AppViewModelProvider.kt` file, in the `HomeViewModel` initializer, pass the `ItemsRepository` object as shown.

```
initializer {
    HomeViewModel(inventoryApplication().container.itemsRepository)
}

```

6. Go back to the `HomeViewModel.kt` file. Notice the type mismatch error. To resolve this, add a transformation map as shown below.

```
val homeUiState: StateFlow<HomeUiState> =
    itemsRepository.getAllItemsStream().map { HomeUiState(it)
}

```

Android Studio still shows you a type mismatch error. This error is because `homeUiState` is of the type `StateFlow` and `getAllItemsStream()` returns a `Flow`.

7. Use the `stateIn` operator to convert the `Flow` into a `StateFlow`.
The `StateFlow` is the observable API for UI state, which enables the UI to update itself.

```
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn

val homeUiState: StateFlow<HomeUiState> =
    itemsRepository.getAllItemsStream().map { HomeUiState(it)
    }
        .stateIn(
            scope = viewModelScope,
            started =
SharingStarted.WhileSubscribed(TIMEOUT_MILLIS),
            initialValue = HomeUiState()
        )
```

8. Build the app to make sure there are no errors in the code. There will not be any visual changes.

4. Display the Inventory data

In this task, you collect and update the UI state in the `HomeScreen`.

1. In the `HomeScreen.kt` file, in the `HomeScreen` composable function, add a new function parameter of the type `HomeViewModel` and initialize it.

```
import androidx.lifecycle.viewModel.compose.viewModel
import com.example.inventory.ui.AppViewModelProvider

@Composable
fun HomeScreen(
    navigateToItemEntry: () -> Unit,
    navigateToItemUpdate: (Int) -> Unit,
    modifier: Modifier = Modifier,
    viewModel: HomeViewModel = viewModel(factory =
AppViewModelProvider.Factory)
)
```

2. In the `HomeScreen` composable function, add a `val` called `homeUiState` to collect the UI state from the `HomeViewModel`. You use `collectAsState()`, which collects values from this `StateFlow` (<https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/index.html>) and represents its latest value via `State` (<https://developer.android.com/reference/kotlin/androidx/compose/runtime/State>).

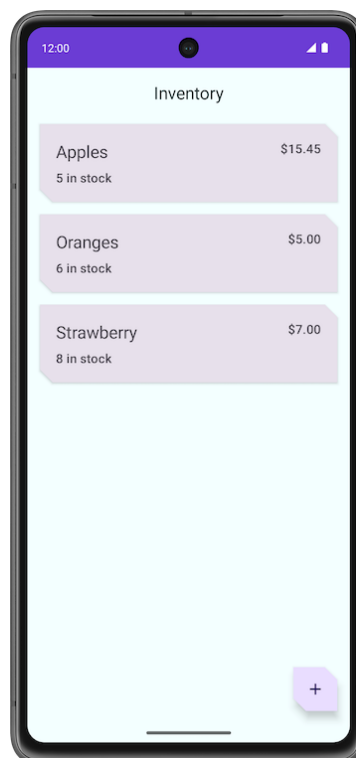
```
import androidx.compose.runtime.collectAsState
import androidx.compose.runtime.getValue

val homeUiState by viewModel.homeUiState.collectAsState()
```

3. Update the `HomeBody()` function call and pass in `homeUiState.itemList` to the `itemList` parameter.

```
HomeBody(
    itemList = homeUiState.itemList,
    onItemClick = navigateToItemUpdate,
    modifier = modifier.padding(innerPadding)
)
```

4. Run the app. Notice that the inventory list displays if you saved items in your app database. If the list is empty, add some inventory items to the app database.



5. Test your database

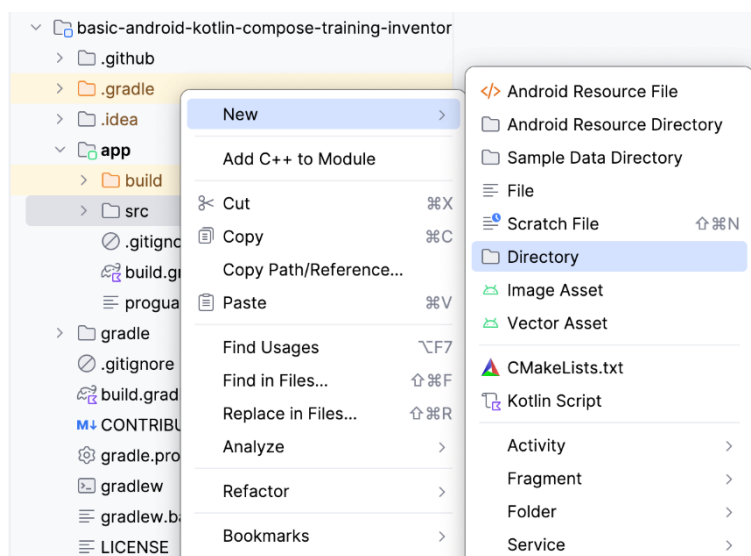
Previous codelabs discuss the importance of testing your code. In this task, you add some unit tests to test your DAO queries, and then you add more tests as you progress through the codelab.

The recommended approach for testing your database implementation is writing a JUnit test that runs on an Android device. Because these tests don't require creating an activity, they are faster to execute than your UI tests.

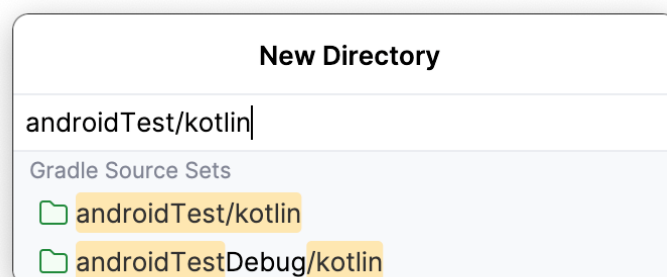
1. In the `build.gradle.kts` (Module :app) file, notice the following dependencies for Espresso and JUnit.

```
// Testing
androidTestImplementation("androidx.test.espresso:espresso-core:3.5.1")
androidTestImplementation("androidx.test.ext:junit:1.1.5")
```

2. Switch to **Project** view and right-click on `src` > **New** > **Directory** to create a test source set for your tests.



