Testing

Testing (Parte II)

{desafío} latam_

Implementar tests unitarios y de instrumentación para la verificación del buen funcionamiento de los componentes de un proyecto Android.

- Unidad 1:
  Acceso a datos en Android
- Unidad 2:
  Consumo de API REST
- Unidad 3:
  Testing
- Unidad 4:
  Distribución del aplicativo Android

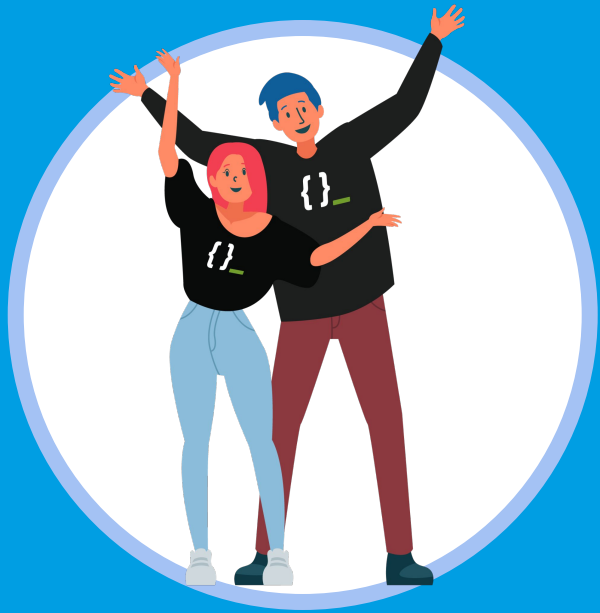
Te encuentras aquí

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *MockWebServer*
- *Robolectric*
- *Cómo testear coroutines*
- *Cómo testear LiveData/StateFlow*

¿Cuál es la diferencia entre probar una función asíncrona y una no asíncrona?



/* MockWebServer */

MockWebServer

MockWebServer es una herramienta para probar clientes y servidores HTTP en aplicaciones de Android.

Permite crear un servidor HTTP local que responde a las solicitudes entrantes con respuestas predefinidas, para que puedas probar tu código de cliente HTTP sin tener que acceder a un servidor real.

MockWebServer

- Con **MockWebServer**, puede simular diferentes condiciones de red (como conexiones lentas o poco confiables), probar casos extremos y condiciones de error, y verificar que tu cliente envíe las solicitudes correctas y maneje las respuestas correctamente.

- MockWebServer es parte de la biblioteca **OkHttp** y se puede usar con cualquier cliente HTTP que admita HTTP/1.1 o HTTP/2. Proporciona una API para configurar respuestas simuladas, inspeccionar solicitudes entrantes y controlar el comportamiento del servidor durante las pruebas.

MockWebServer - Ejemplo

En el siguiente ejemplo, tenemos la clase HttpClientTest, la cual vamos a separar en 3 partes: Before, After y Test.

Comenzamos definiendola como cualquier clase:

```
class HttpClientTest {

    private lateinit var mockWebServer: MockWebServer
    private lateinit var httpClient: OkHttpClient

    . . .
}
```

Before: en before creamos e iniciamos el webserver y el cliente HTTP

```
@Before
    fun setUp() {
        mockWebServer = MockWebServer()
        mockWebServer.start()

        httpClient = OkHttpClient.Builder()
            .connectTimeout(500, TimeUnit.MILLISECONDS)
            .readTimeout(500, TimeUnit.MILLISECONDS)
            .build()
    }
```

After: aquí necesitamos definir una función que detenga el webserver:

```
@After
    fun tearDown() {
        mockWebServer.shutdown()
    }
```

Test: en esta parte definimos la función que realiza el test, puede ser más de una función, dependiendo que es lo que quieres probar :

```
@Test
    fun testHttpCall() {
        val responseBody = "Hello, world!"
        val mockResponse = MockResponse()
            .setResponseCode(HttpURLConnection.HTTP_OK)
            .setBody(responseBody)
        mockWebServer.enqueue(mockResponse)

        val request = Request.Builder()
            .url(mockWebServer.url("/path/to/resource"))
            .build()

        val response = httpClient.newCall(request).execute()

        assertEquals(responseBody, response.body?.string())
    }
```

MockWebServer - Explicación

En este ejemplo, estamos creando una instancia de MockWebServer en el método `setUp()`, iniciándola y creando una nueva instancia de `OkHttpClient` que usaremos para realizar solicitudes `HTTP`. En el método `testHttpCall()`, estamos configurando una respuesta simulada que se devolverá cuando el cliente realice una solicitud al servidor simulado, poniendo en cola la respuesta con `mockWebServer.enqueue()`, realizando una solicitud al servidor simulado usando `httpClient.newCall()`, y afirmando que el cuerpo de la respuesta coincide con lo que esperamos.

