Logotipo, nombre de la empresa

El contenido generado por IA puede ser incorrecto.

**INSTITUTO TECNOLÓGICO NACIONAL DE MÉXICO**

**INSTITUTO TECNOLÓGICO DE CIUDAD MADERO**

**PERSISTENCIA DE DATOS**

**LUIS FERNANDO HERNANDEZ MERCADO 20070531**

**HUGO ANDRE MARTINEZ GALVAN 20070545**

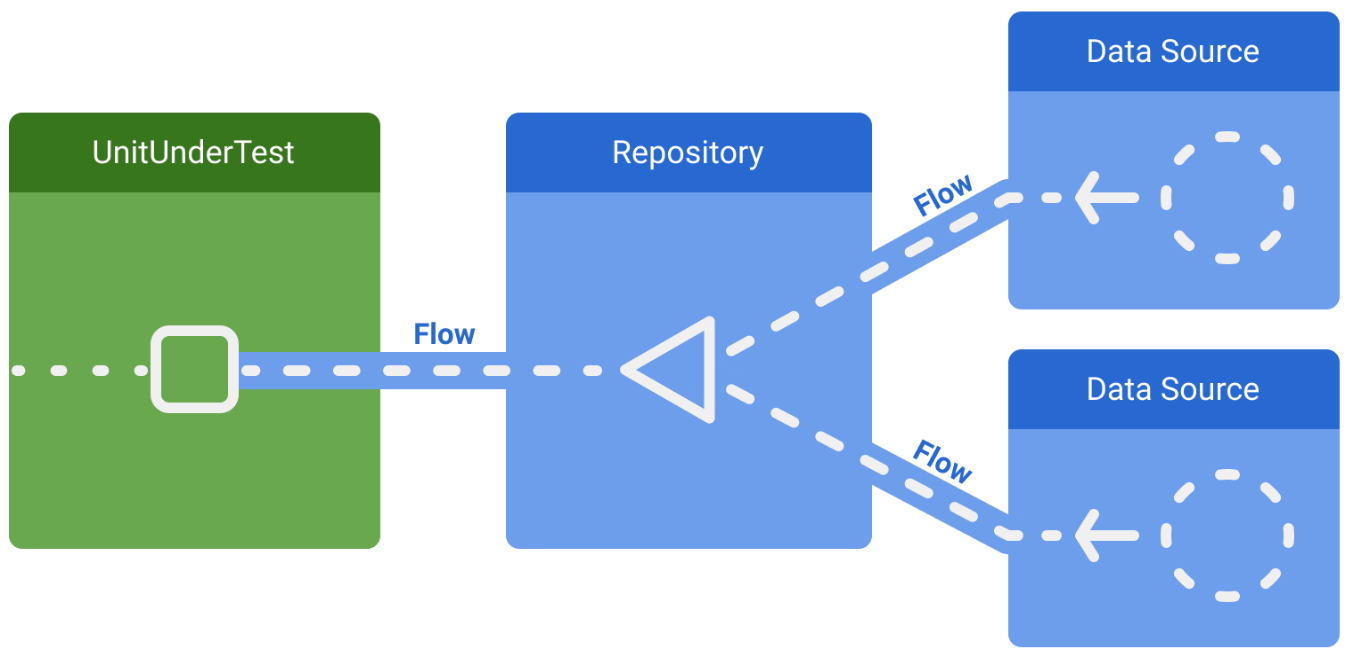
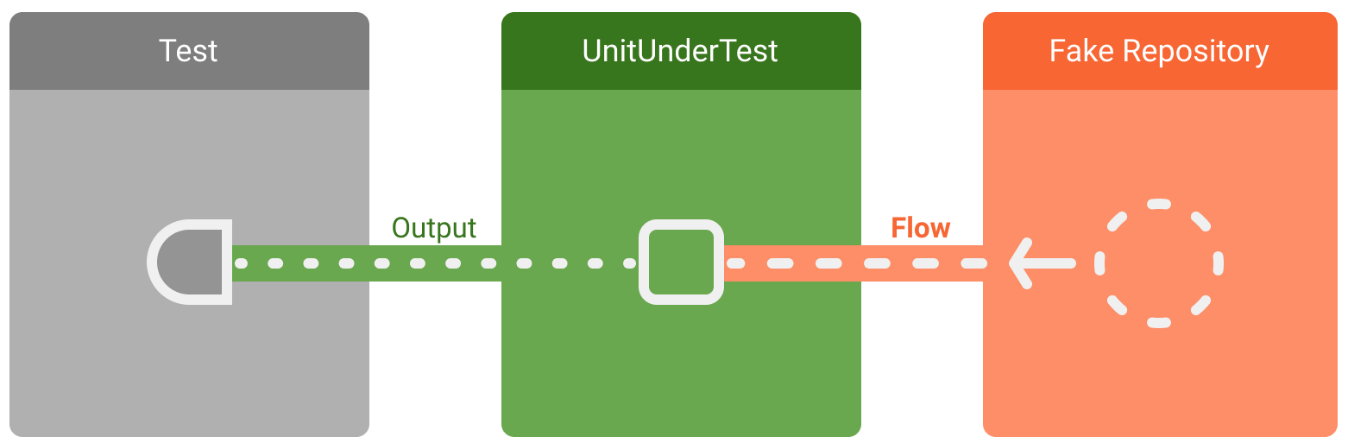
**Unidad 6: Persistencia de datos**

**Ruta de aprendizaje 2**

**Cómo usar Room para la persistencia de datos**

1. En este video aprendemos sobre las potentes y expresivas APIs de Flow y a aprovechar al máximo Flows en una app para Android.
2. Prueba de flujos

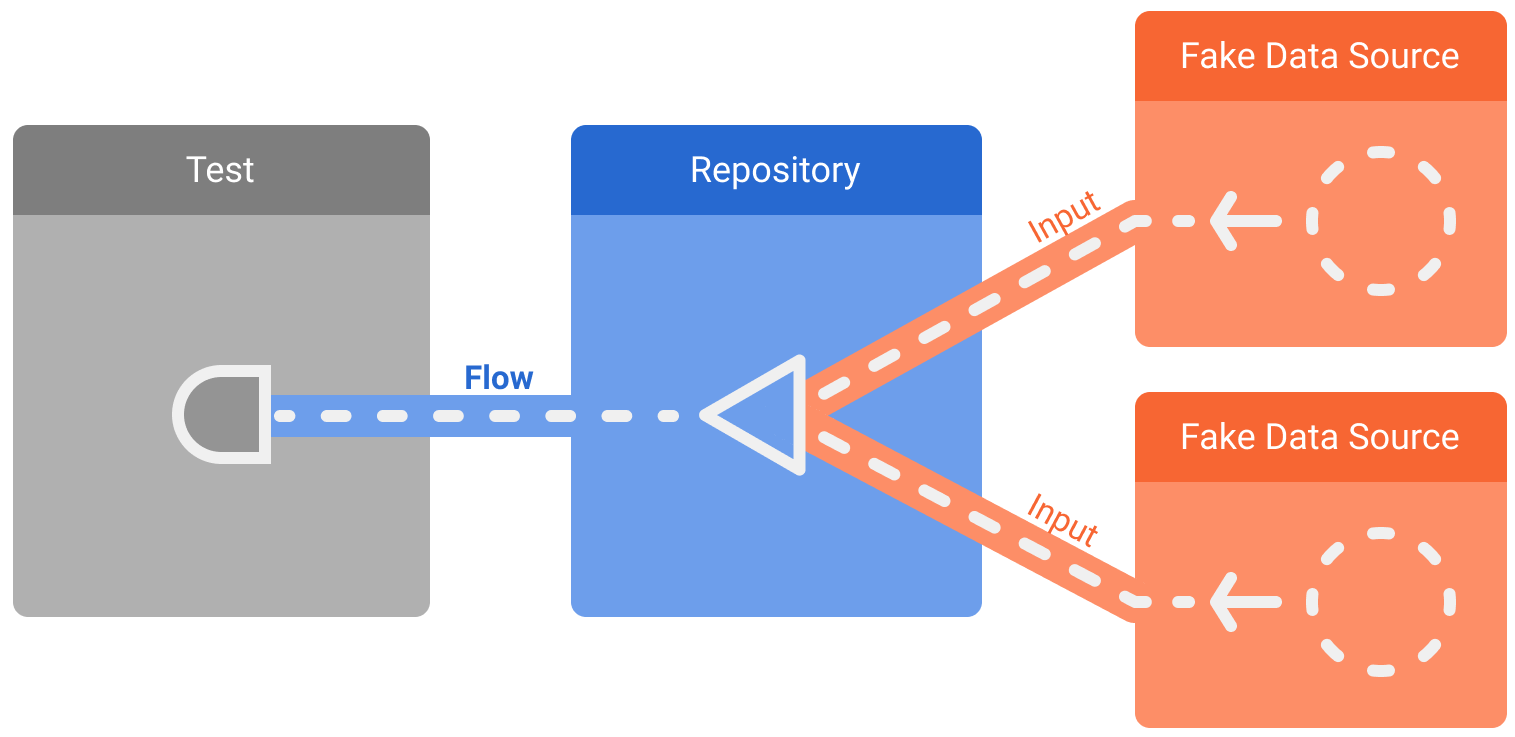
La forma en que pruebas las unidades o los módulos que se comunican con un [flujo](https://developer.android.com/kotlin/flow?hl=es-419) depende de si el sujeto de prueba usa el flujo como entrada o como salida.

* Si el sujeto de prueba observa un flujo, puedes generar flujos dentro de dependencias falsas que puedes controlar en las pruebas.
* Si la unidad o el módulo expone un flujo, puedes leer y verificar uno o varios elementos emitidos por el flujo en la prueba.
* **Cómo crear un productor falso**
* Cuando el sujeto de prueba es consumidor de un flujo, una forma común de probarlo es reemplazar el productor por una implementación falsa. Por ejemplo, en una determinada clase que observa un repositorio que toma datos de dos fuentes de datos en producción:
* **Figura 1:** El sujeto de prueba y la capa de datos.
* Para que la prueba sea determinante, puedes reemplazar el repositorio y sus dependencias por un repositorio falso que siempre emita los mismos datos falsos:
* **Figura 2:** Reemplazo de las dependencias por una implementación falsa.
* Para emitir una serie predefinida de valores en un flujo, usa el compilador flow:
* class MyFakeRepository : MyRepository {  
      fun observeCount() = flow {  
          emit(ITEM\_1)  
      }  
  }
* En la prueba, se inserta este repositorio falso para reemplazar la implementación real:
* @Test  
  fun myTest() {  
      // Given a class with fake dependencies:  
      val sut = MyUnitUnderTest(MyFakeRepository())  
      // Trigger and verify  
      ...  
  }
* Ahora que tienes el control de los resultados del sujeto de prueba, puedes verificar que funcione correctamente revisando sus resultados.

## Cómo confirmar las emisiones del flujo en una prueba

Si el sujeto de prueba expone un flujo, la prueba debe realizar aserciones sobre los elementos del flujo de datos.

Supongamos que el repositorio del ejemplo anterior expone un flujo:

**Figura 3:** Repositorio (sujeto de prueba) con dependencias falsas que expone un flujo.

Con ciertas pruebas, solo deberás verificar la primera emisión o una cantidad limitada de elementos provenientes del flujo.

Puedes consumir la primera emisión al flujo llamando a first(). Esta función espera hasta que se reciba el primer elemento y, luego, envía la señal de cancelación al productor.

@Test  
fun myRepositoryTest() = runTest {  
    // Given a repository that combines values from two data sources:  
    val repository = MyRepository(fakeSource1, fakeSource2)  
  
    // When the repository emits a value  
    val firstItem = repository.counter.first() // Returns the first item in the flow  
  
    // Then check it's the expected item  
    assertEquals(ITEM\_1, firstItem)  
}

Si la prueba necesita verificar varios valores, llamar a toList() hace que el flujo espere a que la fuente emita todos sus valores y, luego, los muestra en una lista. Esta opción solo funciona para flujos de datos finitos.

@Test  
fun myRepositoryTest() = runTest {  
    // Given a repository with a fake data source that emits ALL\_MESSAGES  
    val messages = repository.observeChatMessages().toList()  
  
    // When all messages are emitted then they should be ALL\_MESSAGES  
    assertEquals(ALL\_MESSAGES, messages)  
}

En el caso de flujos de datos que requieren una recopilación de elementos más compleja o que no muestran una cantidad limitada de elementos, puedes usar la API de Flow para seleccionar y transformar elementos. Estos son algunos ejemplos:

// Take the second item  
outputFlow.drop(1).first()  
  
// Take the first 5 items  
outputFlow.take(5).toList()  
  
// Takes the first item verifying that the flow is closed after that  
outputFlow.single()  
  
// Finite data streams  
// Verify that the flow emits exactly N elements (optional predicate)  
outputFlow.count()  
outputFlow.count(predicate)

### **Recopilación continua durante una prueba**

La recopilación de un flujo con toList(), como en el ejemplo anterior, usa collect() de forma interna y se suspende hasta que la lista completa de resultados esté preparada para mostrarse.

Para intercalar acciones que hagan que el flujo emita valores y aserciones sobre los valores que se emitieron, puedes recopilar valores de un flujo de manera continua durante una prueba.

