Evidencia dia 02 - Semana 16

Leonardo Rodenas Escobar

Reflexión:

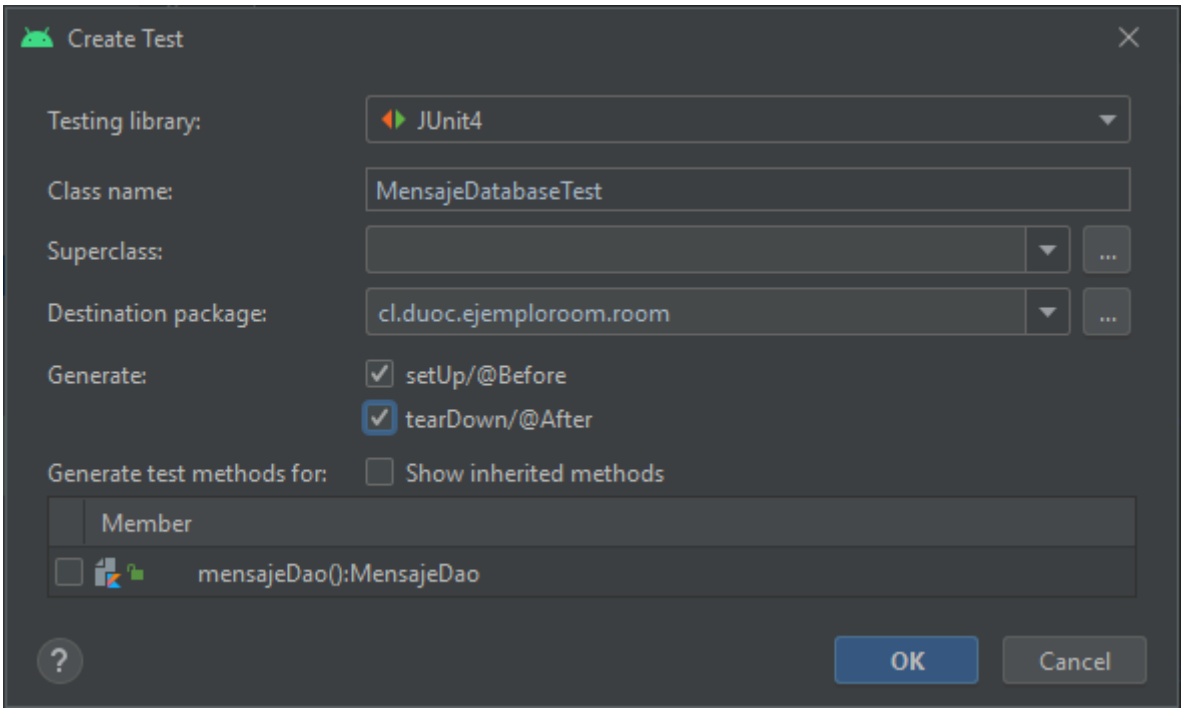
La clase fue mitad pruebas unitarias en Kotlin, probando los métodos de Room (los Dao) y seguir con el ejercicio anterior. Por mi parte estoy atascadísimo con el ejercicio, así que para desatascarme un rato y cambiar aire (por decirlo de alguna manera) comencé a repasar las pruebas del inicio de la clase y a preparar mis apuntes. Fuera de eso a momentos la clase va bien supongo, no queda mucho para terminar y eso es bueno.

Pruebas Unitarias en Room

Observación: Para el ejemplo su usa un proyecto de ejemplo del Profesor, el cual consta de una UI en la que permite escribir un mensaje (dandole click a un botón flotante) y guardarlo en la base de datos para luego mostrar este mensaje en un recyclerview. Con esto explicado, las pruebas comienzan así:

Preparando el antes (@Before) y después (@After)

1. Se inicia probando si los mensajes que manda el usuario son guardados, o no, en la base de datos (por consiguiente las pruebas serán ejecutadas sobre la base de datos). Para esto se inicia creando los métodos de prueba en androidTests pues es una prueba instrumental (para probarlos con la interfaz, es decir usando el telefono). Es así que desde la clase de la base de datos, se le da click derecho a la función y se selecciona la opción "Generate" y luego "Create Test", dejando las opciones como se muestra a continuación y dandole click a OK (recordando que el directorio de destino es AndroidTest, no Test).