3. Select **androidTest/kotlin** from the **New Directory** popup.



4. Create a Kotlin class called `ItemDaoTest.kt`.

5. Annotate the `ItemDaoTest` class with `@RunWith(AndroidJUnit4::class)`. Your class now looks something like the following example code:

```
package com.example.inventory

import androidx.test.ext.junit.runners.AndroidJUnit4
import org.junit.runner.RunWith

@RunWith(AndroidJUnit4::class)
class ItemDaoTest {
}
```

6. Inside the class, add private `var` variables of the type `ItemDao` and `InventoryDatabase`.

```
import com.example.inventory.data.InventoryDatabase
import com.example.inventory.data.ItemDao

private lateinit var itemDao: ItemDao
private lateinit var inventoryDatabase: InventoryDatabase
```

7. Add a function to create the database and annotate it with `@Before` so that it can run before every test.
8. Inside the method, initialize `itemDao`.

```
import android.content.Context
import androidx.room.Room
import androidx.test.core.app.ApplicationProvider
import org.junit.Before

@Before
fun createDb() {
    val context: Context =
        ApplicationProvider.getApplicationContext()
    // Using an in-memory database because the information
    // stored here disappears when the
    // process is killed.
    inventoryDatabase = Room.inMemoryDatabaseBuilder(context,
        InventoryDatabase::class.java)
        // Allowing main thread queries, just for testing.
        .allowMainThreadQueries()
        .build()
    itemDao = inventoryDatabase.itemDao()
}
```

In this function, you use an in-memory database and do not persist it on the disk. To do so, you use the `inMemoryDatabaseBuilder()` function. You do this because the information need not be persisted, but rather, needs to be deleted when the process is killed. You are running the DAO queries in the main thread with `.allowMainThreadQueries()`, just for testing.

9. Add another function to close the database. Annotate it with `@After` to close the database and run after every test.

```
import org.junit.After
import java.io.IOException

@After
@Throws(IOException::class)
fun closeDb() {
    inventoryDatabase.close()
}
```

10. Declare items in the class `ItemDaoTest` for the database to use, as shown in the following code example:

```
import com.example.inventory.data.Item

private var item1 = Item(1, "Apples", 10.0, 20)
private var item2 = Item(2, "Bananas", 15.0, 97)
```

11. Add utility functions to add one item, and then two items, to the database. Later, you use these functions in your test. Mark them as `suspend` so they can run in a coroutine.

```
private suspend fun addOneItemToDb() {
    itemDao.insert(item1)
}

private suspend fun addTwoItemsToDb() {
    itemDao.insert(item1)
    itemDao.insert(item2)
}
```

12. Write a test for inserting a single item into the database, `insert()`. Name the test `daoInsert_insertsItemIntoDB` and annotate it with `@Test`.

```
import kotlinx.coroutines.flow.first
import kotlinx.coroutines.runBlocking
import org.junit.Assert.assertEquals
import org.junit.Test
```

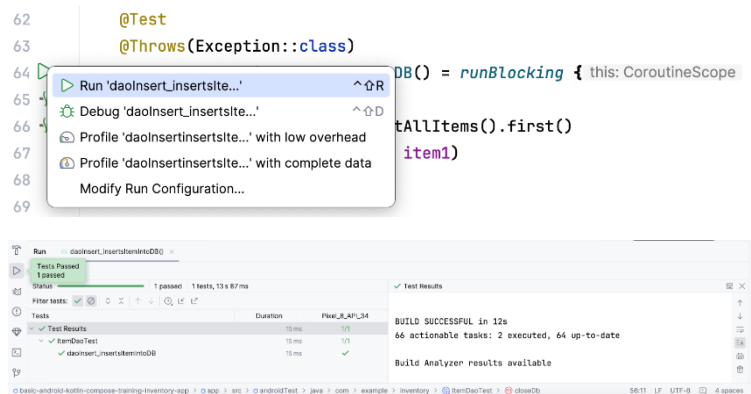
```

@Test
@Throws(Exception::class)
fun daoInsert_insertsItemIntoDB() = runBlocking {
    addOneItemToDb()
    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], item1)
}

```

In this test, you use the utility function `addOneItemToDb()` to add one item to the database. Then, you read the first item in the database. With `assertEquals()`, you compare the expected value with the actual value. You run the test in a new coroutine with `runBlocking{}`. This setup is the reason you mark the utility functions as suspend.

13. Run the test and make sure it passes.



14. Write another test for `getAllItems()` from the database. Name the test `daoGetAllItems_returnsAllItemsFromDB`.

```

@Test
@Throws(Exception::class)
fun daoGetAllItems_returnsAllItemsFromDB() = runBlocking {
    addTwoItemsToDb()
    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], item1)
    assertEquals(allItems[1], item2)
}

```

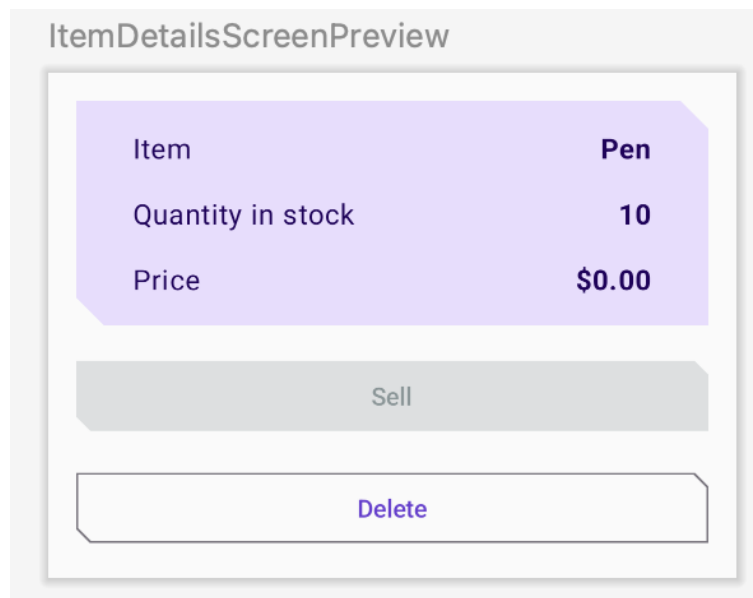
In the above test, you add two items to the database inside a coroutine. Then you read the two items and compare them with the expected values.

6. Display item details

In this task, you read and display the entity details on the **Item Details** screen. You use the item UI state, such as name, price, and quantity from the inventory app database and display them on the **Item Details** screen with the `ItemDetailsScreen` composable. The `ItemDetailsScreen` composable function is prewritten for you and contains three `Text` composables that display the item details.

ui/item/ItemDetailsScreen.kt

This screen is part of the starter code and displays the details of the items, which you see in a later codelab. You do not work on this screen in this codelab. The `ItemDetailsViewModel.kt` is the corresponding `ViewModel` for this screen.