Por ejemplo, toma la siguiente clase Repository para probar y una implementación de fuente de datos falsa complementaria que tenga un método emit para producir valores de forma dinámica durante la prueba:

class Repository(private val dataSource: DataSource) {  
    fun scores(): Flow<Int> {  
        return dataSource.counts().map { it \* 10 }  
    }  
}  
  
class FakeDataSource : DataSource {  
    private val flow = MutableSharedFlow<Int>()  
    suspend fun emit(value: Int) = flow.emit(value)  
    override fun counts(): Flow<Int> = flow  
}

Cuando uses esta implementación falsa en una prueba, podrás crear una corrutina de recopilación que reciba los valores de Repository de manera continua. En este ejemplo, los recopilamos en una lista y, luego, realizamos aserciones sobre su contenido:

@Test  
fun continuouslyCollect() = runTest {  
    val dataSource = FakeDataSource()  
    val repository = Repository(dataSource)  
  
    val values = mutableListOf<Int>()  
    backgroundScope.launch(UnconfinedTestDispatcher(testScheduler)) {  
        repository.scores().toList(values)  
    }  
  
    dataSource.emit(1)  
    assertEquals(10, values[0]) // Assert on the list contents  
  
    dataSource.emit(2)  
    dataSource.emit(3)  
    assertEquals(30, values[2])  
  
    assertEquals(3, values.size) // Assert the number of items collected  
}

Debido a que aquí no se completa el flujo que expone Repository, la llamada a toList que la recopila nunca se muestra. Iniciar la corrutina de recopilación en [TestScope.backgroundScope](https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-test/kotlinx.coroutines.test/-test-scope/background-scope.html) garantiza que esta se cancele antes de que finalice la prueba. De lo contrario, runTest seguiría esperando su finalización, lo que haría que la prueba dejara de responder y, al final, fallara.

Observa cómo se usa [UnconfinedTestDispatcher](https://developer.android.com/kotlin/coroutines/test?hl=es-419" \l "unconfinedtestdispatcher) aquí para la corrutina de recopilación. Esto garantiza que la corrutina de recopilación se inicie con anticipación y esté lista para recibir valores después de que se muestre launch.

#### **Cómo usar Turbine**

La biblioteca de terceros [Turbine](https://github.com/cashapp/turbine) ofrece una práctica API para crear una corrutina de recopilación, así como otras funciones útiles para probar flujos:

@Test  
fun usingTurbine() = runTest {  
    val dataSource = FakeDataSource()  
    val repository = Repository(dataSource)  
  
    repository.scores().test {  
        // Make calls that will trigger value changes only within test{}  
        dataSource.emit(1)  
        assertEquals(10, awaitItem())  
  
        dataSource.emit(2)  
        awaitItem() // Ignore items if needed, can also use skip(n)  
  
        dataSource.emit(3)  
        assertEquals(30, awaitItem())  
    }  
}

**Cómo probar StateFlows**

[StateFlow](https://developer.android.com/kotlin/flow/stateflow-and-sharedflow?hl=es-419#stateflow) es un contenedor de datos observables que se pueden recopilar para observar los valores que retienen con el tiempo como una transmisión continua. Ten en cuenta que esta transmisión continua de valores se combina, lo que significa que, si los valores se establecen en un StateFlow rápidamente, los recopiladores de ese StateFlow no tienen garantía de recibir todos los valores intermedios, solo el más reciente.

En las pruebas, si tienes en cuenta la combinación, puedes recopilar los valores de StateFlow de la misma manera que puedes recopilar cualquier otro flujo, incluso con Turbine. En algunas situaciones de prueba, te recomendamos que intentes recopilar y confirmar todos los valores intermedios.

Sin embargo, generalmente recomendamos tratar a StateFlow como un contenedor de datos y, en su lugar, confirmar en su propiedad value. De esta manera, las pruebas validan el estado actual del objeto en un momento dado y no dependen de si se produce o no la combinación.

Por ejemplo, toma este ViewModel que recopila valores de un Repository y los expone en la IU en un StateFlow:

class MyViewModel(private val myRepository: MyRepository) : ViewModel() {  
    private val \_score = MutableStateFlow(0)  
    val score: StateFlow<Int> = \_score.asStateFlow()  
  
    fun initialize() {  
        viewModelScope.launch {  
            myRepository.scores().collect { score ->  
                \_score.value = score  
            }  
        }  
    }  
}

**Nota:** Esta implementación de **ViewModel** usa directamente un **MutableStateFlow**. Para probar las instancias de **StateFlow** creadas con **stateIn**, consulta [la siguiente sección](https://developer.android.com/kotlin/flow/test?hl=es-419&continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-compose-unit-6-pathway-2%3Fhl%3Des-419%23article-https%3A%2F%2Fdeveloper.android.com%2Fkotlin%2Fflow%2Ftest#statein).

Una implementación falsa para este Repository podría verse de la siguiente manera:

class FakeRepository : MyRepository {  
    private val flow = MutableSharedFlow<Int>()  
    suspend fun emit(value: Int) = flow.emit(value)  
    override fun scores(): Flow<Int> = flow  
}

Cuando pruebas el ViewModel con esta implementación falsa, puedes emitir valores a partir de ella para activar las actualizaciones en el StateFlow del ViewModel y, luego, confirmar el value actualizado:

@Test  
fun testHotFakeRepository() = runTest {  
    val fakeRepository = FakeRepository()  
    val viewModel = MyViewModel(fakeRepository)  
  
    assertEquals(0, viewModel.score.value) // Assert on the initial value  
  
    // Start collecting values from the Repository  
    viewModel.initialize()  
  
    // Then we can send in values one by one, which the ViewModel will collect  
    fakeRepository.emit(1)  
    assertEquals(1, viewModel.score.value)  
  
    fakeRepository.emit(2)  
    fakeRepository.emit(3)  
    assertEquals(3, viewModel.score.value) // Assert on the latest value  
}

Cómo trabajar con StateFlows creados por stateIn

En la sección anterior, ViewModel usa un MutableStateFlow para almacenar el valor más reciente emitido por un flujo de Repository. Este es un patrón común, que se suele implementar de una manera más simple mediante el operador [stateIn](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/state-in.html), que convierte un flujo frío en un StateFlow caliente:

class MyViewModelWithStateIn(myRepository: MyRepository) : ViewModel() {  
    val score: StateFlow<Int> = myRepository.scores()  
        .stateIn(viewModelScope, SharingStarted.WhileSubscribed(5000L), 0)  
}

El operador stateIn tiene un parámetro SharingStarted que determina cuándo se activa y comienza a consumir el flujo subyacente. Las opciones como SharingStarted.Lazily y SharingStarted.WhileSubscribed se usan con frecuencia en los modelos de vista.

Incluso si confirmas en el value de StateFlow en tu prueba, deberás crear un recopilador. Puede ser un recopilador vacío:

@Test  
fun testLazilySharingViewModel() = runTest {  
    val fakeRepository = HotFakeRepository()  
    val viewModel = MyViewModelWithStateIn(fakeRepository)  
  
    // Create an empty collector for the StateFlow  
    backgroundScope.launch(UnconfinedTestDispatcher(testScheduler)) {  
        viewModel.score.collect {}  
    }  
  
    assertEquals(0, viewModel.score.value) // Can assert initial value  
  
    // Trigger-assert like before  
    fakeRepository.emit(1)  
    assertEquals(1, viewModel.score.value)  
  
    fakeRepository.emit(2)  
    fakeRepository.emit(3)  
    assertEquals(3, viewModel.score.value)  
}

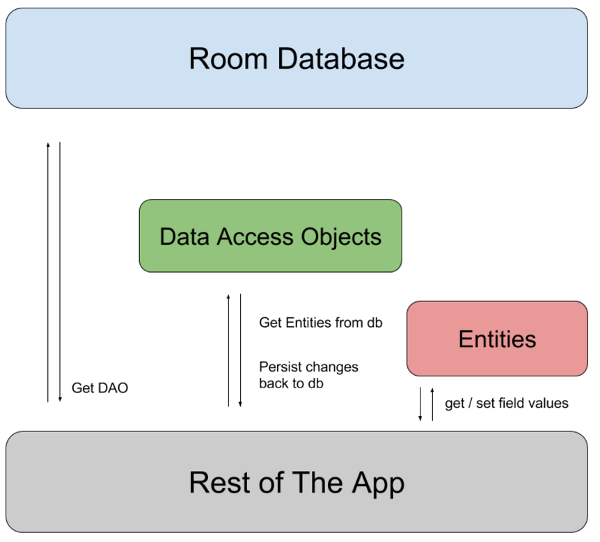
1. Room es una biblioteca de bases de datos que forma parte de Android Jetpack. Simplifica tu trabajo con las bases de datos para Android. En este video, Florina nos explicará cómo usar y probar las APIs de Room.
2. Cómo conservar datos con Room

Kotlin ofrece una manera fácil de trabajar con datos a través de clases de datos. Si bien es fácil trabajar con datos en la memoria usando clases de datos, cuando se trata de datos persistentes, debes convertirlos en un formato compatible con el almacenamiento de bases de datos. De este modo, necesitas *tablas* para almacenar los datos y *consultas* para acceder a ellos y modificarlos.

Los siguientes tres componentes de [Room](https://developer.android.com/topic/libraries/architecture/room?hl=es-419" \t "_blank) facilitan estos flujos de trabajo.

* Las [entidades de Room](https://developer.android.com/training/data-storage/room/defining-data?hl=es-419) representan tablas de la base de datos de tu app. Se usan para actualizar los datos almacenados en filas de las tablas y crear filas nuevas para insertarlas.
* Los [DAO](https://developer.android.com/training/data-storage/room/accessing-data?hl=es-419) de Room proporcionan métodos que tu app usa para recuperar, actualizar, insertar y borrar datos en la base de datos.
* La [clase de Database](https://developer.android.com/reference/kotlin/androidx/room/Database?hl=es-419) de Room es la clase de base de datos que proporciona a tu app instancias de los DAO asociados con esa base de datos.

Más adelante en este codelab, implementarás estos componentes y aprenderás más sobre ellos. En el siguiente diagrama, se muestra cómo los componentes de Room funcionan en conjunto para interactuar con la base de datos.



Agrega dependencias de Room

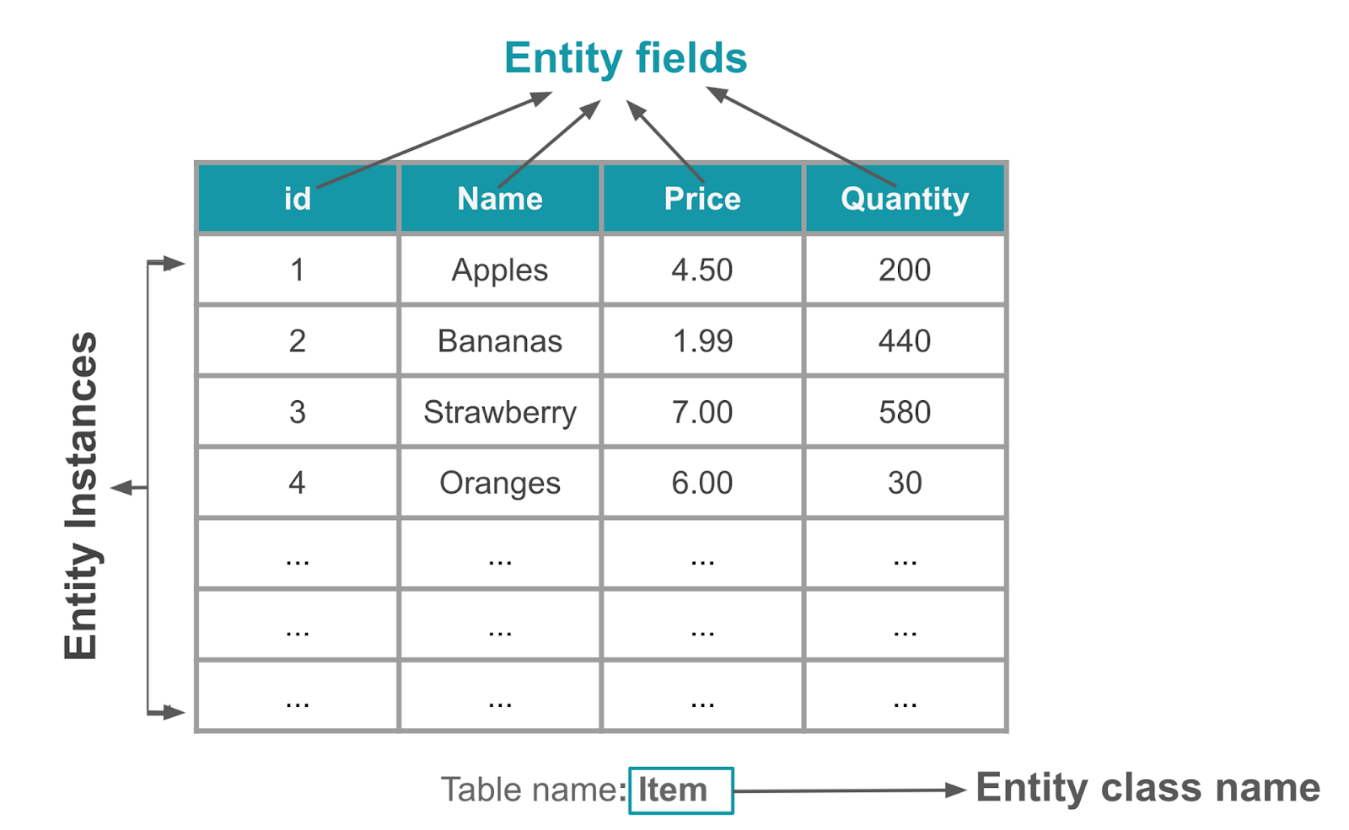
En esta tarea, agregarás las bibliotecas de componentes de Room necesarias a tus archivos Gradle.

1. Abre el archivo de Gradle de nivel de módulo build.gradle.kts (Module: InventoryApp.app).
2. En el bloque dependencies, agrega las dependencias para la biblioteca Room que se muestra en el siguiente código.

//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 es una API simple y potente para analizar anotaciones de Kotlin.

Una clase [Entity](https://developer.android.com/reference/androidx/room/Entity?hl=es-419" \t "_blank) define una tabla, y cada instancia de esta clase representa una fila en la tabla de la base de datos. Asimismo, tiene asignaciones para indicarle a Room cómo pretende presentar la información en la base de datos e interactuar con ella. En tu app, la entidad conserva información sobre los elementos del inventario, como el nombre, el precio y la cantidad disponible.