Con esto se genera el archivo de pruebas que luce de la siguiente manera:

```
class MensajeDatabaseTest {

    @Before
    fun setUp() {

    }

    @After
    fun tearDown() {

    }
}
```

2. Luego, para continuar se necesita crear las variables para la base de datos y el Dao en la clase de las pruebas, pues los métodos posteriores necesitaran el uso de estos elementos.

```
    private lateinit var db: MensajeDatabase
    private lateinit var dao: MensajeDao
```

3. Se continúa trabajando dentro de la función setUp (etiquetada @Before, pues es lo que se ejecuta antes de la prueba). Acá, se crea la base de datos para poder utilizarla, esto se hace con un nuevo método distinto a como creabamos las bases de datos antes (el método Room.databaseBuilder), ahora se usa "inMemoryDatabaseBuilder", la razon de este cambio es que como esto es una prueba, con este nuevo método no va a crear una base de datos persistente, es una base temporal que solo mientras se ejecuta la prueba se crea y almacena información, al terminar la prueba se elimina pues no necesitamos mantener persistencia de las pruebas realizadas.

Es importante mencionar que este nuevo método tambien pide el contexto y la clase de la base de datos, y acá en las pruebas el contexto se obtiene a través del ApplicationProvider (mira la val context). En cuanto a la clase, es la misma que creamos para hacer la base de datos.

Finalmente, como se adelantó creamos el dao (no grandes cambio, se hace como se viene haciendo hace rato).

```
 @Before
    fun setUp() {
        val context = ApplicationProvider.getApplicationContext<Context>()
        db =
Room.inMemoryDatabaseBuilder(context,MensajeDatabase::class.java).build()
        dao = db.mensajeDao()
    }
```

4. Luego en la función tearDown, lo único que se hace es cerrar la base de datos de la prueba una vez que esta se ha ejecutado (db.close()).

```
@After
    fun tearDown() {
        db.close()
    }
```

Recordar que como estamos preparando el entorno de la prueba, hay que decirle a la clase quien la va a ejecutar, por ende sobre el nombre de la misma clase se da la notación "@RunWith(AndroidJUnit4::class)", indicando que correra con JUnit4 para ejecutar el test y como algo nuevo, indicar que se usará "@SmallTest" (hay un uno medio y uno grande también, la diferencia entre ellos son los recursos disponibles que tengo para utilizar mediante la prueba, comparativa en la imagen), el cual nos permite probar la base de datos a nivel de memoria (sin la necesidad de construirla en realidad, porque este método no permite el uso de base de datos, si por algun motivo quisiera guardarlo fisicamente en el telefono, podria usar un MediumTest, pero ese no es el caso hoy)

Feature	Small	Medium	Large
Network access	No	localhost only	Yes
Database	No	Yes	Yes
File system access	No	Yes	Yes
Use external systems	No	Discouraged	Yes
Multiple threads	No	Yes	Yes
Sleep statements	No	Yes	Yes
System properties	No	Yes	Yes
Time limit (seconds)	60	300	900+

Lo dicho antes hace que el código quede de esta manera:

```
@RunWith(AndroidJUnit4::class)
@SmallTest
class MensajeDatabaseTest {

    private lateinit var db: MensajeDatabase
    private lateinit var dao: MensajeDao

    @Before
    fun setUp() {
        val context = ApplicationProvider.getApplicationContext<Context>()
        db =
Room.inMemoryDatabaseBuilder(context,MensajeDatabase::class.java).build()
        dao = db.mensajeDao()
    }

    @After
    fun tearDown() {
```

```
            db.close()
        }
```

Creando los métodos a probar:

Probando el agregar:

Para probar este método se crea la anotación "@Test" y se crea la función test_agregar(), la cual tiene un objeto mensaje con contenido "Prueba" y el dao para agregar el mensaje (dao.agregar(mensaje)). Al crear este dao, nos genera un error, pues como la función del Dao es de tipo suspend, señala que ese método solo puede ser llamado dentro de una corrutina

```
@Dao
interface MensajeDao {

    @Insert
    suspend fun agregar(mensaje: Mensaje)

    @Delete
    suspend fun eliminar(mensaje: Mensaje)

    @Update
    fun actualizar(mensaje: Mensaje)

    @Query( value: "select id,contenido from mensaje_table order by contenido")
    fun listar(): LiveData<List<Mensaje>>

    @Query( value: "select id,contenido from mensaje_table where id = :id")
    fun buscar(id:Int): LiveData<Mensaje>
}
```