1. In the `HomeScreen` composable function, notice the `HomeBody()` function call. `navigateToItemUpdate` is being passed to the `onItemClick` parameter, which gets called when you click on any item in your list.

```
// No need to copy over
HomeBody(
    itemList = homeUiState.itemList,
    onItemClick = navigateToItemUpdate,
    modifier = modifier
        .padding(innerPadding)
        .fillMaxSize()
)
```

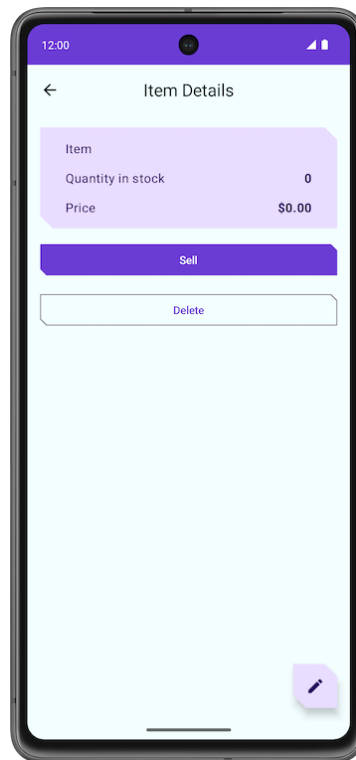
2. Open `ui/navigation/InventoryNavGraph.kt` and notice the `navigateToItemUpdate` parameter in the `HomeScreen` composable.

This parameter specifies the destination for navigation as the item details screen.

```
// No need to copy over
HomeScreen(
    navigateToItemEntry = {
        navController.navigate(ItemEntryDestination.route) },
    navigateToItemUpdate = {
        navController.navigate("${ItemDetailsDestination.route}
}/${it}")
    }
)
```

This part of the `onItemClick` functionality is already implemented for you. When you click the list item, the app navigates to the item details screen.

3. Click any item in the inventory list to see the item details screen with empty fields.



To fill the text fields with item details, you need to collect the UI state in `ItemDetailsScreen()`.

4. In `UI/Item/ItemDetailsScreen.kt`, add a new parameter to the `ItemDetailsScreen` composable of the type `ItemDetailsViewModel` and use the factory method to initialize it.

```
import androidx.lifecycle.viewmodel.compose.viewModel
import com.example.inventory.ui.AppViewModelProvider
```

```

@Composable
fun ItemDetailsScreen(
    navigateToEditItem: (Int) -> Unit,
    navigateBack: () -> Unit,
    modifier: Modifier = Modifier,
    viewModel: ItemDetailsViewModel = viewModel(factory =
AppViewModelProvider.Factory)
)

```

5. Inside the `ItemDetailsScreen()` composable, create a `val` called `uiState` to collect the UI state. Use `collectAsState()` to collect `uiState` `StateFlow` and represent its latest value via `State`. Android Studio displays an unresolved reference error.

```

import androidx.compose.runtime.collectAsState

val uiState = viewModel.uiState.collectAsState()

```

6. To resolve the error, create a `val` called `uiState` of the type `StateFlow<ItemDetailsUiState>` in the `ItemDetailsViewModel` class.
7. Retrieve the data from the item repository and map it to `ItemDetailsUiState` using the extension function `toItemDetails()`. The extension function `Item.toItemDetails()` is already written for you as part of the starter code.

```

import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.flow.SharingStarted
import kotlinx.coroutines.flow.StateFlow
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.flow.stateIn

val uiState: StateFlow<ItemDetailsUiState> =
    itemsRepository.getItemStream(itemId)
        .filterNotNull()
        .map {
            ItemDetailsUiState(itemDetails =
it.toItemDetails())
        }.stateIn(
            scope = viewModelScope,
            started =
SharingStarted.WhileSubscribed(TIMEOUT_MILLIS),
            initialValue = ItemDetailsUiState()
        )

```

8. Pass `ItemsRepository` into the `ItemDetailsViewModel` to resolve the Unresolved reference: `itemsRepository` error.

```
class ItemDetailsViewModel(  
    savedStateHandle: SavedStateHandle,  
    private val itemsRepository: ItemsRepository  
    ) : ViewModel() {
```

9. In `ui/AppViewModelProvider.kt`, update the initializer for `ItemDetailsViewModel` as shown in the following code snippet:

```
initializer {  
    ItemDetailsViewModel(  
        this.createSavedStateHandle(),  
        inventoryApplication().container.itemsRepository  
    )  
}
```

10. Go back to the `ItemDetailsScreen.kt` and notice the error in the `ItemDetailsScreen()` composable is resolved.
11. In the `ItemDetailsScreen()` composable, update the `ItemDetailsBody()` function call and pass in `uiState.value` to `itemUiState` argument.

```
ItemDetailsBody(  
    itemUiState = uiState.value,  
    onSellItem = { },  
    onDelete = { },  
    modifier = modifier.padding(innerPadding)  
)
```

12. Observe the implementations of `ItemDetailsBody()` and `ItemInputForm()`. You are passing the current selected item from `ItemDetailsBody()` to `ItemDetails()`.

```
// No need to copy over  
  
@Composable  
private fun ItemDetailsBody(  
    itemUiState: ItemUiState,  
    onSellItem: () -> Unit,  
    onDelete: () -> Unit,  
    modifier: Modifier = Modifier  
) {  
    Column(  
        //...  
    ) {
```



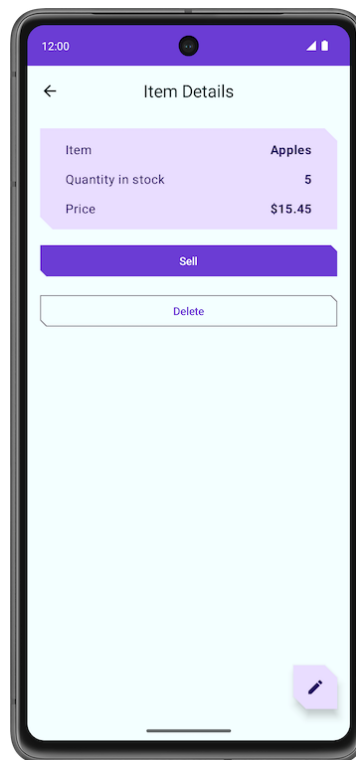
```

        var deleteConfirmationRequired by rememberSaveable {
mutableStateOf(false) }
        ItemDetails(
            item = itemDetailsUiState.itemDetails.toItem(),
modifier = Modifier.fillMaxWidth()
        )

        //...
    }

```

13. Run the app. When you click any list element on the **Inventory** screen, the **Item Details** screen displays.
14. Notice that the screen is not blank anymore. It displays the entity details retrieved from the inventory database.



15. Tap the **Sell** button. Nothing happens!

In the next section, you implement the functionality of the **Sell** button.

7. Implement Item details screen

ui/item/ItemEditScreen.kt

The Item edit screen is already provided to you as part of the starter code.

This layout contains text field composables to edit the details of any new inventory item.