La anotación @Entity marca una clase como una clase Entity de base de datos. Para cada clase Entity, la app crea una tabla de base de datos que contenga los elementos. Cada campo de Entity se representa como una columna en la base de datos, a menos que se indique lo contrario (consulta la documentación sobre [Entity](https://developer.android.com/reference/androidx/room/Entity?hl=es-419" \t "_blank) para obtener más información). Cada instancia de entidad que se almacena en la base de datos debe tener una clave primaria. La [clave primaria](https://developer.android.com/reference/androidx/room/PrimaryKey?hl=es-419) se usa para identificar de manera única cada registro o entrada en las tablas de tu base de datos. Una vez que la app asigna una clave primaria, no se puede modificar. Representa el objeto de la entidad, siempre que exista en la base de datos.

En esta tarea, crearás una clase Entity y definirás campos para almacenar la siguiente información de inventario para cada elemento: Int para almacenar la clave primaria, String para almacenar el nombre del elemento, double para almacenar el precio del elemento y Int para almacenar la cantidad en stock.

1. Abre el código de partida en Android Studio.
2. Abre el paquete data en el paquete base com.example.inventory.
3. Dentro del paquete data, abre la clase de Kotlin Item, que representa una entidad de base de datos en tu 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  
)

**Nota**: Recuerda que el constructor principal es parte del encabezado de la clase en Kotlin. Se coloca después del nombre de la clase (y los parámetros de tipo opcional).

Clases de datos

Las clases de datos se usan principalmente para conservar datos en Kotlin. Se definen con la palabra clave data. Los objetos de clase de datos de Kotlin tienen algunos beneficios adicionales. Por ejemplo, el compilador genera automáticamente utilidades para comparar, imprimir y copiar elementos como toString(), [copy()](https://kotlinlang.org/docs/data-classes.html" \l "copying" \t "_blank) y equals().

**Ejemplo:**

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

Para garantizar la coherencia y el comportamiento significativo del código generado, las clases de datos deben cumplir con los siguientes requisitos:

* El constructor principal debe tener al menos un parámetro.
* Todos los parámetros del constructor principal deben ser val o var.
* Las clases de datos no pueden ser abstract, open ni sealed.

**Advertencia**: El compilador solo usa las propiedades definidas dentro del constructor principal para las funciones generadas automáticamente. El compilador excluye las propiedades declaradas dentro del cuerpo de la clase de las implementaciones generadas.

Para obtener más información sobre las clases de datos, consulta la [documentación correspondiente](https://kotlinlang.org/docs/data-classes.html).

1. Prefija la definición de la clase Item con la palabra clave data para convertirla en una clase de datos.

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

1. Sobre la declaración de clase Item, anota la clase de datos con @Entity. Usa el argumento tableName para establecer items como el nombre de la tabla de SQLite.

import androidx.room.Entity  
  
@Entity(tableName = "items")  
data class Item(  
   ...  
)

**Nota**: La anotación @Entity tiene varios argumentos posibles. De forma predeterminada (no hay argumentos para @Entity), el nombre de la tabla es el mismo que el nombre de la clase. Usa el argumento tableName para personalizar el nombre de la tabla. Para simplificar, usas un item. Hay muchos otros argumentos para @Entity que puedes investigar en la [documentación sobre entidades](https://developer.android.com/reference/androidx/room/Entity?hl=es-419).

1. Anota la propiedad id con @PrimaryKey para que id sea la clave primaria. Una clave primaria es un ID para identificar de manera única cada registro o entrada en la tabla Item.

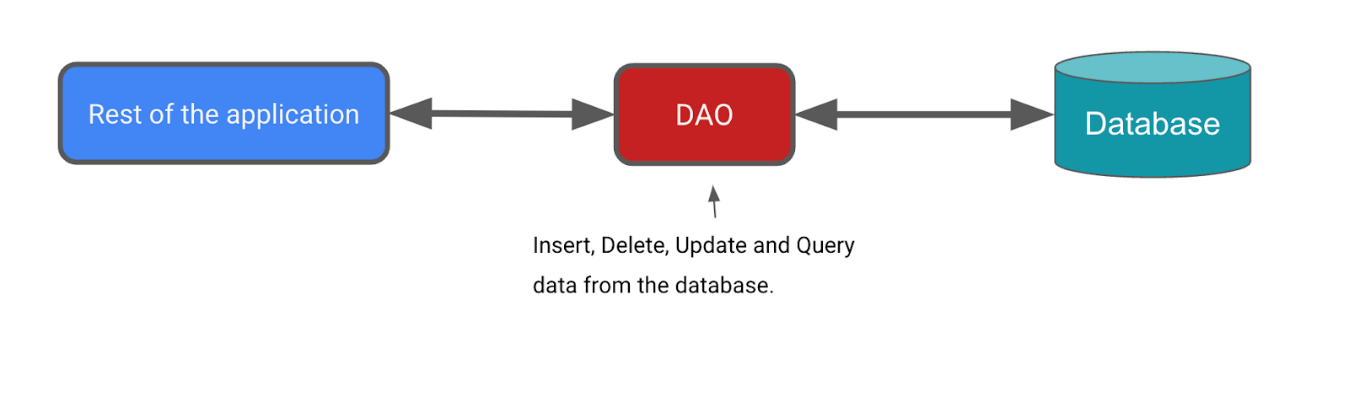
import androidx.room.PrimaryKey  
  
@Entity(tableName = "items")  
data class Item(  
    @PrimaryKey  
    val id: Int,  
    ...  
)

1. Asigna a id un valor predeterminado de 0, que es necesario para que id genere automáticamente valores de id.
2. Agrega el parámetro autoGenerate a la anotación @PrimaryKey para especificar si la columna de clave primaria se debe generar de forma automática. Si autoGenerate está configurado como true, Room generará automáticamente un valor único para la columna de clave primaria cuando se inserte una nueva instancia de entidad en la base de datos. Esto garantiza que cada instancia de la entidad tenga un identificador único, sin tener que asignar valores manualmente a la columna de clave primaria.

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

El [objeto de acceso a datos](https://developer.android.com/reference/androidx/room/Dao?hl=es-419) (DAO) es un patrón que puedes usar para separar la capa de persistencia del resto de la aplicación proporcionando una interfaz abstracta. Este aislamiento sigue el [principio de responsabilidad única](https://en.wikipedia.org/wiki/Single-responsibility_principle), que viste en los codelabs anteriores.

La funcionalidad del DAO es ocultar todas las complejidades relacionadas con la realización de operaciones de la base de datos en la capa de persistencia, aparte del resto de la aplicación. Esto te permite cambiar la capa de datos independientemente del código que usa los datos.



En esta tarea, definirás un DAO para Room. Los DAO son los componentes principales de Room que son responsables de definir la interfaz que accede a la base de datos.

El DAO que creas es una interfaz personalizada que proporciona métodos convenientes para consultar/recuperar, insertar, borrar y actualizar la base de datos. Room genera una implementación de esta clase en el tiempo de compilación.

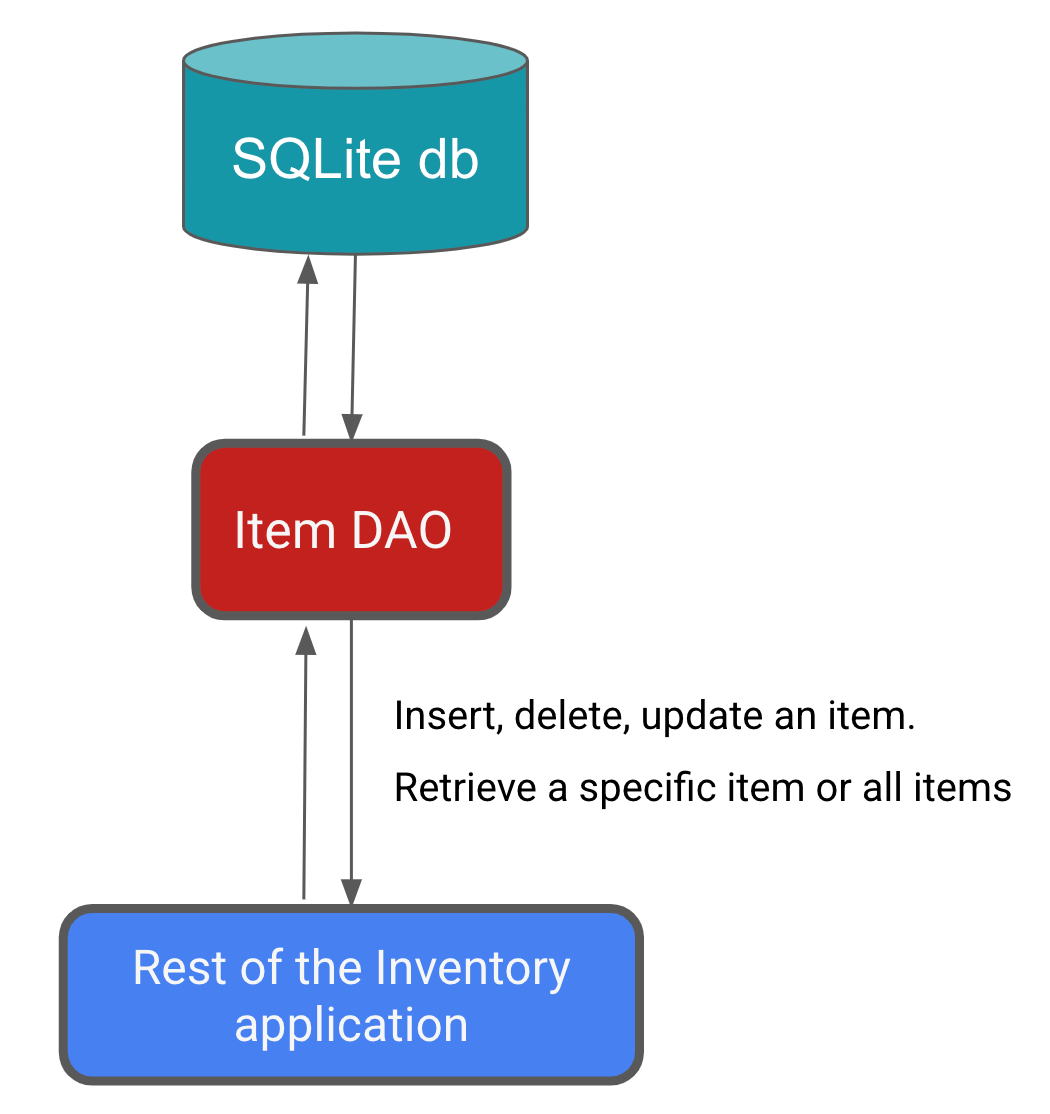
La biblioteca de Room proporciona anotaciones de conveniencia, como @Insert, @Delete y @Update, para definir métodos que realizan inserciones, actualizaciones y eliminaciones simples sin necesidad de escribir una instrucción de SQL.

Si necesitas definir operaciones más complejas para la inserción, actualización o eliminación, o si necesitas consultar los datos en la base de datos, usa una anotación @Query.

Como beneficio adicional, a medida que escribes tus consultas en Android Studio, el compilador comprueba si las consultas de SQL tienen errores de sintaxis.

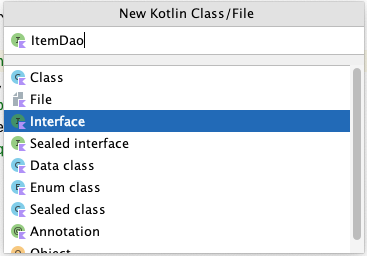
En el caso de la app de Inventory, debes poder hacer lo siguiente:

* **Insertar** o agregar un elemento nuevo
* **Actualizar** un elemento existente para actualizar el nombre, el precio y la cantidad
* **Obtener** un elemento específico según su clave primaria, id
* **Obtener todos los elementos** para que puedas mostrarlos
* **Borrar** una entrada de la base de datos



Completa los siguientes pasos para implementar el elemento DAO en tu app:

1. En el paquete data, crea la interfaz de Kotlin ItemDao.kt.



1. Anota la interfaz ItemDao con @Dao.

import androidx.room.Dao  
  
@Dao  
interface ItemDao {  
}

1. Dentro del cuerpo de la interfaz, agrega una anotación @Insert.
2. Debajo de @Insert, agrega una función insert() que tome una instancia del item de la clase Entity como su argumento.
3. Marca la función con la palabra clave suspend para permitir que se ejecute en un subproceso separado.

Las operaciones de la base de datos pueden demorar mucho tiempo en ejecutarse, por lo que deben hacerlo en un subproceso independiente. Room no permite el acceso a la base de datos en el subproceso principal.

import androidx.room.Insert  
  
@Insert  
suspend fun insert(item: Item)

Cuando se insertan elementos en la base de datos, se pueden generar conflictos. Por ejemplo, varios lugares en el código intentan actualizar la entidad con valores diferentes, en conflicto, como la misma clave primaria. Una entidad es una fila en DB. En la app de Inventory, solo insertamos la entidad desde un lugar que es la pantalla **Agregar elemento**, por lo que no esperamos que haya ningún conflicto y podemos establecer la estrategia de conflicto como *Ignorar*.

1. Agrega un argumento onConflict y asígnale un valor de OnConflictStrategy.*IGNORE*.

El argumento onConflict le indica a Room qué hacer en caso de conflicto. La estrategia OnConflictStrategy.*IGNORE* ignora un elemento nuevo.

Para obtener más información sobre las estrategias de conflicto disponibles, consulta la documentación de [OnConflictStrategy](https://developer.android.com/reference/androidx/room/OnConflictStrategy.html?hl=es-419" \t "_blank).

import androidx.room.OnConflictStrategy  
  
@Insert(onConflict = OnConflictStrategy.IGNORE)  
suspend fun insert(item: Item)

Ahora, Room genera todo el código necesario para insertar item en la base de datos. Cuando llamas a cualquiera de las funciones DAO que están marcadas con anotaciones de Room, Room ejecuta la consulta en SQL correspondiente en la base de datos. Por ejemplo, cuando llamas al método anterior, insert() desde tu código Kotlin, Room ejecuta una consulta en SQL para insertar la entidad en la base de datos.