Es así que como ventaja, para solucionar este error, Kotlin permite correr los bloques de código ("runBlocking") que se necesitan, pero a este punto no tenemos el runBlockingTest (de las pruebas), sólo el de las corrutinas normales del proyecto. Por ende necesitamos importar una dependencia en Gradle para ello, la cual es la siguiente:

```
//Test con Corrutinas
androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.4.2'
```

Agregando esto, ya puedo ejecutar el test directamente ejecutandolo como un bloque de prueba (runBlockingTest {...}). Al hacerlo, Android me sugiere que añada la notación "@ExperimentalCoroutineApi", pues es experimental y se ejecuta así. En este punto podriamos escoger entre darle la notación a cada

método a probar, o darsela a la clase, para que esta indique que cada método en test es Experimental (se opta por la segunda opción)

```
@ExperimentalCoroutineApi
@RunWith(AndroidJUnit4::class)
@SmallTest
class MensajeDatabaseTest { ... }
```

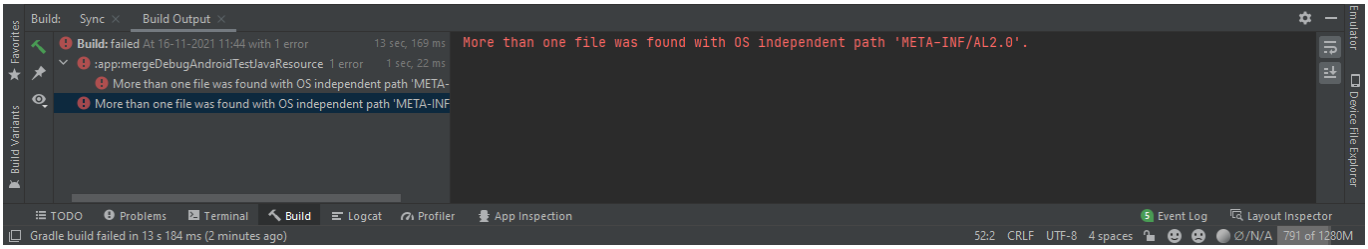
Con esto solucionado, se agrega al mismo método test_agregar() la lista (val lista = dao.listar()) que va a mostrar los mensajes agregados y para tener Asserts (aserciones) más claras, añadimos la libreria de Google Truth al proyecto.

```
//test Google Truth
androidTestImplementation "com.google.truth:truth:1.1.3"
androidTestImplementation "androidx.arch.core:core-testing:2.0.0"
```

Con esto, a la lista agregada le puedo agregar la aserción de Google Truth e indicar (con él código) la siguiente consulta, "De la lista que saco de la base de datos, esta contiene el mensaje que acabo de agregar?" (Truth.assertThat(lista.value).contains(mensaje)), quedando el método de la siguiente manera.

```
    @Test
    fun test_agregar() = runBlockingTest {
        var mensaje:Mensaje = Mensaje("Prueba")
        dao.agregar(mensaje)
        val lista = dao.listar()
        Truth.assertThat(lista.value).contains(mensaje)
    }
```

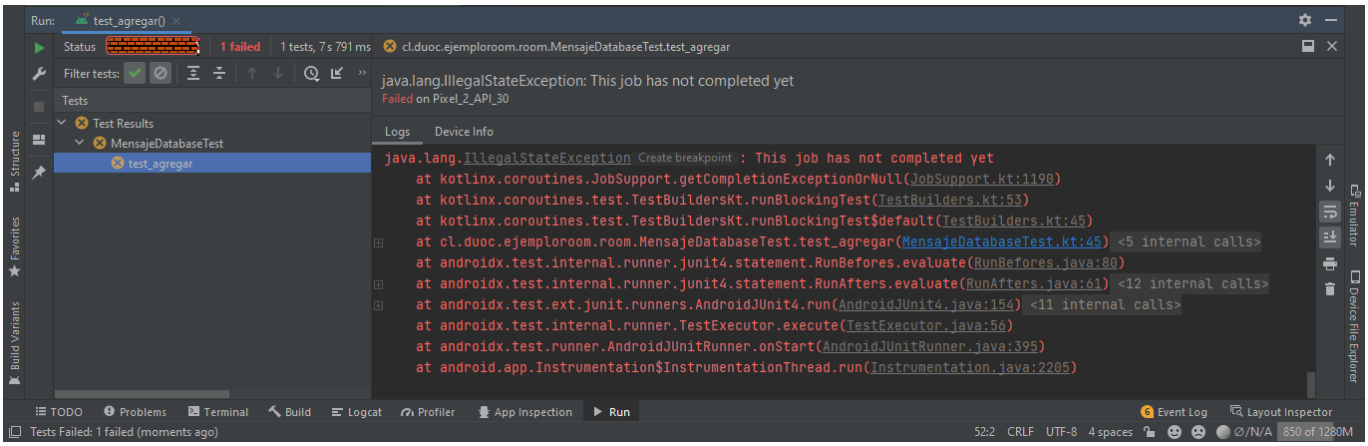
Es así que hasta este punto en el método ya esta gregado el mensaje, listado para ser mostrado y disponible para iniciar la prueba, por ende le damos click al play verde para obtener los resultado de la ejecución de la prueba, y.... ***!!Chingale wey, Error¡¡***