ItemEntryScreenPreview

Item Name*

Item name

Item Price*

\$ 10.00

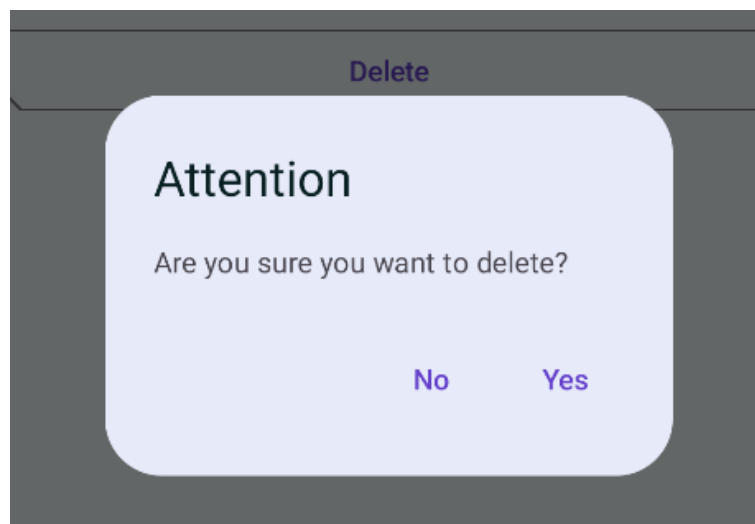
Quantity in Stock*

5

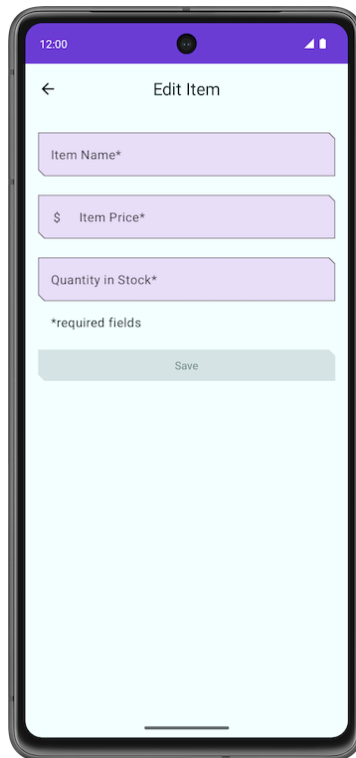
*required fields

Save

The code for this app still isn't fully functional. For example, in the **Item Details** screen, when you tap the **Sell** button, the **Quantity in Stock** does not decrease. When you tap the **Delete** button, the app does prompt you with a confirmation dialog. However, when you select the **Yes** button, the app does not actually delete the item.



Lastly, the FAB button  opens an empty **Edit Item** screen.



In this section, you implement the functionalities of **Sell**, **Delete** and the FAB buttons.

8. Implement sell item

In this section, you extend the features of the app to implement the sell functionality. This update involves the following tasks:

- Add a test for the DAO function to update an entity.
- Add a function in the `ItemDetailsViewModel` to reduce the quantity and update the entity in the app database.
- Disable the **Sell** button if the quantity is zero.

1. In `ItemDaoTest.kt`, add a function called `daoUpdateItems_updatesItemsInDB()` with no parameters. Annotate with `@Test` and `@Throws(Exception::class)`.

```
@Test
@Throws(Exception::class)
fun daoUpdateItems_updatesItemsInDB()
```

2. Define the function and create a `runBlocking` block. Call `addTwoItemsToDb()` inside it.

```
fun daoUpdateItems_updatesItemsInDB() = runBlocking {
    addTwoItemsToDb()
}
```

3. Update the two entities with different values, calling `itemDao.update`.

```
itemDao.update(Item(1, "Apples", 15.0, 25))
itemDao.update(Item(2, "Bananas", 5.0, 50))
```

4. Retrieve the entities with `itemDao.getAllItems()`. Compare them to the updated entity and assert.

```
val allItems = itemDao.getAllItems().first()
assertEquals(allItems[0], Item(1, "Apples", 15.0, 25))
assertEquals(allItems[1], Item(2, "Bananas", 5.0, 50))
```

5. Make sure the completed function looks like the following:

```
@Test
@Throws(Exception::class)
fun daoUpdateItems_updatesItemsInDB() = runBlocking {
    addTwoItemsToDb()
    itemDao.update(Item(1, "Apples", 15.0, 25))
    itemDao.update(Item(2, "Bananas", 5.0, 50))

    val allItems = itemDao.getAllItems().first()
    assertEquals(allItems[0], Item(1, "Apples", 15.0, 25))
    assertEquals(allItems[1], Item(2, "Bananas", 5.0, 50))
}
```

6. Run the test and make sure it passes.

Add a function in the ViewModel

1. In `ItemDetailsViewModel.kt`, inside the `ItemDetailsViewModel` class, add a function called `reduceQuantityByOne()` with no parameters.

```
fun reduceQuantityByOne() {
}
```

2. Inside the function, start a coroutine with `viewModelScope.launch{}`.

Note: You must run database operations inside a coroutine.

```
import kotlinx.coroutines.launch
import androidx.lifecycle.viewModelScope
```

```
viewModelScope.launch {  
}
```

3. Inside the `launch` block, create a `val` called `currentItem` and set it to `uiState.value.toItem()`.

```
val currentItem = uiState.value.toItem()
```

The `uiState.value` is of the type `ItemUiState`. You convert it to the `Item` entity type with the extension function `toItem()`.

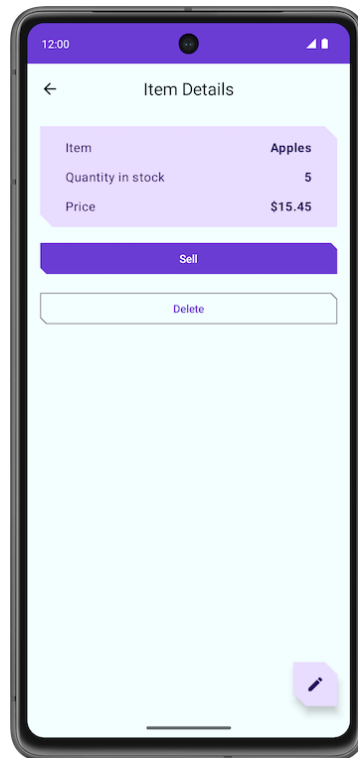
4. Add an `if` statement to check if the `quantity` is greater than 0.
5. Call `updateItem()` on `itemsRepository` and pass in the updated `currentItem`. Use `copy()` to update the `quantity` value so that the function looks like the following:

```
fun reduceQuantityByOne() {  
    viewModelScope.launch {  
        val currentItem = uiState.value.itemDetails.toItem()  
        if (currentItem.quantity > 0) {  
            itemsRepository.updateItem(currentItem.copy(quantity =  
currentItem.quantity - 1))  
        }  
    }  
}
```

6. Go back to `ItemDetailsScreen.kt`.
7. In the `ItemDetailsScreen` composable, go to the `ItemDetailsBody()` function call.
8. In the `onSellItem` lambda, call `viewModel.reduceQuantityByOne()`.

```
ItemDetailsBody(  
    itemUiState = uiState.value,  
    onSellItem = { viewModel.reduceQuantityByOne() },  
    onDelete = { },  
    modifier = modifier.padding(innerPadding)  
)
```

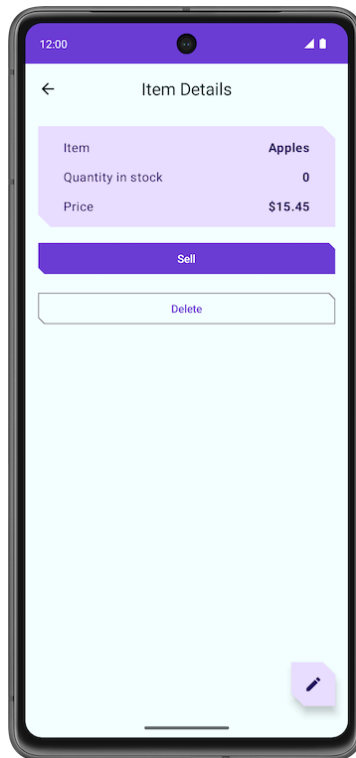
9. Run the app.
10. On the **Inventory** screen, click a list element. When the **Item Details** screen displays, tap the **Sell** and notice that the `quantity` value decreases by one.