1. Agrega una función nueva con la anotación @Update que tome un Item como parámetro.

La entidad que se actualiza tiene la misma clave primaria que la que se pasa. Puedes actualizar algunas o todas las demás propiedades de la entidad.

1. Al igual que con el método insert(), marca esta función con la palabra clave suspend.

import androidx.room.Update  
  
@Update  
suspend fun update(item: Item)

Agrega otra función con la anotación @Delete para borrar elementos y convertirla en una función de suspensión.

**Nota**: La anotación @Delete borra un elemento o una lista de elementos. Debes pasar las entidades que deseas borrar. Si no tienes la entidad, es posible que debas recuperarla antes de llamar a la función delete().

import androidx.room.Delete  
  
@Delete  
suspend fun delete(item: Item)

No hay ninguna anotación de conveniencia para la funcionalidad restante, por lo que debes usar la anotación @Query y proporcionar consultas de SQLite.

1. Escribe una consulta de SQLite para recuperar un elemento específico de la tabla de elementos según el id especificado. El siguiente código proporciona una consulta de muestra que selecciona todas las columnas de items, donde id coincide con un valor específico y id es un identificador único.

**Ejemplo:**

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

1. Agrega una anotación @Query.
2. Usa la consulta de SQLite del paso anterior como un parámetro de cadena a la anotación @Query.
3. Agrega un parámetro String a @Query, que es una consulta de SQLite para recuperar un elemento de la tabla correspondiente.

La consulta ahora indica que se seleccionen todas las columnas de items, donde id coincide con el argumento :id. Observa que :id usa la notación de dos puntos en la consulta para hacer referencia a argumentos en la función.

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

1. Después de la anotación @Query, agrega una función getItem() que tome un argumento Int y muestre un Flow<Item>.

import androidx.room.Query  
import kotlinx.coroutines.flow.Flow  
  
@Query("SELECT \* from items WHERE id = :id")  
fun getItem(id: Int): Flow<Item>

Se recomienda usar Flow en la capa de persistencia. Con Flow como el tipo de datos que se muestra, recibirás una notificación cada vez que cambien los datos de la base de datos. Room mantiene este Flow actualizado por ti, lo que significa que solo necesitas obtener los datos de forma explícita una vez. Esta configuración es útil para actualizar la lista de inventario, que implementarás en el siguiente codelab. Debido al tipo de datos que se muestra para Flow, Room también ejecuta la búsqueda en el subproceso en segundo plano. No necesitas convertirla de manera explícita en una función suspend ni llamar dentro del alcance de la corrutina.

**Nota**: Flow en la base de datos de Room puede mantener los datos *actualizados* ya que emite una notificación cada vez que cambien los datos de la base de datos. De esta manera, podrás observar los datos y actualizar tu IU según corresponda.

1. Agrega una @Query con una función getAllItems().
2. Haz que la consulta de SQLite muestre todas las columnas de la tabla item, ordenadas de forma ascendente.
3. Haz que getAllItems() muestre una lista de entidades Item como Flow. Room mantiene este Flow actualizado por ti, lo que significa que solo necesitas obtener los datos de forma explícita una vez.

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

ItemDao completado:

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>>  
}

La clase [Database](https://developer.android.com/reference/androidx/room/Database?hl=es-419" \t "_blank) proporciona a tu app instancias de los DAO que definas. A su vez, la app puede usar los DAO para recuperar datos de la base de datos como instancias de objetos de entidad de datos asociados. La app también puede usar las entidades de datos definidas para actualizar filas de las tablas correspondientes o crear filas nuevas para su inserción.

Debes crear una clase abstracta RoomDatabase y anotarla con @Database. Esta clase tiene un método que muestra la instancia existente de RoomDatabase si la base de datos no existe.

Este es el proceso general para obtener la instancia RoomDatabase:

* Crea una clase public abstract que extienda RoomDatabase. La nueva clase abstracta que defines actúa como un contenedor de la base de datos. La clase que defines es abstracta porque Room crea la implementación por ti.
* Anota la clase con @Database. En los argumentos, enumera las entidades para la base de datos y establece el número de versión.
* Define una propiedad o un método abstracto que muestre una instancia de ItemDao, y Room genera la implementación por ti.
* Solo necesitas una instancia de RoomDatabase para toda la app, así que haz que RoomDatabase sea un singleton.
* Usa el [Room.databaseBuilder](https://developer.android.com/reference/androidx/room/Room?hl=es-419" \l "databaseBuilder(android.content.Context,java.lang.Class,kotlin.String)" \t "_blank) de Room para crear tu base de datos (item\_database), solo si no existe. De lo contrario, muestra la base de datos existente.

Crea la base de datos

1. En el paquete data, crea una clase de Kotlin InventoryDatabase.kt.
2. En el archivo InventoryDatabase.kt, haz que la clase InventoryDatabase sea una clase abstract que extienda RoomDatabase.
3. Anota la clase con @Database. Ignora el error de parámetros faltantes, ya que lo corregirás en el siguiente paso.

import androidx.room.Database  
import androidx.room.RoomDatabase  
  
@Database  
abstract class InventoryDatabase : RoomDatabase() {}

La anotación @Database requiere varios argumentos para que Room pueda compilar la base de datos.

1. Especifica el Item como la única clase con la lista de entities.
2. Establece version como 1**.** Cada vez que cambies el esquema de la tabla de la base de datos, debes aumentar el número de versión.
3. Establece exportSchema como false para que no se conserven las copias de seguridad del historial de versiones de esquemas.

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

1. Dentro del cuerpo de la clase, declara una función abstracta que muestre el ItemDao de modo que la base de datos sepa sobre el DAO.

abstract fun itemDao(): ItemDao

1. Debajo de la función abstracta, define un companion object, que permite el acceso a los métodos para crear u obtener la base de datos y usa el nombre de clase como calificador.

 companion object {}

1. Dentro del objeto companion, declara una variable anulable privada Instance para la base de datos y, luego, inicialízala en null.

La variable Instance conserva una referencia a la base de datos, cuando se crea una. Esto ayuda a mantener una sola instancia de la base de datos abierta en un momento determinado, que es un recurso costoso para crear y mantener.

1. Anota Instance con @Volatile.

El valor de una variable volátil nunca se almacena en caché, y todas las lecturas y escrituras son desde y hacia la memoria principal. Estas funciones ayudan a garantizar que el valor de Instance esté siempre actualizado y sea el mismo para todos los subprocesos de ejecución. Eso significa que los cambios realizados por un subproceso en Instance son visibles de inmediato para todos los demás subprocesos.

@Volatile  
private var Instance: InventoryDatabase? = null

1. Debajo de Instance, mientras estás dentro del objeto companion, define un método getDatabase() con un parámetro Context que necesite el compilador de bases de datos.
2. Muestra un tipo InventoryDatabase. Aparecerá un mensaje de error porque getDatabase() aún no muestra nada.

import android.content.Context  
  
fun getDatabase(context: Context): InventoryDatabase {}

Es posible que varios subprocesos soliciten una instancia de base de datos al mismo tiempo, lo que genera dos bases de datos en lugar de una. Este problema se conoce como [condición de carrera](https://en.wikipedia.org/wiki/Race_condition). Unir el código para obtener la base de datos dentro de un bloque synchronized significa que solo un subproceso de ejecución a la vez puede ingresar este bloque de código, lo que garantiza que la base de datos solo se inicialice una vez. Usa el bloque synchronized{} para evitar la condición de carrera.

1. Dentro de getDatabase(), muestra la variable Instance o, si Instance es nula, inicialízala dentro de un bloque synchronized{}. Para ello, usa el operador elvis (?:).
2. Pasa this, el objeto complementario. Solucionarás el error en los pasos posteriores.

return Instance ?: synchronized(this) { }

1. Dentro del bloque sincronizado, usa el compilador de bases de datos para obtener la base de datos. Ignora los errores, ya que los corregirás en los próximos pasos.

import androidx.room.Room  
  
Room.databaseBuilder()

1. Dentro del bloque synchronized, usa el compilador de bases de datos para obtener una base de datos. Pasa a Room.databaseBuilder() el contexto de la aplicación, la clase de la base de datos y un nombre para la base de datos, item\_database.

Room.databaseBuilder(context, InventoryDatabase::class.java, "item\_database")

Android Studio genera un error de discrepancia de tipos. Para quitar este error, debes agregar una build() en los pasos siguientes.

1. Agrega la estrategia de migración necesaria al compilador. Usa . [fallbackToDestructiveMigration()](https://developer.android.com/reference/androidx/room/RoomDatabase.Builder?hl=es-419" \l "fallbackToDestructiveMigration()" \t "_blank).

.fallbackToDestructiveMigration()

**Nota**: Por lo general, deberías brindar un objeto de migración con una estrategia para cuando cambie el esquema. Un *objeto de migración* es un objeto que define cómo tomas todas las filas con el esquema anterior y las conviertes en filas en el esquema nuevo para que no se pierdan datos. La [migración](https://medium.com/androiddevelopers/understanding-migrations-with-room-f01e04b07929) se encuentra fuera del alcance de este codelab, pero el término se refiere a cuando se cambia el esquema y debes trasladar tu fecha sin perder los datos. Como esta es una app de ejemplo, una alternativa simple es destruir y volver a compilar la base de datos, lo que significa que se pierden los datos de inventario. Por ejemplo, si cambias algo en la clase de entidad, como agregar un parámetro nuevo, puedes permitir que la app borre y vuelva a inicializar la base de datos.

1. Para crear la instancia de base de datos, llama a .build(). Esta llamada quita los errores de Android Studio.

.build()

1. Después de build(), agrega un bloque also y asigna Instance = it para mantener una referencia a la instancia de base de datos recién creada.

.also { Instance = it }

1. Al final del bloque synchronized, muestra instance. El código final se ve como el siguiente:

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 }  
            }  
        }  
    }  
}

 implementarás la interfaz ItemsRepository y la clase OfflineItemsRepository para proporcionar entidades get, insert, delete y update de la base de datos.

1. Abre el archivo ItemsRepository.kt bajo el paquete data.
2. Agrega las siguientes funciones a la interfaz, que se asignan a la implementación de DAO.

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)  
}

1. Abre el archivo OfflineItemsRepository.kt bajo el paquete data.
2. Pasa un parámetro de constructor del tipo ItemDao.

class OfflineItemsRepository(private val itemDao: ItemDao) : ItemsRepository

1. En la clase OfflineItemsRepository, anula las funciones definidas en la interfaz ItemsRepository y llama a las funciones correspondientes desde el 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)  
}

Implementa la clase AppContainer

En esta tarea, crearás una instancia de la base de datos y pasarás la instancia de DAO a la clase OfflineItemsRepository.

1. Abre el archivo AppContainer.kt bajo el paquete data.
2. Pasa la instancia ItemDao() al constructor OfflineItemsRepository.
3. Para crear una instancia de la instancia de base de datos, llama a getDatabase() en la clase InventoryDatabase que pasa el contexto y llama a .itemDao() para crear la instancia de Dao.

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

Explicación de la clase de estado de la IU

Abre el archivo ui/item/ItemEntryViewModel.kt. La clase de datos ItemUiState representa el estado de la IU de un elemento. La clase de datos ItemDetails representa un solo elemento.

El código de partida te proporciona tres funciones de extensión:

* La función de extensión ItemDetails.toItem() convierte el objeto de estado de la IU de ItemUiState en el tipo de entidad Item.
* La función de extensión Item.toItemUiState() convierte el objeto de entidad Item de Room en el tipo de estado de la IU de ItemUiState.
* La función de extensión Item.toItemDetails() convierte el objeto de entidad de Room Item en 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()  
)

Usa la clase anterior en los ViewModels para leer y actualizar la IU.

Actualiza ViewModel de InputEntry

En esta tarea, debes pasar el repositorio al archivo ItemEntryViewModel.kt. También guardas los detalles del elemento ingresados en la pantalla **Add Item** en la base de datos.

1. Observa la función privada validateInput() en la clase ItemEntryViewModel.

// 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()  
    }  
}

La función anterior verifica si name, price y quantity están vacíos. Usarás esta función para verificar la entrada del usuario antes de agregar o actualizar la entidad en la base de datos.

1. Abre la clase ItemEntryViewModel y agrega un parámetro de constructor predeterminado private del tipo ItemsRepository.

import com.example.inventory.data.ItemsRepository  
  
class ItemEntryViewModel(private val itemsRepository: ItemsRepository) : ViewModel() {  
}

1. Actualiza el initializer del ViewModel de entrada de elementos en ui/AppViewModelProvider.kt y pasa la instancia del repositorio como parámetro.

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

1. Ve al archivo ItemEntryViewModel.kt y, al final de la clase ItemEntryViewModel, agrega una función de suspensión llamada saveItem() para insertar un elemento en la base de datos de Room. Esta función agrega los datos a la base de datos sin bloqueos.

suspend fun saveItem() {  
}

1. Dentro de la función, verifica si itemUiState es válido y conviértelo al tipo Item para que Room pueda comprender los datos.
2. Llama a insertItem() en itemsRepository y pasa los datos. La IU llama a esta función para agregar detalles del elemento a la base de datos.

suspend fun saveItem() {  
    if (validateInput()) {  
        itemsRepository.insertItem(itemUiState.itemDetails.toItem())  
    }  
}

Agregaste todas las funciones necesarias para incluir entidades en la base de datos. En la siguiente tarea, actualizarás la IU para usar las funciones anteriores.