Este error presentado arriba es de la corrutinas (del runBlockingTest), ya que como es experimental aún tiene algunos bugs, pero lo importante, ¿Cómo se arregla?, copiando el siguiente código dentro de android del build Gradle (donde estan las dependencias) para que así no incluya ciertas rutas que son las que dan problema a la corrutina.

```
android{

    packagingOptions {
        exclude "**/attach_hotspot_windows.dll"
        exclude "META-INF/licenses/**"
        exclude "META-INF/AL2.0"
        exclude "META-INF/LGPL2.1"
    }
}
```

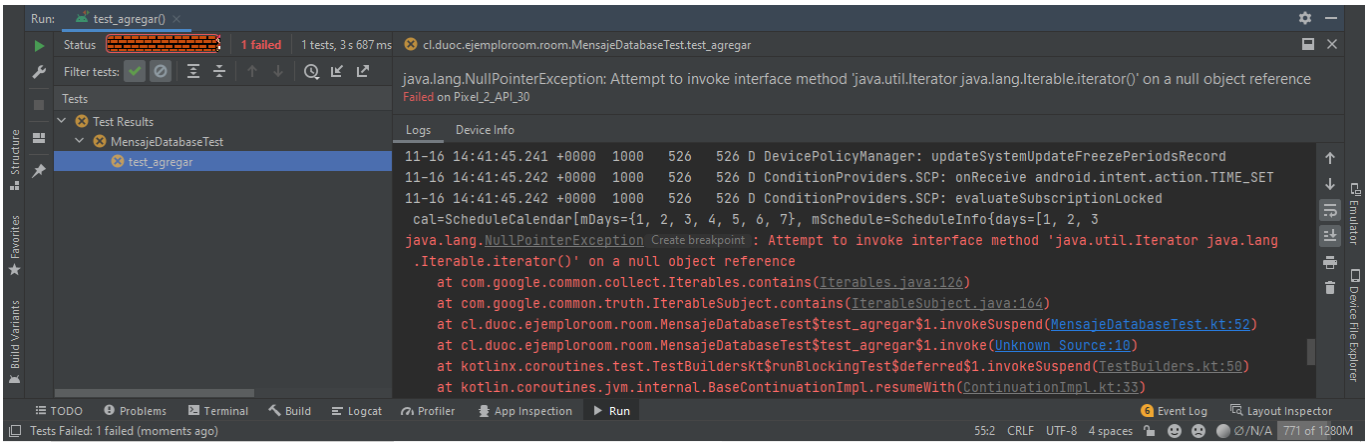
Con esta primera solución, se ejecuta nuevamente la prueba y observamos que genera otro error distinto al de antes (indicando que solucionamos uno, pero se desencadena otro jajajajaja).



Este error indica que el trabajo no se a completado aún ("job has not completed yet"). Traducido a humano esto es que el test corre en un bloque de prueba, por lo tanto el Dao se hace dentro de esa prueba, pero para obtener el contenido se tiene que hacer en otro hilo (el hilo principal, porque es un LiveData), por lo tanto falla el test porque no puede hacer lo que se le pide. Para corregir esto, hay que agregar un regla al proyecto, la cual va a hacer que todo se ejecute al instante y no tenga que esperar nada. La regla es la siguiente (notar la anotacion @get:Rule que la precede e indica lo que es)

```
@get:Rule
var instantTaskExecutorRule = InstantTaskExecutorRule()
```

Con esto, para no ser menos y tener pocos errores, probamos de nuevo y nos arroja el siguiente error a solucionar.