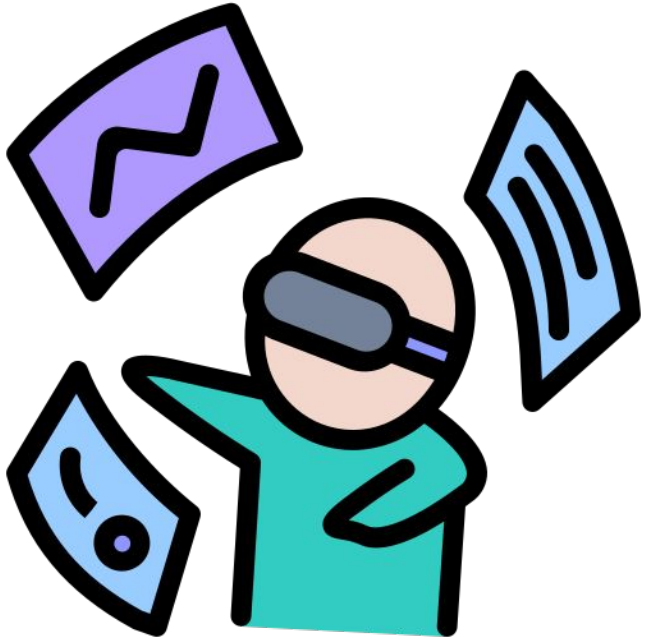
Este es solo un ejemplo simple, pero MockWebServer ofrece muchas más funciones y opciones para configurar su servidor simulado y verificar las solicitudes.

/* Robolectric */

Robolectric

Robolectric es un framework de prueba de código abierto que permite a los desarrolladores ejecutar pruebas unitarias en aplicaciones de Android sin necesidad de usar un emulador de Android o un dispositivo físico. Simula el sistema operativo Android y proporciona un entorno de espacio aislado para que se ejecuten las pruebas. Esto hace que las pruebas sean más rápidas y eficientes, ya que elimina la necesidad de implementar e iniciar la aplicación en un dispositivo o emulador para cada prueba.

Robolectric también brinda fácil acceso a los servicios del sistema Android y permite a los desarrolladores simular diferentes configuraciones, como tamaños de pantalla y orientaciones, para garantizar que la aplicación se comporte correctamente en diferentes condiciones.



Demostración
"¿Cómo testear coroutines?"

¿Cómo testear coroutines?

Probar rutinas en Android con Kotlin es una tarea común cuando se escribe código asíncrono en aplicaciones de Android. Aquí hay un ejemplo de cómo probar una rutina usando JUnit y KotlinTest:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.test.*
import org.junit.*
import org.junit.runner.*
import kotlin.coroutines.*
```

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

    private val testCoroutineScope = TestCoroutineScope(TestCoroutineDispatcher())

    @Before
    fun setUp() {
        Dispatchers.setMain(testCoroutineScope.dispatcher)
    }

    @After
    fun tearDown() {
        Dispatchers.resetMain()
        testCoroutineScope.cleanupTestCoroutines()
    }

    @Test
    fun testCoroutine() = testCoroutineScope.runBlockingTest {
        val result = async {
            // Perform some asynchronous operation
            delay(1000)
            "Hello, World!"
        }

        val message = result.await()
        Assert.assertEquals("Hello, World!", message)
    }
}
```

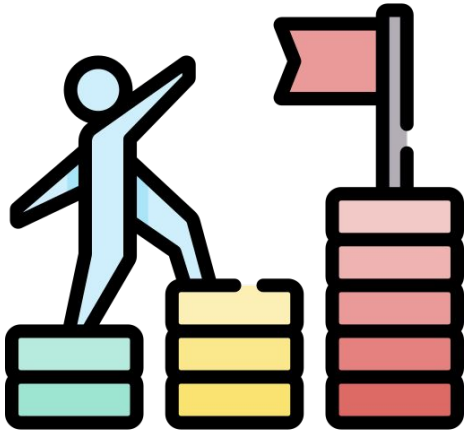
¿Cómo testear coroutines?