Explicación del elemento componible ItemEntryBody()

1. En el archivo ui/item/ItemEntryScreen.kt, la función de componibilidad ItemEntryBody() se implementa de forma parcial como parte del código de partida. Observa la función de componibilidad ItemEntryBody() en la llamada a función ItemEntryScreen().

// 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()  
)

1. Ten en cuenta que el estado de la IU y la lambda updateUiState se pasan como parámetros de función. Mira la definición de la función para ver cómo se actualiza el estado de la IU.

// 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))  
         }  
    }  
}

Estás mostrando ItemInputForm y un botón **Save** en esta función de componibilidad. En el elemento componible ItemInputForm(), se muestran tres campos de texto. El botón **Save** solo está habilitado si se ingresa texto en los campos de texto. El valor *isEntryValid* es "true" si el texto de todos los campos es válido (no está vacío).

|  |  |
| --- | --- |
| Pantalla de teléfono con los detalles del elemento completados parcialmente y el botón Guardar inhabilitado | Pantalla de teléfono con los detalles del elemento completados y el botón Guardar habilitado |

1. Observa la implementación de la función de componibilidad ItemInputForm() y el parámetro de la función onValueChange. Estás actualizando el valor de *itemDetails* con el valor que ingresó el usuario en los campos de texto. Cuando el botón **Save** está habilitado, itemUiState.itemDetails tiene los valores que se deben guardar.

// 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)) },  
            //...  
        )  
    }  
}

Agrega un objeto de escucha de clics al botón Save

Para vincular todo, agrega un controlador de clics al botón **Save**. Dentro del controlador de clics, inicias una corrutina y llamas a saveItem() para guardar los datos en la base de datos de Room.

1. En ItemEntryScreen.kt, dentro de la función de componibilidad ItemEntryScreen, crea un val llamado coroutineScope con la función de componibilidad rememberCoroutineScope().

**Nota:** rememberCoroutineScope() es una función de componibilidad que muestra un CoroutineScope vinculado a la composición a la que se llama. Puedes usar la función de componibilidad rememberCoroutineScope() cuando desees iniciar una corrutina fuera de un elemento componible y asegurarte de que se cancele después de que el alcance salga de la composición. Puedes usar esta función cuando necesitas controlar el ciclo de vida de las corrutinas de forma manual; por ejemplo, para cancelar una animación cada vez que ocurre un evento de usuario.

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

1. Actualiza la llamada a la función *ItemEntryBody*() y, luego, inicia una corrutina dentro de la lambda onSaveClick.

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

1. Observa la implementación de la función saveItem() en el archivo ItemEntryViewModel.kt para verificar si itemUiState es válido a través de la conversión de itemUiState en el tipo Item y, luego, su inserción en la base de datos con itemsRepository.insertItem().

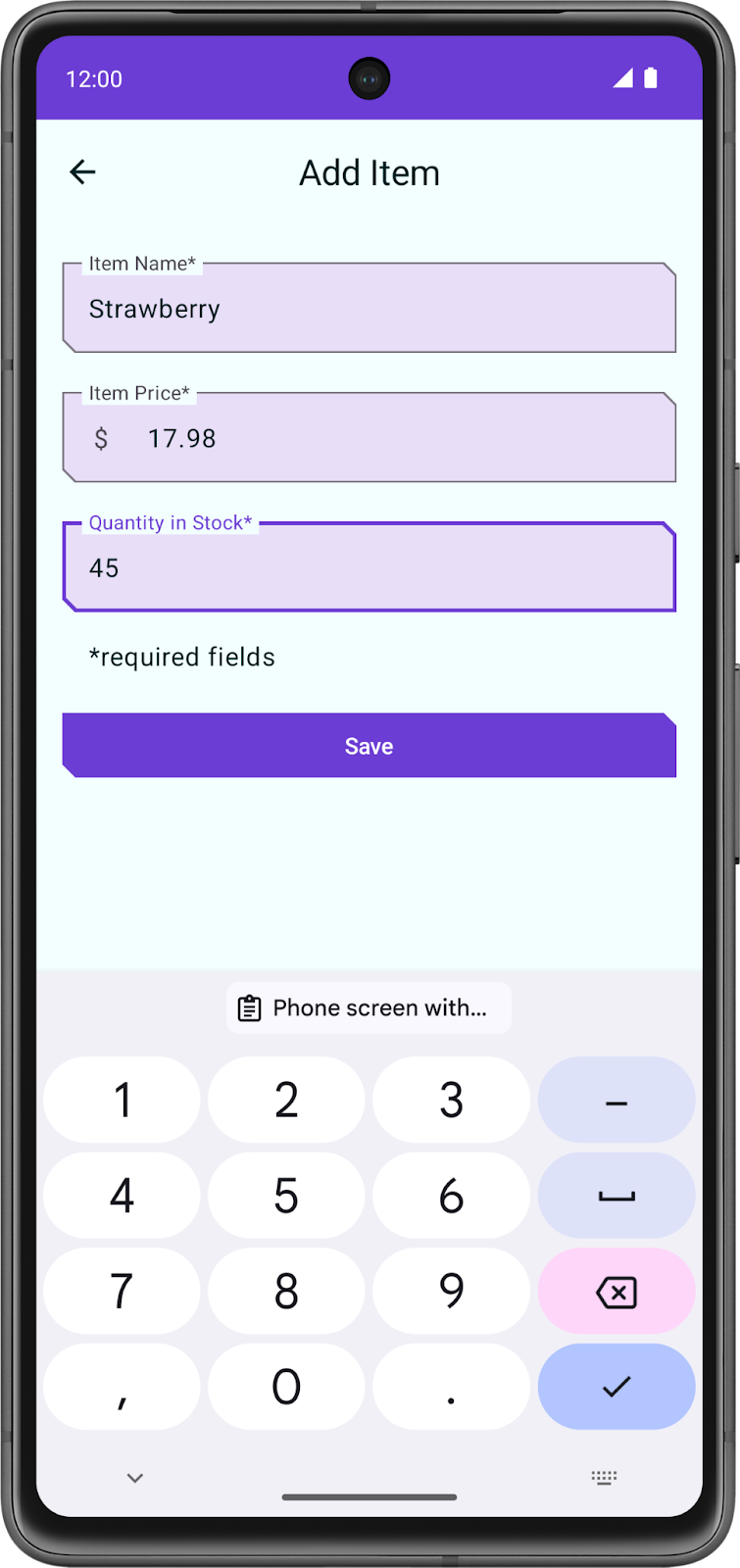
// No need to copy over, you have already implemented this as part of the Room implementation   
  
suspend fun saveItem() {  
    if (validateInput()) {  
        itemsRepository.insertItem(itemUiState.itemDetails.toItem())  
    }  
}

1. En ItemEntryScreen.kt, dentro de la función de componibilidad ItemEntryScreen, dentro de la corrutina, llama a viewModel.saveItem() para guardar el elemento en la base de datos.

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

Observa que no usaste viewModelScope.launch() para saveItem() en el archivo ItemEntryViewModel.kt, pero es necesario para *ItemEntryBody*() cuando llamas a un método de repositorio. Solo puedes llamar a funciones de suspensión desde una corrutina o desde otra función de suspensión. La función viewModel.saveItem() es de suspensión.

1. Compila y ejecuta tu app.
2. Presiona el BAF **+**.
3. En la pantalla **Add Item**, agrega los detalles del elemento y presiona **Save**. Observa que, si presionas el botón **Save**, no se cierra la pantalla **Add Item**.

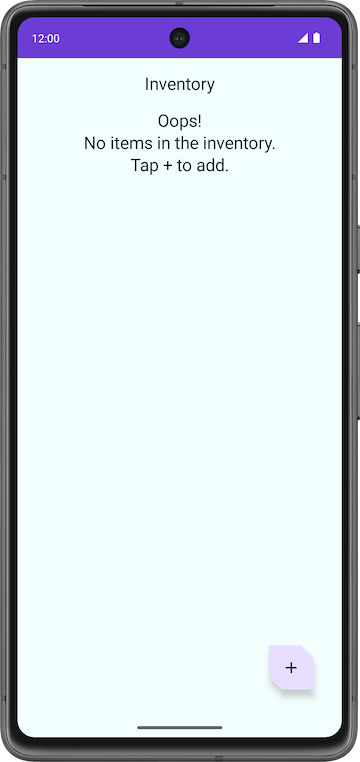


1. En la lambda onSaveClick, agrega una llamada a navigateBack() después de la llamada a viewModel.saveItem() para volver a la pantalla anterior. Tu función ItemEntryBody() se ve como el siguiente código:

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

1. Vuelve a ejecutar la app y realiza los mismos pasos para ingresar y guardar los datos. Observa que, esta vez, la app regresa a la pantalla **Inventory**.

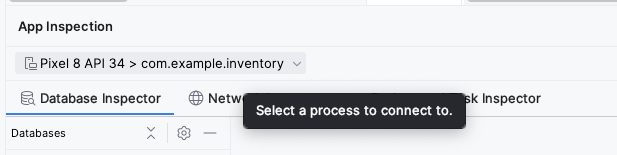
Esta acción guarda los datos, pero no puede ver los datos de inventario en la app. En la próxima tarea, usarás el [Inspector de bases de datos](https://developer.android.com/studio/inspect/database?hl=es-419) para ver los datos que guardaste.



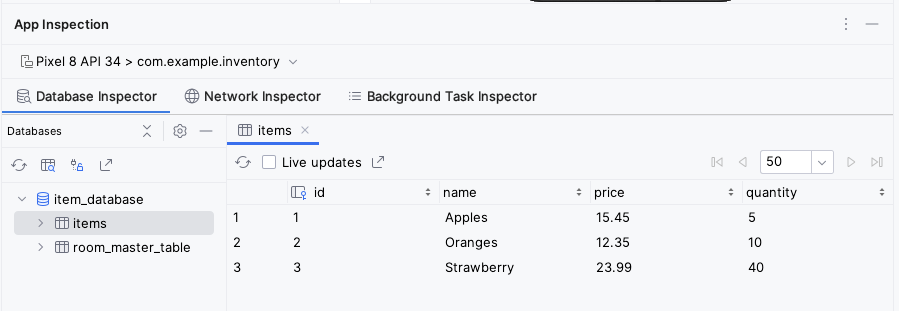
El [Inspector de bases de datos](https://developer.android.com/studio/inspect/database?hl=es-419) te permite inspeccionar, consultar y modificar las bases de datos de tu app mientras se ejecuta. Esto es particularmente útil para depurar bases de datos. El Inspector de bases de datos funciona con SQLite simple y bibliotecas compiladas sobre SQLite, como Room. El Inspector de bases de datos funciona mejor en emuladores o dispositivos que ejecutan el nivel de API 26.

**Nota**: El Inspector de bases de datos solo funciona con la biblioteca de SQLite que se incluye en el sistema operativo Android en el nivel de API 26 y superiores. No funciona con las otras bibliotecas de SQLite que empaquetas con tu app.

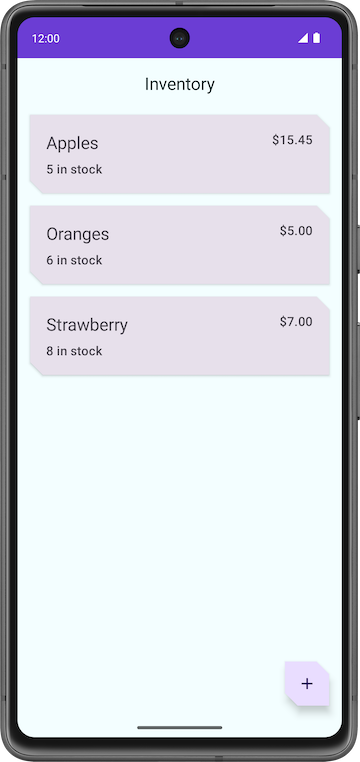
1. Si aún no lo hiciste, ejecuta tu app en un emulador o un dispositivo conectado con nivel de API 26 o superior.
2. En Android Studio, selecciona **View** > **Tool Windows** > **App Inspection** en la barra de menú.
3. Elige la pestaña **Database Inspector**.
4. En el panel **Database Inspector**, selecciona com.example.inventory en el menú desplegable si aún no está seleccionado. **item\_database** de la app de Inventory aparece en el panel **Databases**.



1. Expande el nodo de **item\_database** del panel **Databases** y selecciona **Item** para inspeccionarlo. Si el panel **Databases** está vacío, usa el emulador para agregar algunos elementos a la base de datos con la pantalla **Add Item**.
2. Marca la casilla de verificación **Live updates** en el Inspector de bases de datos para actualizar automáticamente los datos que presenta mientras interactúas con tu app en ejecución en el emulador o dispositivo.



1. Como leer y actualizar datos Room



Explicación de la función de componibilidad de HomeScreen

* Abre el archivo ui/home/HomeScreen.kt y observa el elemento componible HomeScreen().

@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()  
        )  
    }

Esta función de componibilidad muestra los siguientes elementos:

* La barra superior de la aplicación con el título
* El botón de acción flotante (BAF) para agregar nuevos elementos al inventario 
* La función de componibilidad HomeBody()

La función de componibilidad HomeBody() muestra elementos de inventario en función de la lista que se pasó. Como parte de la implementación del código de partida, se pasa una lista vacía (listOf()) a la función de componibilidad HomeBody(). Para pasar la lista de inventario a este elemento componible, debes recuperar los datos del inventario del repositorio y pasarlos a HomeViewModel.

Emite un estado de IU en HomeViewModel

Cuando agregaste métodos a ItemDao para obtener elementos (getItem() y getAllItems()), especificaste un Flow como el tipo de datos que se muestra. Recuerda que un Flow representa una transmisión de datos genérica. Si se devuelve un Flow, solo debes llamar explícitamente a los métodos desde el DAO una vez para un ciclo de vida determinado. Room controla las actualizaciones de los datos subyacentes de manera asíncrona.