Este tercer error nos aclara que no se puede invocar el método pedido sobre un objeto nulo. esto es, que la "lista.value" da null, ¿Por qué da null?, porque como es LiveData necesito de un observe que vigile la información, no lo puede hacer uno mismo (no le puedo decir saquemos contenido a penas lo listemos), por ende al no tener este componente la lista esta nula porque no podemos ver el mensaje en la base de datos. Para dar solución a esto, es que implementamos una clase de ejemplo que va a actuar como observer (fue copiar y pegar, así que no profundizaré demasiado en ese código, el profe lo sacó de acá --> link, otro ejemplo directo de Google desde acá --> link). Como dije, esta clase es copiar y pegar y la generamos en la raíz de las pruebas quedando de la siguiente manera (la nueva clase es un archivo que se llama LiveDataUtilTest)

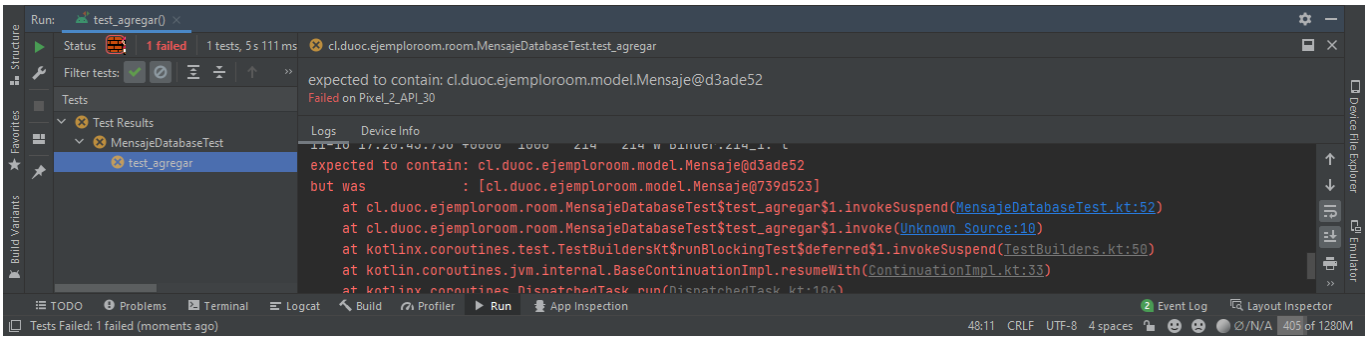
```
package cl.duoc.ejemploroom

import androidx.lifecycle.LiveData
import androidx.lifecycle.Observer
import java.util.concurrent.CountDownLatch
import java.util.concurrent.TimeUnit
import java.util.concurrent.TimeoutException

fun <T> LiveData<T>.getOrAwaitValue(
    time: Long = 2,
    timeUnit: TimeUnit = TimeUnit.SECONDS
): T {
    var data: T? = null
    val latch = CountDownLatch(1)
    val observer = object : Observer<T> {
        override fun onChanged(o: T?) {
            data = o
            latch.countDown()
            this@getOrAwaitValue.removeObserver(this)
        }
    }

    this.observeForever(observer)

    // Don't wait indefinitely if the LiveData is not set.
    if (!latch.await(time, timeUnit)) {
        throw TimeoutException("LiveData value was never set.")
    }
```

```
    @Suppress("UNCHECKED_CAST")
    return data as T
```

Con esta clase creada, se origina automáticamente el observer del LiveData en las pruebas, esto mediante un nuevo método llamado llamado "getOrAwaitValue()" (notar que este método observa los valores, values, de lista, por tanto ya no es necesario ponerlos abajo en la aserción). Lo anterior dicho queda de la siguiente manera.

```
    @Test
    fun test_agregar() = runBlockingTest {
        var mensaje:Mensaje = Mensaje("Prueba")
        dao.agregar(mensaje)
        val lista = dao.listar().getOrAwaitValue()
        Truth.assertThat(lista).contains(mensaje)
    }
```