11. In the `Item Details` screen, continuously tap the **Sell** button until the quantity is zero.

Tip: To save time, you might want to use an item for this task with a low quantity. If none of your items have a low quantity, you can create a new one with a low quantity.

After the quantity reaches zero, tap **Sell** again. There is no visual change because the function `reduceQuantityByOne()` checks if the quantity is greater than zero before updating the quantity.

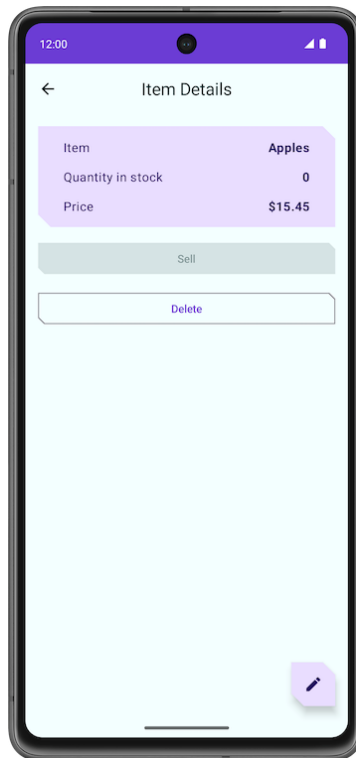


To give users better feedback, you might want to disable the **Sell** button when there is no item to sell.

12. In the `ItemDetailsViewModel` class, set `outOfStock` value based on the `it.quantity` in the `map` transformation.

```
val uiState: StateFlow<ItemDetailsUiState> =
    itemsRepository.getItemStream(itemId)
        .filterNotNull()
        .map {
            ItemDetailsUiState(outOfStock = it.quantity <= 0,
            itemDetails = it.toItemDetails())
        }.stateIn(
            //...
        )
```

13. Run your app. Notice that the app disables the **Sell** button when the quantity in stock is zero.



Congratulations on implementing the **Sell** item feature in your app!

Delete item entity

As with the previous task, you must extend the features of your app further by implementing delete functionality. This feature is much easier to implement than the sell feature. The process involves the following tasks:

- Add a test for the delete DAO query.
- Add a function in the `ItemDetailsViewModel` class to delete an entity from the database.
- Update the `ItemDetailsBody` composable.

Add DAO test

1. In `ItemDaoTest.kt`, add a test called `daoDeleteItems_deletesAllItemsFromDB()`.

```
@Test
@Throws(Exception::class)
fun daoDeleteItems_deletesAllItemsFromDB()
```

2. Launch a coroutine with `runBlocking {}`.


```
fun daoDeleteItems_deletesAllItemsFromDB() = runBlocking {  
}
```

3. Add two items to the database and call `itemDao.delete()` on those two items to delete them from the database.

```
addTwoItemsToDb()  
itemDao.delete(item1)  
itemDao.delete(item2)
```

4. Retrieve the entities from the database and check that the list is empty. The completed test should look like the following:

```
import org.junit.Assert.assertTrue  
  
@Test  
@Throws(Exception::class)  
fun daoDeleteItems_deletesAllItemsFromDB() = runBlocking {  
    addTwoItemsToDb()  
    itemDao.delete(item1)  
    itemDao.delete(item2)  
    val allItems = itemDao.getAllItems().first()  
    assertTrue(allItems.isEmpty())  
}
```

Add delete function in the `ItemDetailsViewModel`

1. In `ItemDetailsViewModel`, add a new function called `deleteItem()` that takes no parameters and returns nothing.
2. Inside the `deleteItem()` function, add an `itemsRepository.deleteItem()` function call and pass in `uiState.value.toItem()`.

```
suspend fun deleteItem() {  
    itemsRepository.deleteItem(uiState.value.itemDetails.toItem())  
}
```

In this function, you convert the `uiState` from `itemDetails` type to `Item` entity type using the `toItem()` extension function.

3. In the `ui/item/ItemDetailsScreen` composable, add a `val` called `coroutineScope` and set it to `rememberCoroutineScope()`. This approach returns a coroutine scope bound to the composition where it's called (`ItemDetailsScreen` composable).

```
import androidx.compose.runtime.rememberCoroutineScope

val coroutineScope = rememberCoroutineScope()
```

4. Scroll to the `ItemDetailsBody()` function.
5. Launch a coroutine with `coroutineScope` inside the `onDelete` lambda.
6. Inside the `launch` block, call the `deleteItem()` method on `viewModel`.

```
import kotlinx.coroutines.launch

ItemDetailsBody(
    itemUiState = uiState.value,
    onSellItem = { viewModel.reduceQuantityByOne() },
    onDelete = {
        coroutineScope.launch {
            viewModel.deleteItem()
        }
    },
    modifier = modifier.padding(innerPadding)
)
```

7. After deleting the item, navigate back to the inventory screen.
8. Call `navigateBack()` after the `deleteItem()` function call.

```
onDelete = {
    coroutineScope.launch {
        viewModel.deleteItem()
        navigateBack()
    }
}
```

9. Still within the `ItemDetailsScreen.kt` file, scroll to the `ItemDetailsBody()` function.

This function is part of the starter code. This composable displays an alert dialog to get the user's confirmation before deleting the item and calls the `deleteItem()` function when you tap Yes.

```
// No need to copy over

@Composable
private fun ItemDetailsBody(
    itemUiState: ItemUiState,
    onSellItem: () -> Unit,
    onDelete: () -> Unit,
    modifier: Modifier = Modifier
) {
    Column(
        /*...*/
    )
}
```