Obtener datos de un flujo se denomina *recopilación de un flujo*. Cuando recopiles datos desde un flujo en la capa de la IU, debes tener en cuenta algunos aspectos.

* Los eventos de ciclo de vida, como los cambios de configuración, por ejemplo, la rotación del dispositivo, hacen que se vuelva a crear la actividad. Esto provoca la recomposición y la recopilación de tu Flow nuevamente.
* Querrás que los valores se almacenen en caché como estados para que los datos existentes no se pierdan entre los eventos de ciclo de vida.
* Los flujos deben cancelarse si no quedan observadores, como después de que finalice el ciclo de vida de un elemento componible.

La forma recomendada de exponer un Flow desde un ViewModel es con un StateFlow. El uso de un StateFlow permite guardar y observar los datos, sin importar el ciclo de vida de la IU. Para convertir un Flow en un StateFlow, usa el operador stateIn.

El operador stateIn tiene tres parámetros que se explican a continuación:

* scope: El viewModelScope define el ciclo de vida del StateFlow. Cuando se cancela el viewModelScope, también se cancela el StateFlow.
* started: La canalización solo debe estar activa cuando la IU sea visible. Se usa SharingStarted.WhileSubscribed() para lograr esto. Para configurar una demora (en milisegundos) entre la desaparición del último suscriptor y la detención de la corrutina de uso compartido, pasa TIMEOUT\_MILLIS al método SharingStarted.WhileSubscribed().
* initialValue: Establece el valor inicial del flujo de estado en HomeUiState().

Una vez que conviertas tu Flow en StateFlow, podrás recopilarlo con el método collectAsState() y convertir sus datos en State del mismo tipo.

En este paso, recuperarás todos los elementos de la base de datos de Room como una API observable de StateFlow para el estado de la IU. Cuando cambian los datos de Inventory de Room, la IU se actualiza automáticamente.

1. Abre el archivo ui/home/HomeViewModel.kt, que contiene una constante TIMEOUT\_MILLIS y una clase de datos HomeUiState con una lista de elementos como parámetro de constructor.

// 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())

1. Dentro de la clase HomeViewModel, declara un val llamado homeUiState del tipo StateFlow<HomeUiState>. En breve, resolverás el error de inicialización.

val homeUiState: StateFlow<HomeUiState>

1. Llama a getAllItemsStream() en itemsRepository y asígnalo al homeUiState que acabas de declarar.

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

Ahora se muestra el error que indica que la referencia itemsRepository no está resuelta. Para solucionar este error, debes pasar el objeto ItemsRepository a HomeViewModel.

1. Agrega un parámetro de constructor del tipo ItemsRepository a la clase HomeViewModel.

import com.example.inventory.data.ItemsRepository  
  
class HomeViewModel(itemsRepository: ItemsRepository): ViewModel() {

1. En el archivo ui/AppViewModelProvider.kt, en el inicializador HomeViewModel, pasa el objeto ItemsRepository como se muestra.

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

1. Regresa al archivo HomeViewModel.kt. Observa el error de discrepancia de tipos. Para resolver esto, agrega un mapa de transformación como se muestra a continuación.

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

Android Studio aún te muestra un error de discrepancia de tipos. Este error se debe a que homeUiState es del tipo StateFlow y getAllItemsStream() devuelve un Flow.

1. Usa el operador stateIn para convertir Flow en StateFlow. StateFlow es la API observable para el estado de la IU, que permite que la IU se actualice sola.

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()  
        )

1. Compila la app para asegurarte de que no haya errores en el código. No habrá cambios visuales.
2. En el archivo HomeScreen.kt, en la función de componibilidad HomeScreen, agrega un parámetro de función nuevo del tipo HomeViewModel y, luego, inicialízalo.

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)  
)

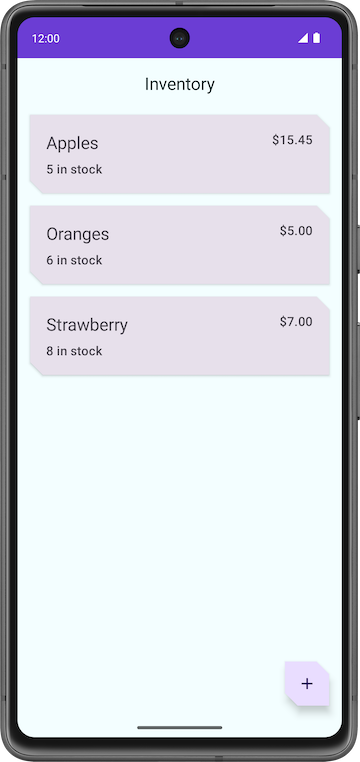
1. En la función de componibilidad HomeScreen, agrega un val llamado homeUiState para recopilar el estado de la IU de HomeViewModel. Usa *collectAsState*(), que recopila valores de este [StateFlow](https://kotlin.github.io/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/-state-flow/index.html" \t "_blank) y representa su valor más reciente mediante [State](https://developer.android.com/reference/kotlin/androidx/compose/runtime/State?hl=es-419" \t "_blank).

import androidx.compose.runtime.collectAsState  
import androidx.compose.runtime.getValue  
  
val homeUiState by viewModel.homeUiState.collectAsState()

1. Actualiza la llamada a la función HomeBody() y pasa homeUiState.itemList al parámetro itemList.

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

1. Ejecuta la app. Observa que la lista de inventario muestra si guardaste elementos en la base de datos de tu app. Si la lista está vacía, agrega algunos elementos de inventario a la base de datos de la app.



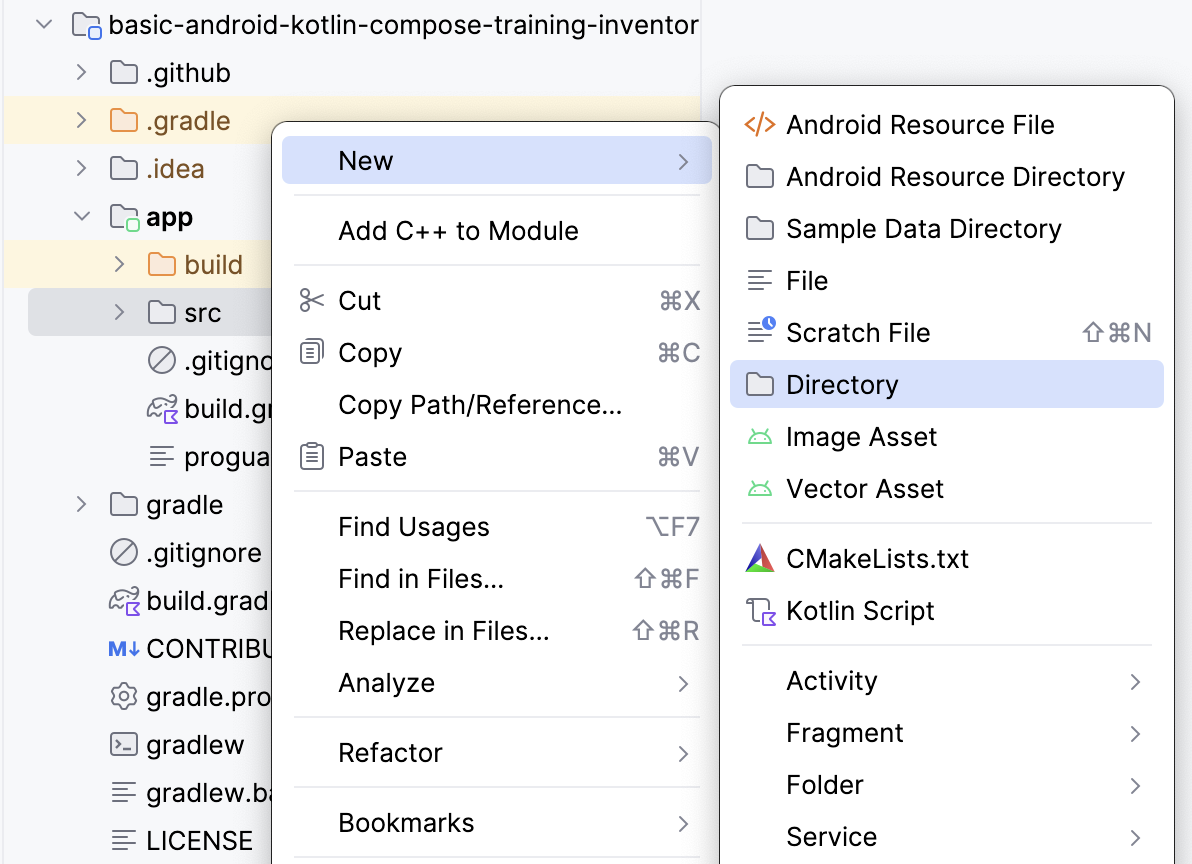
En los codelabs anteriores, se analiza la importancia de probar el código. En esta tarea, agregarás algunas pruebas de unidades para probar tus consultas DAO y, luego, agregarás más pruebas a medida que avances en el codelab.

El enfoque recomendado para probar la implementación de la base de datos es escribir una prueba JUnit que se ejecute en un dispositivo Android. Como estas pruebas no requieren la creación de una actividad, se ejecutan más rápido que tus pruebas de IU.

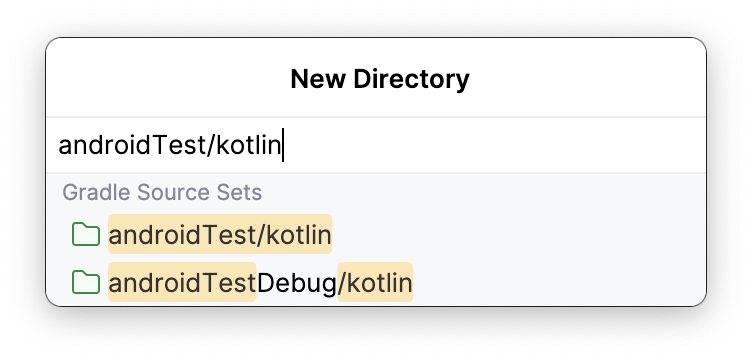
1. En el archivo build.gradle.kts (Module :app), observa las siguientes dependencias para Espresso y JUnit.

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

1. Cambia a la vista **Project** y haz clic con el botón derecho en **src** > **New** > **Directory** para crear un conjunto de orígenes de prueba para tu pruebas.



1. Selecciona **androidTest/kotlin** en la ventana emergente **New Directory**.



1. Crea una clase de Kotlin llamada ItemDaoTest.kt.
2. Anota la clase ItemDaoTest con @RunWith(AndroidJUnit4::class). Ahora tu clase se verá como el siguiente código de ejemplo:

package com.example.inventory  
  
import androidx.test.ext.junit.runners.AndroidJUnit4  
import org.junit.runner.RunWith  
  
@RunWith(AndroidJUnit4::class)  
class ItemDaoTest {  
}

1. Dentro de la clase, agrega variables privadas var de tipo ItemDao y InventoryDatabase.

import com.example.inventory.data.InventoryDatabase  
import com.example.inventory.data.ItemDao  
  
private lateinit var itemDao: ItemDao  
private lateinit var inventoryDatabase: InventoryDatabase

1. Agrega una función para crear la base de datos y anótala con @Before para que pueda ejecutarse antes de cada prueba.
2. Dentro del método, inicializa 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()  
}

En esta función, se usa una base de datos en la memoria y no se conserva en el disco. Para hacerlo, usa la función inMemoryDatabaseBuilder(). Debes hacerlo porque la información no debe ser persistente, sino que debe borrarse cuando se cierra el proceso. Estás ejecutando las consultas DAO en el subproceso principal con .allowMainThreadQueries() solo para pruebas.

1. Agrega otra función para cerrar la base de datos. Anótala con @After para cerrar la base de datos y ejecutarla después de cada prueba.

import org.junit.After  
import java.io.IOException  
  
@After  
@Throws(IOException::class)  
fun closeDb() {  
    inventoryDatabase.close()  
}

1. Declara elementos en la clase ItemDaoTest para que los use la base de datos, como se muestra en el siguiente código de ejemplo:

import com.example.inventory.data.Item  
  
private var item1 = Item(1, "Apples", 10.0, 20)  
private var item2 = Item(2, "Bananas", 15.0, 97)

1. Agrega funciones de utilidad para agregar un elemento y, luego, dos elementos a la base de datos. Más adelante, usarás estas funciones en la prueba. Márcalas como suspend para que puedan ejecutarse en una corrutina.