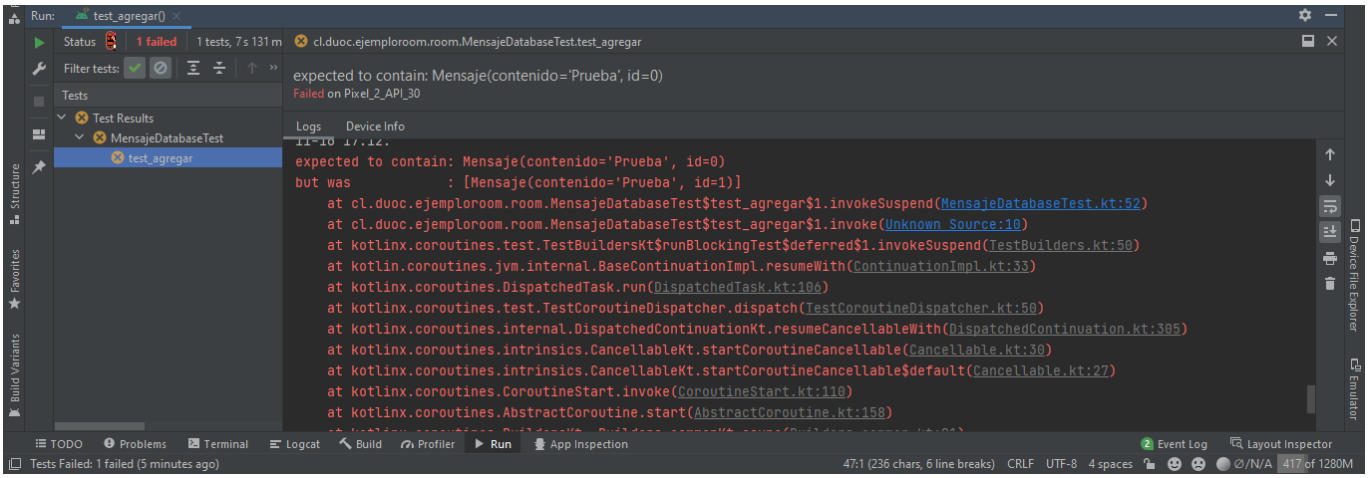
Luego, como es lo usual, se arregla esto y se nos generará otro error, el cual muestro a continuación.



En este error se muestra que el test no puede efectuarse, pero la descripción es un poco críptica pues nos da esta descripción, **"expected to contain: cl.duoc.ejemploroom.model.Mensaje@d3ade52, but was : [cl.duoc.ejemploroom.model.Mensaje@739d523]"**, esto se debe a que está mostrando los espacion en memoria del error y no la descripción del error en si misma. Para corregir esto y ver un error más "amigable" se sobrescribe el método "toString" en el modelo, quedando como se ve a continuación.

```
@Entity(tableName = "mensaje_table")
class Mensaje(var contenido:String) {

    @PrimaryKey(autoGenerate = true)
    var id:Int = 0
    override fun toString(): String {
        return "Mensaje(contenido='$contenido', id=$id)"
    }
}
```

De esta manera el error (ahora legible queda de la siguiente manera).

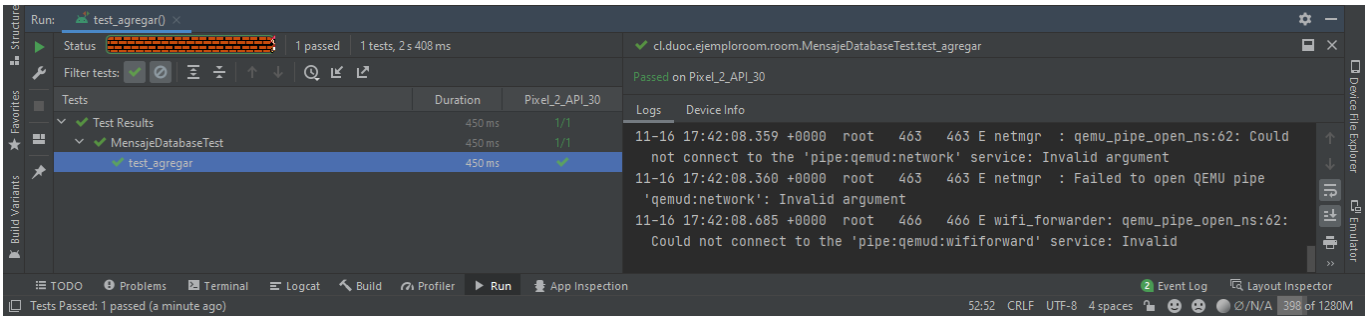


Es así que mirando esto nos damos cuenta que el test falla porque las id de lo que se obtiene y lo que esperamos obtener son diferentes, la razón de esto se origina en la @PrimaryKey del modelo, pues al ser autogenerada esperabamos para el primer id=0, pero la instrucción no genera una id=1 para el ítem y por esto origina la variación. Ahora bien, esta prueba es perfectamente válida para cuando nosotros queremos probar el método con la id **NO** autogenerada (autogenerated = false), pues es correcto que las id sean distintas, por lo que en ese caso, a pesar de ser una prueba fallida, esta prueba es correcta (es correcto que falle).