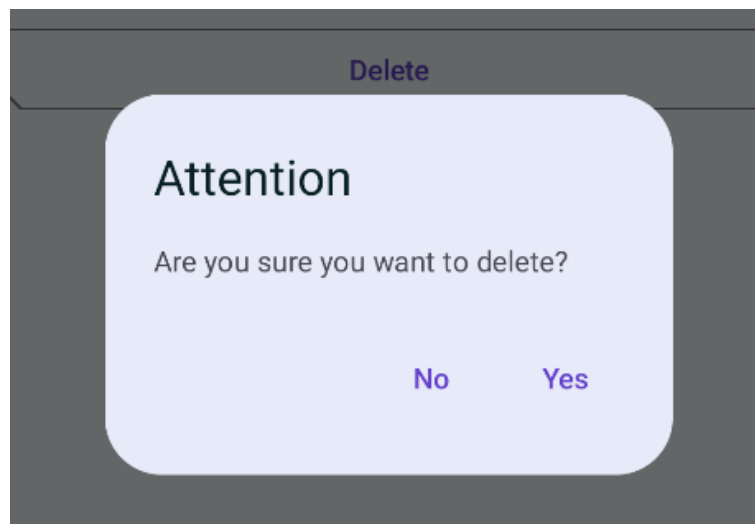
```

    ) {
        //...

        if (deleteConfirmationRequired) {
            DeleteConfirmationDialog(
                onDeleteConfirm = {
                    deleteConfirmationRequired = false
                    onDelete()
                },
                //...
            )
        }
    }
}

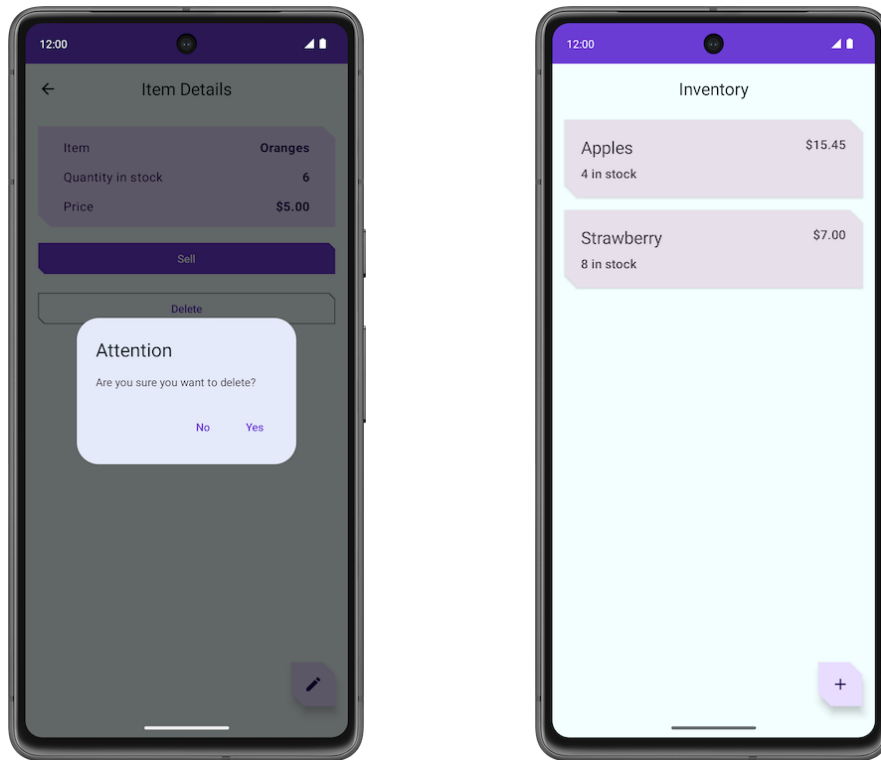
```

When you tap **No**, the app closes the alert dialog. The `showConfirmationDialog()` function displays the following alert:



10. Run the app.
11. Select a list element on the **Inventory** screen.
12. In the **Item Details** screen, tap **Delete**.
13. Tap **Yes** in the alert dialog, and the app navigates back to the **Inventory** screen.
14. Confirm that the entity you deleted is no longer in the app database.

Congratulations on implementing the delete feature!



Edit item entity

Similar to the previous sections, in this section, you add another feature enhancement to the app that edits an item entity.

Here is a quick run through of the steps to edit an entity in the app database:

- Add a test to the test get item DAO query.
- Populate the text fields and the **Edit Item** screen with the entity details.
- Update the entity in the database using Room.

Add DAO test

1. In **ItemDaoTest.kt**, add a test called **daoGetItem_returnsItemFromDB()**.

```
@Test
@Throws(Exception::class)
fun daoGetItem_returnsItemFromDB()
```

2. Define the function. Inside the coroutine, add one item to the database.

```
@Test
@Throws(Exception::class)
fun daoGetItem_returnsItemFromDB() = runBlocking {
```

```
        addOneItemToDb()  
    }
```

3. Retrieve the entity from the database using the `itemDao.getItem()` function and set it to a `val` named `item`.

```
val item = itemDao.getItem(1)
```

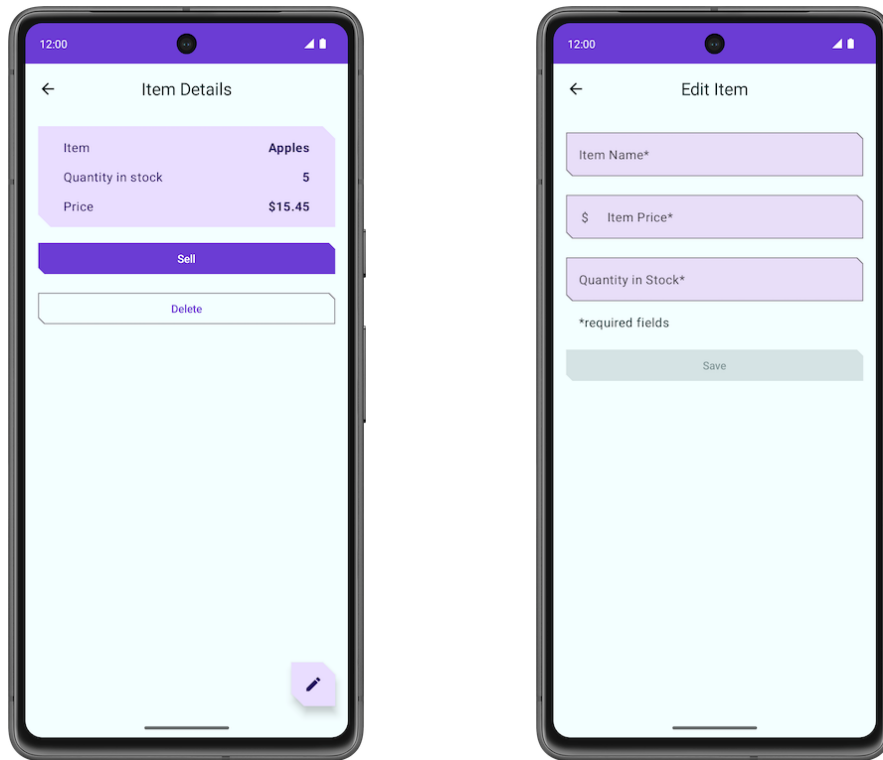
4. Compare the actual value with the retrieved value and assert using `assertEquals()`. Your completed test looks like the following:

```
@Test  
@Throws(Exception::class)  
fun daoGetItem_returnsItemFromDB() = runBlocking {  
    addOneItemToDb()  
    val item = itemDao.getItem(1)  
    assertEquals(item.first(), item1)  
}
```

5. Run the test and ensure it passes.

Populate text fields

If you run the app, go to the **Item Details** screen, and then click the FAB, you can see that the title of the screen now is **Edit Item**. However, all the text fields are empty. In this step, you populate the text fields in the **Edit Item** screen with the entity details.