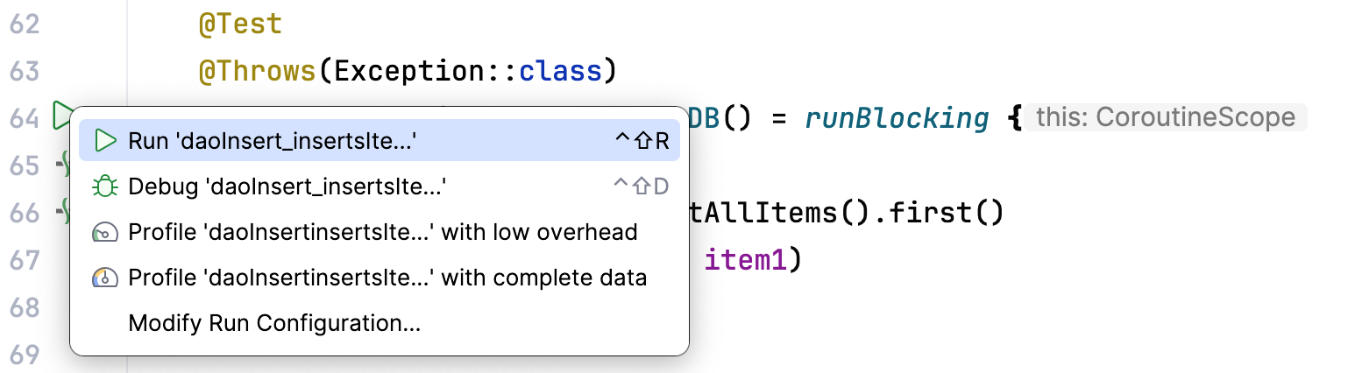
private suspend fun addOneItemToDb() {  
    itemDao.insert(item1)  
}  
  
private suspend fun addTwoItemsToDb() {  
    itemDao.insert(item1)  
    itemDao.insert(item2)  
}

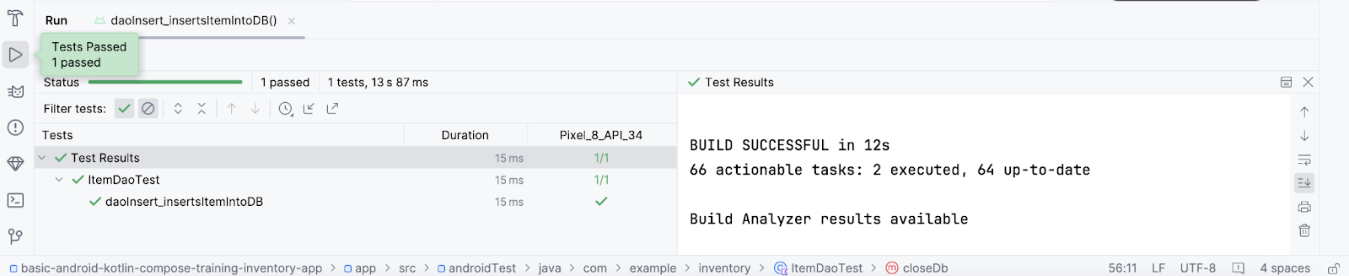
1. Escribe una prueba para insertar un solo elemento en la base de datos, insert(). Asigna el nombre daoInsert\_insertsItemIntoDB a la prueba y anótala con @Test.

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)  
}

En esta prueba, usarás la función de utilidad addOneItemToDb() para agregar un elemento a la base de datos. Luego, debes leer el primer elemento de la base de datos. Con assertEquals(), puedes comparar el valor esperado con el real. Ejecuta la prueba en una corrutina nueva con runBlocking{}. Esta configuración es la razón por la que marcas las funciones de utilidad como suspend.

1. Ejecuta la prueba y asegúrate de que sea exitosa.





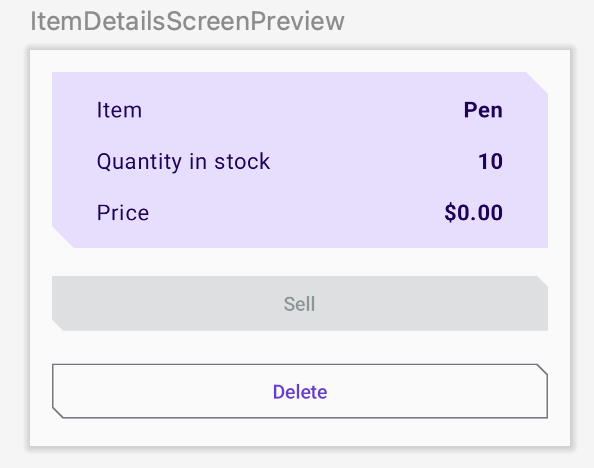
1. Escribe otra prueba para getAllItems() desde la base de datos. Asigna el nombre daoGetAllItems\_returnsAllItemsFromDB a la prueba.

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

La función de componibilidad ItemDetailsScreen ya está escrita y contiene tres elementos componibles de texto que muestran los detalles del elemento.

ui/item/ItemDetailsScreen.kt

Esta pantalla forma parte del código de partida y muestra los detalles de los elementos, que verás en un codelab posterior. En este codelab, no trabajarás en esa pantalla. ItemDetailsViewModel.kt es el ViewModel correspondiente de esta pantalla.



1. En la función de componibilidad HomeScreen, observa la llamada a la función HomeBody(). navigateToItemUpdate se pasa al parámetro onItemClick, al que se llama cuando haces clic en cualquier elemento de la lista.

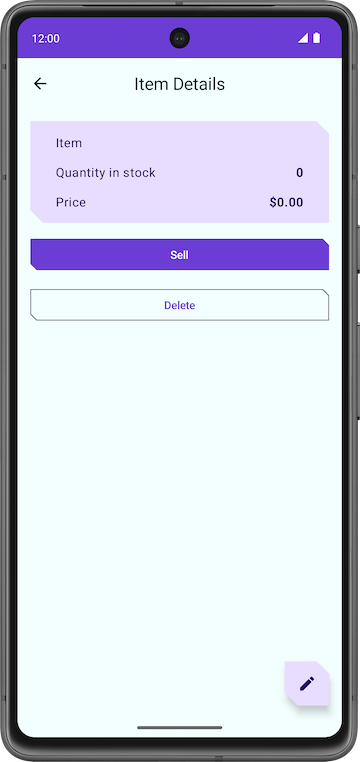
// No need to copy over   
HomeBody(  
    itemList = homeUiState.itemList,  
    onItemClick = navigateToItemUpdate,  
    modifier = modifier  
        .padding(innerPadding)  
        .fillMaxSize()  
)

1. Abre ui/navigation/InventoryNavGraph.kt y observa el parámetro navigateToItemUpdate en el elemento componible HomeScreen. Este parámetro especifica el destino de la navegación como la pantalla de detalles del elemento.

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

Ya se implementó esta parte de la funcionalidad de onItemClick. Cuando haces clic en el elemento de la lista, la app navega a la pantalla de detalles del elemento.

1. Haz clic en cualquier elemento de la lista del inventario para ver la pantalla de detalles del elemento con campos vacíos.



Para completar los campos de texto con los detalles del elemento, debes recopilar el estado de la IU en ItemDetailsScreen().

1. En UI/Item/ItemDetailsScreen.kt, agrega un parámetro nuevo a la función de componibilidad ItemDetailsScreen del tipo ItemDetailsViewModel y usa el método de fábrica para inicializarlo.

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)  
)

1. Dentro de la función de componibilidad ItemDetailsScreen(), crea un val llamado uiState para recopilar el estado de la IU. Usa collectAsState() para recopilar uiState StateFlow y representar su valor más reciente mediante State. Android Studio muestra un error de referencia sin resolver.

import androidx.compose.runtime.collectAsState  
  
val uiState = viewModel.uiState.collectAsState()

1. Para resolver el error, crea un val llamado uiState del tipo StateFlow<ItemDetailsUiState> en la clase ItemDetailsViewModel.
2. Recupera los datos del repositorio de elementos y asígnalos a ItemDetailsUiState con la función de extensión toItemDetails(). La función de extensión Item.toItemDetails() ya está escrita como parte del código de partida.

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()  
             )

1. Pasa ItemsRepository a ItemDetailsViewModel para resolver el error Unresolved reference: itemsRepository.

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

1. En ui/AppViewModelProvider.kt, actualiza el inicializador para ItemDetailsViewModel como se muestra en el siguiente fragmento de código:

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

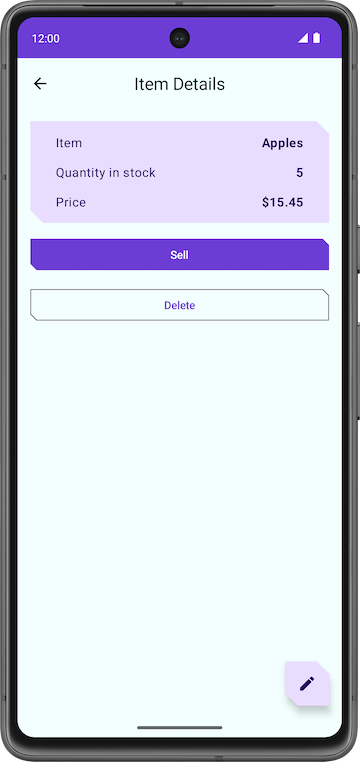
1. Vuelve a ItemDetailsScreen.kt y observa que se resolvió el error en el elemento componible ItemDetailsScreen().
2. En el elemento componible ItemDetailsScreen(), actualiza la llamada a función ItemDetailsBody() y pasa uiState.value al argumento itemUiState.

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

1. Observa las implementaciones de ItemDetailsBody() y ItemInputForm(). Estás pasando el elemento item seleccionado actual de ItemDetailsBody() a 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()  
         )  
  
      //...  
    }

1. Ejecuta la app. Cuando hagas clic en cualquier elemento de la lista en la pantalla **Inventory**, se mostrará la pantalla **Item Details**.
2. Observa que la pantalla ya no está en blanco. Muestra los detalles de la entidad recuperados de la base de datos de inventario.

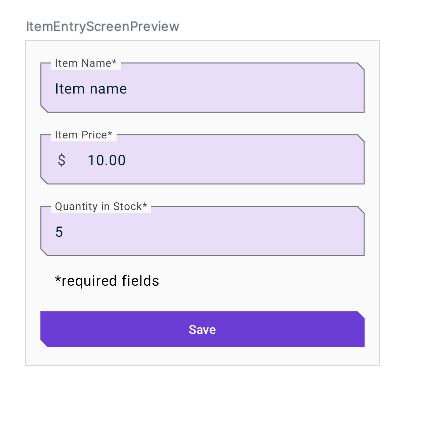


1. Presiona el botón **Sell**. No pasa nada.

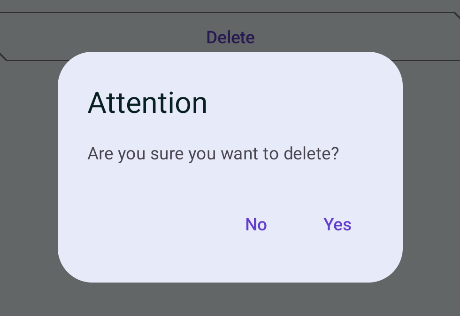
### **ui/item/ItemEditScreen.kt**

La pantalla Edit Item ya se te proporcionó como parte del código de partida.

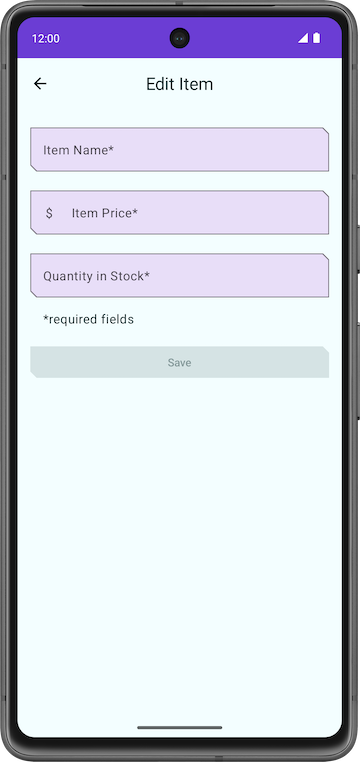
Este diseño contiene elementos componibles de campo de texto para editar los detalles de cualquier elemento de inventario nuevo.



El código de esta app todavía no funciona por completo. Por ejemplo, en la pantalla **Item Details**, cuando presionas el botón **Sell**, la **Quantity in Stock** no disminuye. Cuando presionas el botón **Delete**, la app te muestra un cuadro de diálogo de confirmación. Sin embargo, si seleccionas el botón **Yes**, la app no borrará el elemento.



Por último, el botón BAF  abre una pantalla **Edit Item** vacía.



En esta sección, extenderás las funciones de la app para implementar la funcionalidad de venta. Esta actualización incluye las siguientes tareas:

* Agregar una prueba para que la función DAO actualice una entidad
* Agregar una función a ItemDetailsViewModel para reducir la cantidad y actualizar la entidad en la base de datos de la app
* Inhabilitar el botón **Sell** si la cantidad es cero

1. En ItemDaoTest.kt, agrega una función llamada daoUpdateItems\_updatesItemsInDB() sin parámetros. Anota con @Test y @Throws(Exception::class).

@Test  
@Throws(Exception::class)  
fun daoUpdateItems\_updatesItemsInDB()

1. Define la función y crea un bloque *runBlocking*. Llama a addTwoItemsToDb() dentro de ella.

fun daoUpdateItems\_updatesItemsInDB() = runBlocking {  
    addTwoItemsToDb()  
}

1. Actualiza las dos entidades con diferentes valores y llama a itemDao.update.

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

1. Recupera las entidades con itemDao.getAllItems(). Compáralas con la entidad actualizada y afirma.

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

1. Asegúrate de que la función completada se vea de la siguiente manera:

@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))  
}

1. Ejecuta la prueba y asegúrate de que sea exitosa.

Agrega una función en ViewModel.

1. En ItemDetailsViewModel.kt, dentro de la clase ItemDetailsViewModel, agrega una función llamada reduceQuantityByOne() sin parámetros.

fun reduceQuantityByOne() {  
}

1. Dentro de la función, inicia una corrutina con viewModelScope.launch{}.

**Nota:** Debes ejecutar operaciones de base de datos dentro de una corrutina.

import kotlinx.coroutines.launch  
import androidx.lifecycle.viewModelScope  
  
  
viewModelScope.launch {  
}

1. Dentro del bloque launch, crea un val llamado currentItem y establécelo en *uiState.value.toItem().*

val currentItem = uiState.value.toItem()

El uiState.value es del tipo ItemUiState. Lo conviertes al tipo de entidad Item con la función de extensión *toItem*().