Si quisieramos por otro lado, probar un método como el de acá, donde la id es autogenerada, tendriamos que hacer la siguiente variación en el código.

```
    @Test
    fun test_agregar() = runBlockingTest {
        var mensaje: Mensaje = Mensaje("Prueba")
        dao.agregar(mensaje)
        //notar el cambio en las lineas de abajo
        val mTest = dao.buscar(1).getOrAwaitValue()
        Truth.assertThat(mTest.contenido).isEqualTo(mTest.contenido)
    }
}
```

Con esto listo, sorpresivamente ya no hay errores y el método al completo pasa la prueba con ticket verde, por lo que el método completo, para este caso en particular (@PrimaryKey (autogenerated = true)) es correcto.

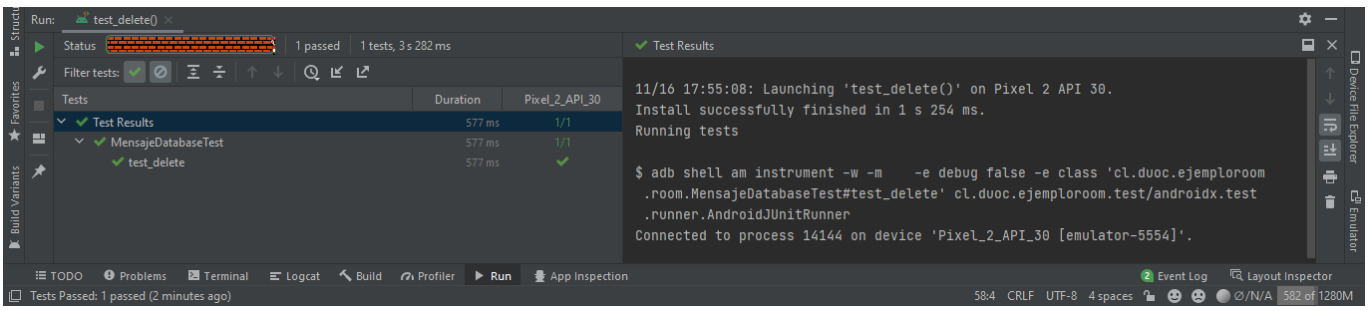


Probando el delete:

Despues de ver toda la montonera de errores del agregar uno pensaria que el borrar (delete) sería igual de complejo, pero en realidad es más corto y sencillo. Al igual que en el anterior se le da la notación "@Test" y se crea la función que corre en el bloque "runBlockingTest" de la corrutina, se crea un objeto de tipo mensaje, se le da el agregar del dao (pues no puedo borrar algo sin agregarlo primero) y luego se elimina, se sigue dandole la lista donde se listará el mensaje agregado y con las aserciones de Google se le dice que, "Si la lista NO contiene m, significa que lo eliminó. Si no es así no se ha eliminado y por tanto el método no es correcto" (Truth.assertThat(lista).doesNotContain(m)). Con esto listo se prueba el método

```
@Test
 fun test_delete() = runBlockingTest {
     var m:Mensaje = Mensaje ("Prueba Eliminar")
     dao.agregar(m)
     dao.eliminar(m)

     val lista = listOf(dao.listar().getOrAwaitValue())
     Truth.assertThat(lista).doesNotContain(m)
 }
```

Como se puede ver acá el método es correcto (ticket verde), por tanto el mensaje ingresado no se encuentra en la lista y por consiguiente fue eliminado al ejecutar el método.



Algo importante a mencionar es que este test también está sujeto a ser modificado, pues podria darse el caso que no borre porque las id son distintas, en ese caso se haría lo mismo que paso con el método agregar (dandole una id de antemano para que lo que espero borrar y lo que borre sea lo mismo).