1. In `ItemDetailsScreen.kt`, scroll to the `ItemDetailsScreen` composable.
2. In `FloatingActionButton()`, change the `onClick` argument to include `uiState.value.itemDetails.id`, which is the `id` of the selected entity. You use this `id` to retrieve the entity details.

```
FloatingActionButton(  
    onClick = {  
        navigateToEditItem(uiState.value.itemDetails.id) },  
    modifier = /*...*/  
)
```

3. In the `ItemEditViewModel` class, add an `init` block.

```
init {  
  
}
```

4. Inside the `init` block, launch a coroutine with `viewModelScope.launch`.

```
import kotlinx.coroutines.launch  
  
viewModelScope.launch { }
```

5. Inside the `launch` block, retrieve the entity details with `itemsRepository.getItemStream(itemId)`.

```
import androidx.lifecycle.ViewModelScope
import kotlinx.coroutines.flow.filterNotNull
import kotlinx.coroutines.flow.first

init {
    viewModelScope.launch {
        itemUiState = itemsRepository.getItemStream(itemId)
            .filterNotNull()
            .first()
            .toItemUiState(true)
    }
}
```

In this launch block, you add a filter to return a flow that only contains values that are not null. With `toItemUiState()`, you convert the `item` entity to `ItemUiState`. You pass the `actionEnabled` value as `true` to enable the **Save** button.


To resolve the `Unresolved reference: itemsRepository` error, you need to pass in the `ItemsRepository` as a dependency to the view model.

6. Add a constructor parameter to the `ItemEditViewModel` class.

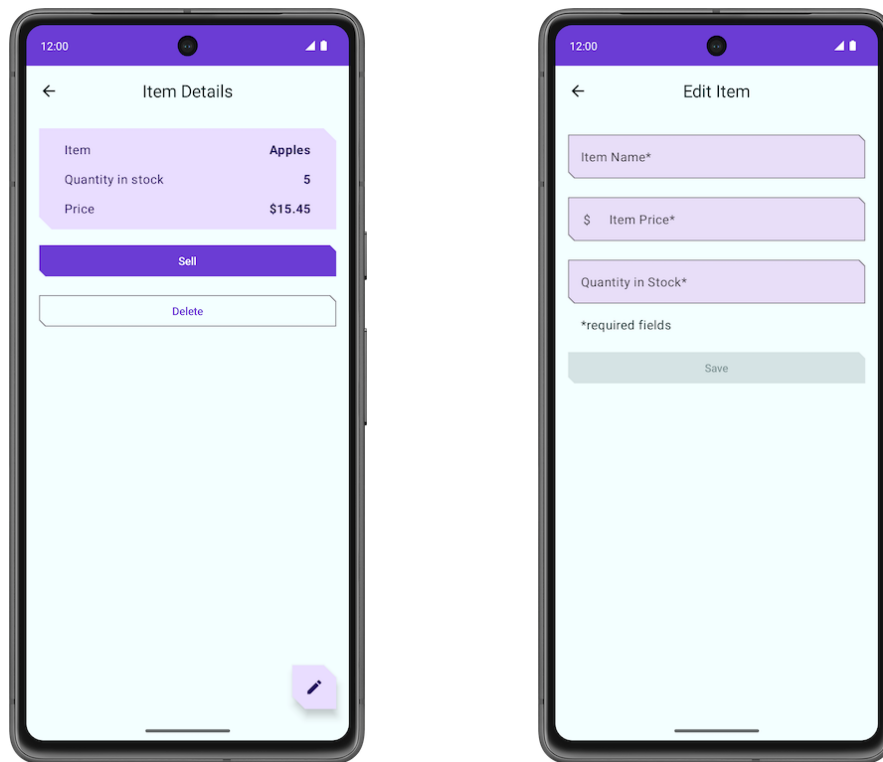
```
class ItemEditViewModel(
    savedStateHandle: SavedStateHandle,
    private val itemsRepository: ItemsRepository
)
```

7. In the `AppViewModelProvider.kt` file, in the `ItemEditViewModel` initializer, add the `ItemsRepository` object as an argument.

```
initializer {
    ItemEditViewModel(
        this.createSavedStateHandle(),
        inventoryApplication().container.itemsRepository
    )
}
```

8. Run the app.
9. Go to **Item Details** and tap  FAB.
10. Notice that the fields populate with the item details.
11. Edit the stock quantity, or any other field, and tap **Save**.

Nothing happens! This is because you are not updating the entity in the app database. You fix this in the next section.



Update the entity using Room

In this final task, you add the final pieces of the code to implement the update functionality. You define the necessary functions in the ViewModel and use them in the `ItemEditScreen`.

It's coding time again!

1. In `ItemEditViewModel` class, add a function called `updateUiState()` that takes an `ItemUiState` object and returns nothing. This function updates the `itemUiState` with new values that the user enters.