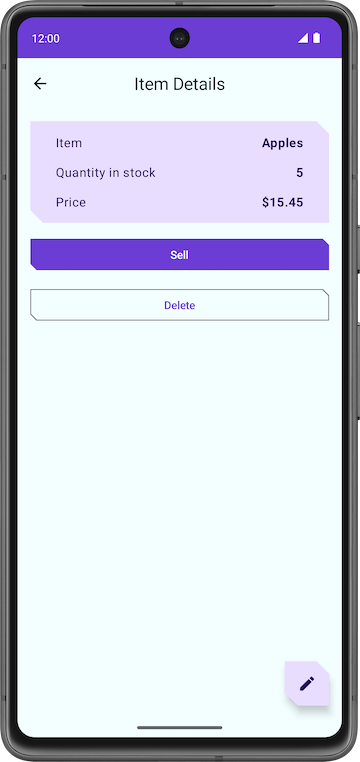
1. Agrega una sentencia if para verificar si quality es mayor que 0.
2. Llama a updateItem() en itemsRepository y pasa el currentItem actualizado. Usa copy() para actualizar el valor quantity y que la función se vea de la siguiente manera:

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

1. Vuelve a ItemDetailsScreen.kt.
2. En el elemento componible ItemDetailsScreen, ve a la llamada a la función ItemDetailsBody().
3. En la lambda onSellItem, llama a viewModel.reduceQuantityByOne().

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

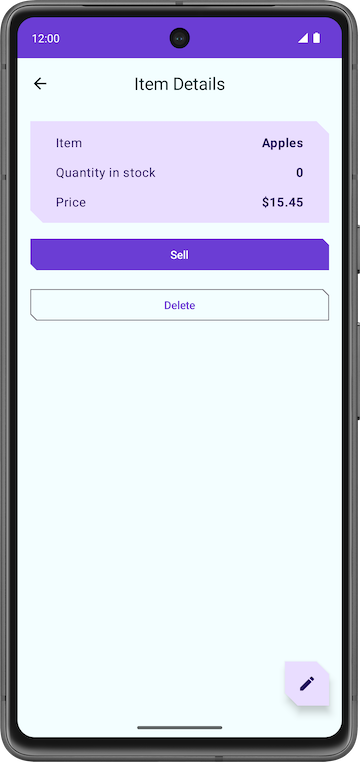
1. Ejecuta la app.
2. En la pantalla **Inventory**, haz clic en un elemento de la lista. Cuando aparezca la pantalla **Item Details**, presiona **Sell** y observa que el valor de la cantidad disminuye en uno.



1. En la pantalla **Item Details**, presiona continuamente el botón **Sell** hasta que la cantidad sea cero.

**Sugerencia:** Si quieres ahorrar tiempo, te recomendamos usar un elemento con una cantidad baja para esta tarea. Si ninguno de los elementos tiene una cantidad baja, puedes crear uno nuevo con una cantidad baja.

Cuando la cantidad llegue a cero, vuelve a presionar **Sell**. No hay cambios visuales porque la función reduceQuantityByOne() comprueba si la cantidad es mayor que cero antes de actualizarla.

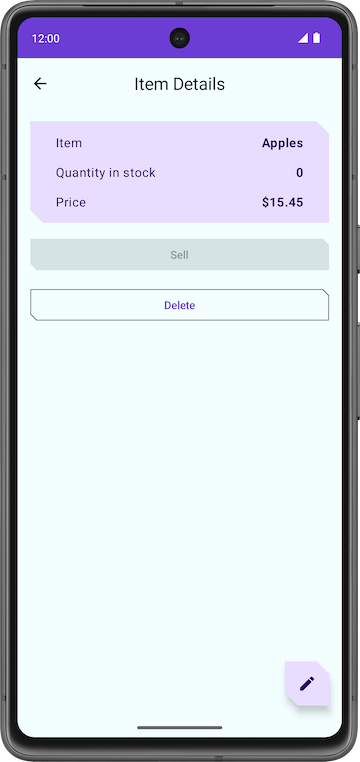


Para proporcionar mejores comentarios a los usuarios, puedes inhabilitar el botón **Sell** cuando no haya ningún artículo para vender.

1. En la clase ItemDetailsViewModel, establece el valor outOfStock según it.quantity en la transformación de map.

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

1. Ejecuta la app. Observa que la app inhabilita el botón **Sell** cuando la cantidad en stock es cero.



Felicitaciones por implementar la función **Sell** de elementos en tu app.

Entidad Delete item

Al igual que con la tarea anterior, debes extender las funciones de tu app con la implementación de la función de borrar. Esta función es mucho más fácil de implementar que la función de venta. El proceso incluye las siguientes tareas:

* Agregar una prueba para la consulta de eliminación de DAO
* Agregar una función en la clase ItemDetailsViewModel para borrar una entidad de la base de datos
* Actualizar el elemento *ItemDetailsBody* componible

Agrega una prueba de DAO

1. En ItemDaoTest.kt, agrega una prueba llamada daoDeleteItems\_deletesAllItemsFromDB().

@Test  
@Throws(Exception::class)  
fun daoDeleteItems\_deletesAllItemsFromDB()

1. Inicia una corrutina con *runBlocking {}*.

fun daoDeleteItems\_deletesAllItemsFromDB() = runBlocking {  
}

1. Agrega dos elementos a la base de datos y llama a itemDao.delete() en esos dos elementos para borrarlos de la base de datos.

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

1. Recupera las entidades de la base de datos y verifica que la lista esté vacía. La prueba completada debería verse de la siguiente manera:

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())  
}

Agrega la función de borrar en ItemDetailsViewModel

1. En ItemDetailsViewModel, agrega una nueva función llamada deleteItem() que no tome parámetros ni devuelva nada.
2. Dentro de la función deleteItem(), agrega una llamada a la función itemsRepository.deleteItem() y pasa uiState.value.*toItem*().

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

En esta función, conviertes uiState del tipo itemDetails al tipo de entidad Item con la función de extensión *toItem*().

1. En el elemento componible ui/item/ItemDetailsScreen, agrega un elemento val llamado coroutineScope y establécelo en rememberCoroutineScope(). Este enfoque muestra un alcance de corrutinas vinculado a la composición a la que se llama (elemento componible ItemDetailsScreen).

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

1. Desplázate hasta la función ItemDetailsBody().
2. Inicia una corrutina con coroutineScope dentro de la lambda onDelete.
3. Dentro del bloque launch, llama al método deleteItem() en viewModel.

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

1. Después de borrar el elemento, vuelve a la pantalla del inventario.
2. Llama a navigateBack() después de la llamada a función deleteItem().

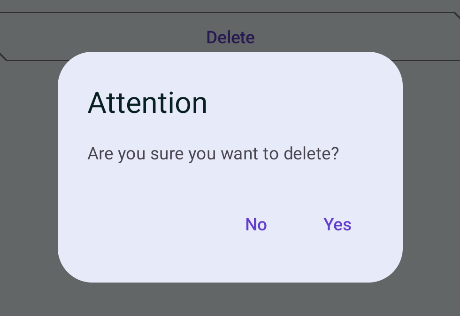
onDelete = {  
    coroutineScope.launch {  
        viewModel.deleteItem()  
        navigateBack()  
    }

1. Dentro del archivo ItemDetailsScreen.kt, desplázate hasta la función ItemDetailsBody().

Esta función es parte del código de partida. Este elemento componible muestra un diálogo de alerta para obtener la confirmación del usuario antes de borrar el elemento y llama a la función deleteItem() cuando presionas **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()  
                },  
                //...  
            )  
        }  
    }  
}

Cuando presionas **No**, la app cierra el diálogo de alerta. La función showConfirmationDialog() muestra la siguiente alerta:



1. Ejecuta la app.
2. Selecciona un elemento de lista en la pantalla **Inventory**.
3. En la pantalla **Item Details**, presiona **Delete**.
4. Presiona **Yes** en el diálogo de alerta y la app regresará a la pantalla **Inventory**.
5. Confirma que la entidad que borraste ya no esté en la base de datos de la app.

Felicitaciones por implementar la función de borrar.

|  |  |
| --- | --- |
| Pantalla Item Details con la ventana de diálogo de una alerta. | Pantalla de teléfono en la que se muestra la lista de inventario sin el elemento borrado |

Edita la entidad del elemento

Al igual que en las secciones anteriores, en esta sección, agregarás otra mejora de funciones a la app que edita la entidad de un elemento.

A continuación, te mostramos los pasos rápidos para editar una entidad en la base de datos de la app:

* Agrega una prueba a la consulta de DAO de obtención de elementos de prueba.
* Propaga los campos de texto y la pantalla **Edit Item** con los detalles de la entidad.
* Actualiza la entidad en la base de datos con Room.

Agrega una prueba de DAO

1. En ItemDaoTest.kt, agrega una prueba llamada daoGetItem\_returnsItemFromDB().

@Test  
@Throws(Exception::class)  
fun daoGetItem\_returnsItemFromDB()

1. Define la función. Dentro de la corrutina, agrega un elemento a la base de datos.

@Test  
@Throws(Exception::class)  
fun daoGetItem\_returnsItemFromDB() = runBlocking {  
    addOneItemToDb()  
}

1. Recupera la entidad de la base de datos con la función itemDao.getItem() y establécela en un val llamado item.

val item = itemDao.getItem(1)

1. Compara el valor real con el valor recuperado y afirma con assertEquals(). La prueba completada se verá de la siguiente manera:

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

1. Ejecuta la prueba y asegúrate de que sea exitosa.

Propaga campos de texto

Si ejecutas la app, ve a la pantalla **Item Details** y haz clic en el BAF. Verás que el título de la pantalla ahora es **Edit Item**. Sin embargo, todos los campos de texto están vacíos. En este paso, completarás los campos de texto en la pantalla **Edit Item** con los detalles de la entidad.

|  |  |
| --- | --- |
| La pantalla Item Details disminuye la cantidad en uno cuando se presiona el botón Sell | Pantalla Edit Item con campos vacíos |

1. En ItemDetailsScreen.kt, desplázate hasta el elemento componible ItemDetailsScreen.
2. En FloatingActionButton(), cambia el argumento onClick para incluir uiState.value.itemDetails.id, que es el id de la entidad seleccionada. Debes usar este id para recuperar los detalles de la entidad.

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

1. En la clase ItemEditViewModel, agrega un bloque init.

init {  
  
}

1. Dentro del bloque init, inicia una corrutina con *viewModelScope*.*launch*.

import kotlinx.coroutines.launch  
  
viewModelScope.launch { }

1. Dentro del bloque launch, recupera los detalles de la entidad con 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)  
    }  
}

En este bloque de lanzamiento, se agrega un filtro para devolver un flujo que solo contenga valores que no son nulos. Con toItemUiState(), conviertes la entidad item en ItemUiState. Pasas el valor actionEnabled como true para habilitar el botón **Save**.

Para resolver el error Unresolved reference: itemsRepository, debes pasar *ItemsRepository* como una dependencia al modelo de vista.

1. Agrega un parámetro de constructor a la clase ItemEditViewModel.

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

1. En el archivo AppViewModelProvider.kt, en el inicializador ItemEditViewModel, agrega el objeto ItemsRepository como argumento.

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

1. Ejecuta la app.
2. Ve a **Item Details** y presiona el BAF .
3. Observa que los campos se propagan con los detalles del elemento.
4. Edita la cantidad en stock o cualquier otro campo, y presiona el botón **Save**.

No pasa nada. Eso se debe a que no estás actualizando la entidad en la base de datos de la app. Solucionarás este problema en la próxima sección.

|  |  |
| --- | --- |
| La pantalla Item Details disminuye la cantidad en uno cuando se presiona el botón Sell | Pantalla Edit Item con campos vacíos |

Actualiza la entidad con Room

En esta tarea final, agregarás las partes finales del código para implementar la funcionalidad de actualización. Definirás las funciones necesarias en el ViewModel y las usarás en el ItemEditScreen.

Es hora de volver a programar.

1. En la clase ItemEditViewModel, agrega una función llamada updateUiState() que tome un objeto ItemUiState y no devuelva nada. Esta función actualiza itemUiState con los valores nuevos que ingresa el usuario.

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

En esta función, asignarás el itemDetails pasado a itemUiState y actualizarás el valor isEntryValid. La app habilita el botón **Save** si itemDetails es true. Establece este valor en true *solo* si la entrada que ingresa el usuario es válida.

1. Ve al archivo ItemEditScreen.kt.
2. En el elemento ItemEditScreen componible, desplázate hacia abajo hasta la llamada a función ItemEntryBody().
3. Establece el valor del argumento onItemValueChange en la nueva función updateUiState.

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

1. Ejecuta la app.
2. Ve a la pantalla **Edit Item**.
3. Deja uno de los valores de la entidad en blanco para que no sea válido. Observa cómo el botón **Save** se inhabilita automáticamente.

|  |  |  |
| --- | --- | --- |
| Pantalla Item Details con el botón Sell habilitado | Pantalla Edit Item con todos los campos de texto completos y el botón Save habilitado | Pantalla Edit Item con el botón Save inhabilitado |

1. Vuelve a la clase ItemEditViewModel y agrega una función suspend llamada updateItem() que no tome nada. Usarás esta función para guardar la entidad actualizada en la base de datos de Room.

suspend fun updateItem() {  
}

1. Dentro de la función getUpdatedItemEntry(), agrega una condición if para validar la entrada del usuario usando la función validateInput().
2. Realiza una llamada a la función updateItem() en itemsRepository y pasa itemUiState.itemDetails.*toItem*(). Las entidades que se pueden agregar a la base de datos de Room deben ser del tipo Item. La función completada se ve de la siguiente manera:

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

1. Vuelve al elemento ItemEditScreen componible. Necesitas un alcance de corrutinas para llamar a la función updateItem(). Crea un val llamado coroutineScope y establécelo en rememberCoroutineScope().

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

1. En la llamada a función *ItemEntryBody()*, actualiza el argumento de función onSaveClick para iniciar una corrutina en coroutineScope.
2. Dentro del bloque launch, llama a updateItem() en viewModel y navega hacia atrás.

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

La llamada a la función *ItemEntryBody()* completada se ve de la siguiente manera:

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

1. Ejecuta la app y edita los elementos del inventario. Ahora puedes editar cualquier elemento en la base de datos de la app de Inventory.

|  |  |
| --- | --- |
| Pantalla Edit Item con los detalles del elemento editados | Pantalla Item Details con detalles del elemento actualizados |