Hasta acá las funciones evaluadas en clases. Añado un Extra de una pregunta del Seba que me parecio interesante.

Extra (Pregunta del Seba):

La pregunta fue la siguiente:

En la prueba eliminar... ¿Valdría la pena evaluar también la cantidad de registros antes de agregar el registro con el después de eliminar el registro y que sean iguales?, para así evitar el problema de que falle la comparación en sí, como una especie de doble checkeo.

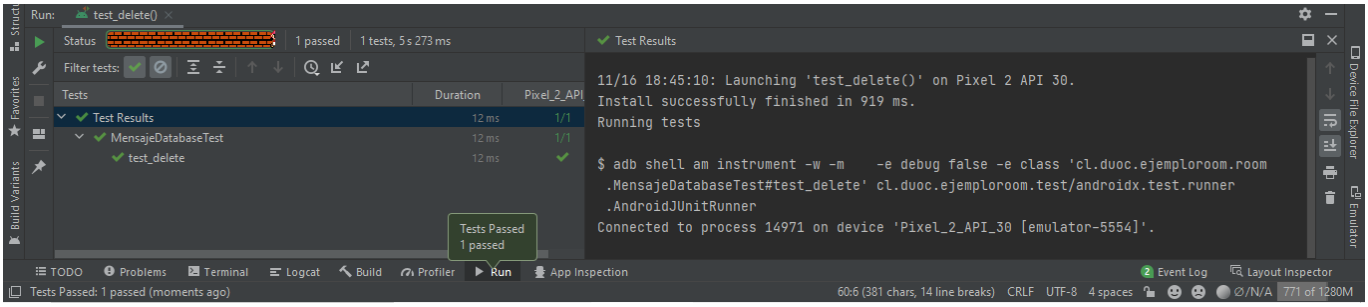
La respuesta del profe fue que es posible, por lo que crea una función que genere hartos objetos para ser evaluados, agregar estos objetos al abase de datos y despues eliminar uno específico dentro de la lista. La función que genera varios objetos es la siguiente:

```
    private fun generarLista() : List<Mensaje> {

        var lista:ArrayList<Mensaje> = ArrayList()
        for (i in 0.. 10){
            var m = Mensaje ("contenido $i")
            lista.add(m)
        }
        return lista
    }
```

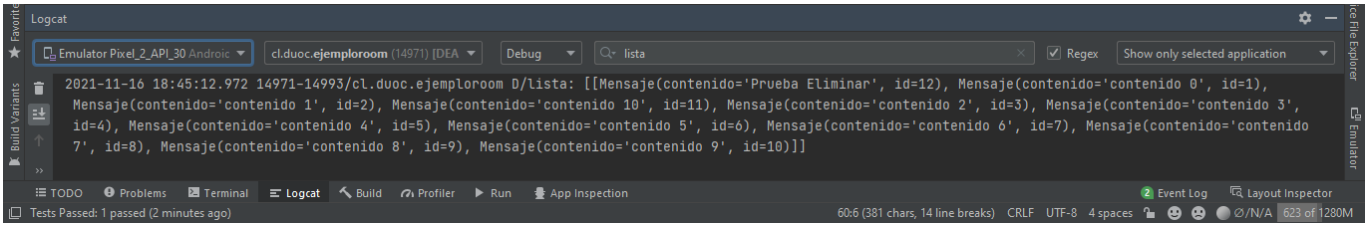
Luego modifica el test_delete() para poder agregarle en la prueba estos 10 objetos creados en la función anterior.

```
    @Test
    fun test_delete() = runBlockingTest {

        for (i in generarLista()){
            dao.agregar(i)
        }

        var m:Mensaje = Mensaje ("Prueba Eliminar")
        dao.agregar(m)
        dao.eliminar(m)

        val lista = listOf(dao.listar().getOrAwaitValue())
        Log.d("lista", lista.toString())
        Truth.assertThat(lista).doesNotContain(m)
    }
```

Y al ejecutarlo finalmente le da correcto, así que si es posible realizar el doble checkeo del Seba agregando 10 elementos a la lista.



(Test exitoso)



(Log que muestra los 10 elementos agregados a la base de datos)

Hasta acá mi avance por hoy, en clases fue más un resumen para prepara los conocimientos en la clase de hoy y dejarlos listos para la continuación mañana. Muchas gracias.

Leonardo Rodenas Escobar 😁