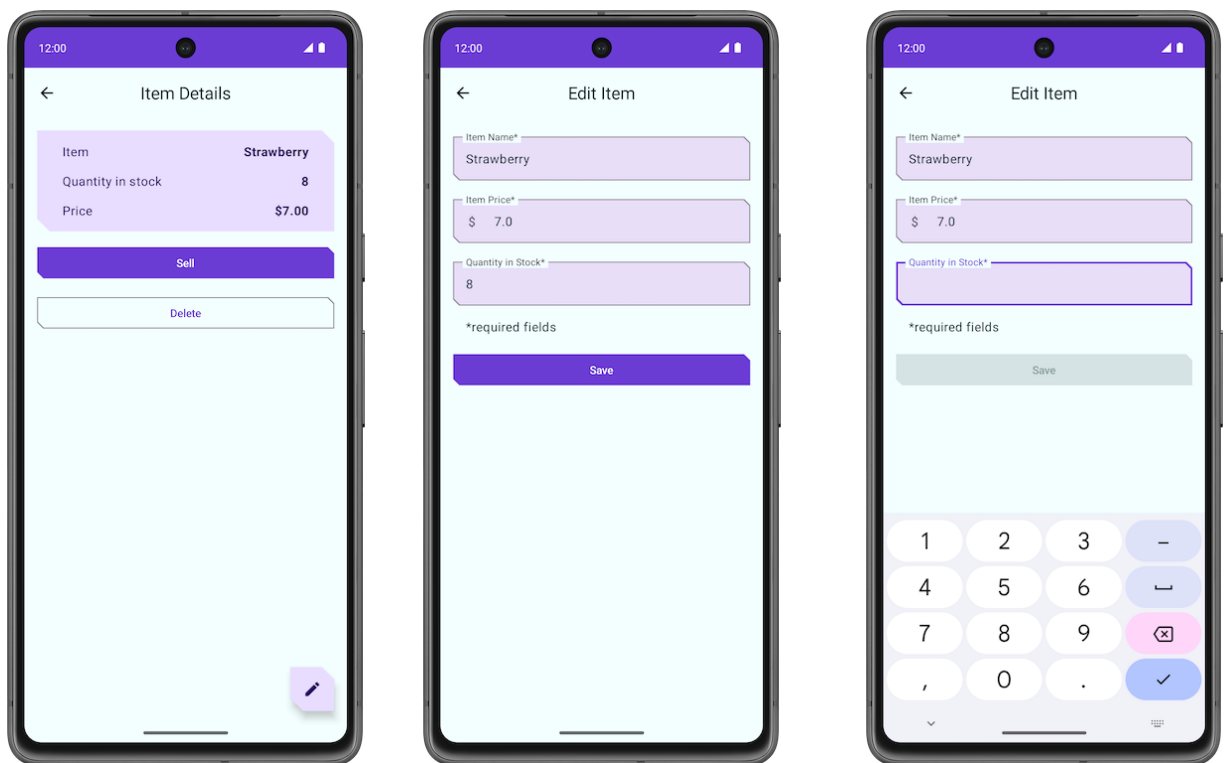
```
fun updateUiState(itemDetails: ItemDetails) {  
    itemUiState =  
        ItemUiState(itemDetails = itemDetails, isEntryValid =  
            validateInput(itemDetails))  
}
```

In this function, you assign the passed in `itemDetails` to the `itemUiState` and update the `isEntryValid` value. The app enables the **Save** button if `itemDetails` is `true`. You set this value to **true** only if the input that the user enters is valid.

2. Go to the `ItemEditScreen.kt` file.
3. In the `ItemEditScreen` composable, scroll down to the `ItemEntryBody()` function call.
4. Set the `onItemValueChange` argument value to the new function `updateUiState`.

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

5. Run the app.
6. Go to the **Edit Item** screen.
7. Make one of the entity values empty so that it is invalid. Notice how the **Save** button disables automatically.



8. Go back to the `ItemEditViewModel` class and add a suspend function called `updateItem()` that takes nothing. You use this function to save the updated entity to the Room database.

```
suspend fun updateItem() {
}
```

9. Inside the `getUpdatedItemEntry()` function, add an `if` condition to validate the user input by using the function `validateInput()`.
10. Make a call to the `updateItem()` function on the `itemsRepository`, passing in the `itemUiState.itemDetails.toItem()`. Entities that can be added to the Room database need to be of the type `Item`. The completed function looks like the following:

```
suspend fun updateItem() {  
    if (validateInput(itemUiState.itemDetails)) {  
        itemsRepository.updateItem(itemUiState.itemDetails.toItem())  
    }  
}
```

11. Go back to the `ItemEditScreen` composable. You need a coroutine scope to call the `updateItem()` function. Create a `val` called `coroutineScope` and set it to `rememberCoroutineScope()`.

```
import androidx.compose.runtime.rememberCoroutineScope  
  
val coroutineScope = rememberCoroutineScope()
```

12. In the `ItemEntryBody()` function call, update the `onSaveClick` function argument to start a coroutine in the `coroutineScope`.
13. Inside the `launch` block, call `updateItem()` on the `viewModel` and `navigate back`.

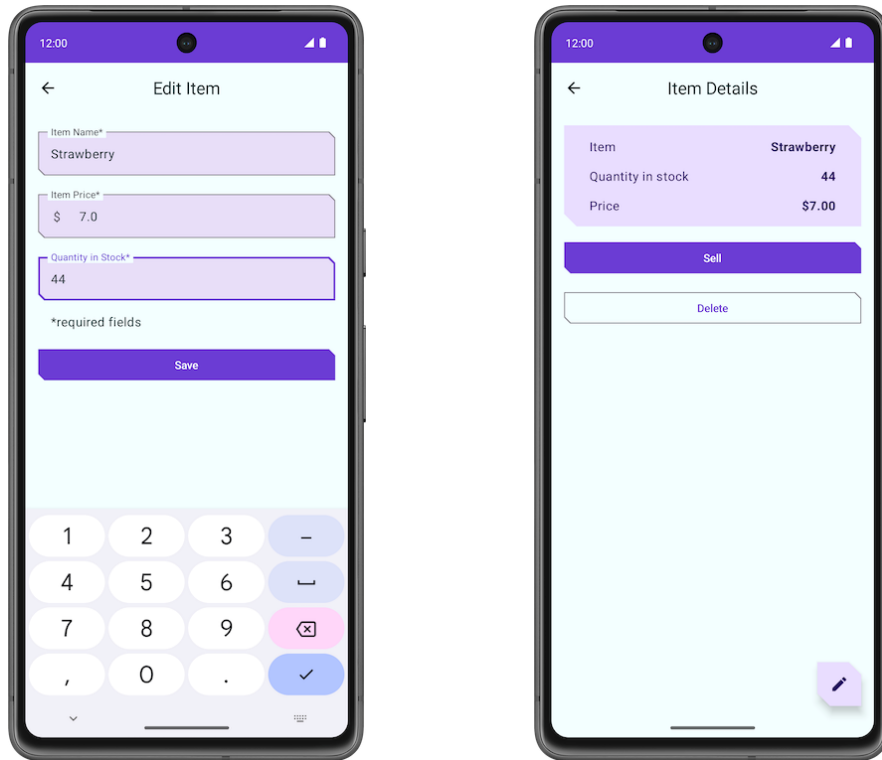
```
import kotlinx.coroutines.launch  
  
onSaveClick = {  
    coroutineScope.launch {  
        viewModel.updateItem()  
        navigateBack()  
    }  
},
```

The completed `ItemEntryBody()` function call looks like the following:

```
ItemEntryBody(  
    itemUiState = viewModel.itemUiState,  
    onItemValueChange = viewModel::updateUiState,  
    onSaveClick = {  
        coroutineScope.launch {  
            viewModel.updateItem()  
            navigateBack()  
        }  
    },
```

```
modifier = modifier.padding(innerPadding)
)
```

14. Run the app and try editing inventory items. You are now able to edit any item in the Inventory app database.



Congratulations on creating your first app that uses Room to manage the database!

9. Solution code

The solution code for this codelab is in the GitHub repo and the branch shown below:

Solution Code URL:

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

10. Learn more

Android Developer Documentation

Debug your database with the Database Inspector	https://developer.android.com/studio/inspect/database
Save data in a local database using Room	https://developer.android.com/training/data-storage/room
Test and debug your database Android Developers	https://developer.android.com/training/data-storage/room/testing-db

Kotlin references

Extensions	https://kotlinlang.org/docs/extensions.html
------------	---