- En este ejemplo, creamos un `TestCoroutineScope` con un `TestCoroutineDispatcher`, que es un despachador que ejecuta rutinas sincrónicas y determinísticas. Luego usamos la función `runBlockingTest` para ejecutar una rutina y esperar a que se complete.

- Ten en cuenta que también configuramos el despachador de rutina principal en el despachador de prueba en la función de configuración y lo restablecemos en la función de desmontaje. Esto es necesario para garantizar que todas las coroutines iniciadas por la prueba se ejecuten en el despachador de prueba.

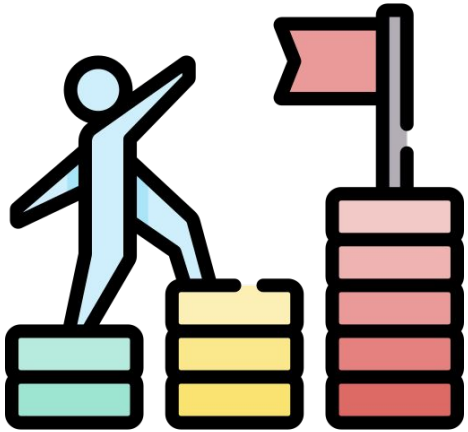
- Mediante el uso de coroutine y `TestCoroutineScope` de esta manera, podemos escribir pruebas para código asíncrono de una manera simple y directa.

Pasos básicos para testear una coroutine



1. Importa las clases y anotaciones necesarias: `kotlinx.coroutines.*`, `kotlinx.coroutines.test.*`, `org.junit.*`, `org.junit.runner.*` y `kotlin.coroutines.*`.
2. Agrega la anotación `@RunWith(AndroidJUnit4::class)` a la clase de prueba para indicar que debe ejecutarse con JUnit.
3. Crea un `TestCoroutineScope` con un `TestCoroutineDispatcher`.
4. Usa la anotación `@Before` para establecer el despachador de rutina principal en el despachador de prueba.

Pasos básicos para testear una coroutine



5. Utiliza la anotación `@After` para restablecer el despachador de corrutinas principal y limpiar las corrutinas de prueba una vez finalizada la prueba.
6. Utiliza la anotación `@Test` para definir la función de prueba.
7. Usa la función `runBlockingTest` para ejecutar una rutina y esperar a que se complete.
8. Usa `async` para realizar alguna operación asíncrona y obtenga el resultado con await.
9. Utiliza una biblioteca de aserciones para verificar el resultado.

Demostración "Cómo testear LiveData/StateFlow"

{desafío} latam_

¿Cómo testear LiveData/StateFlow?

Para probar una función de StateFlow en Android, puede seguir estos pasos generales:

- Importa las clases y anotaciones necesarias: kotlinx.coroutines.*, kotlinx.coroutines.test.*, org.junit.*, org.junit.runner.* y kotlinx.coroutines.flow.*.
- Agrega la anotación `@RunWith(AndroidJUnit4::class)` a la clase de prueba para indicar que debe ejecutarse con JUnit.
- Crea un TestCoroutineScope con un TestCoroutineDispatcher.
- Usa la anotación @Before para establecer el despachador de rutina principal en el despachador de prueba.

¿Cómo testear LiveData/StateFlow?

- Utiliza la anotación **`@After`** para restablecer el despachador de corrutinas principal y limpiar las corrutinas de prueba una vez finalizada la prueba.
- Utiliza la anotación **`@Test`** para definir la función de prueba.
- Cree una instancia del StateFlow que desea probar.
- Recopila los valores emitidos por StateFlow usando **`flowTest`**.
- Utiliza la función **`assertThat`** de la biblioteca Truth para verificar los valores emitidos por StateFlow.

Imports necesarios

```
import com.google.common.truth.Truth.assertThat
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.test.TestCoroutineDispatcher
import kotlinx.coroutines.test.TestCoroutineScope
import kotlinx.coroutines.test.runBlockingTest
import org.junit.After
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4
```

Crea una clase `StateFlowTest` y agrega las siguientes funciones con sus respectivas anotaciones

```
private val testDispatcher = TestCoroutineDispatcher()
    private val testScope = TestCoroutineScope(testDispatcher)

    @Before
    fun setup() {
        Dispatchers.setMain(testDispatcher)
    }

    @After
    fun cleanup() {
        Dispatchers.resetMain()
        testScope.cleanupTestCoroutines()
    }
```

Luego agrega la función `@Test`

```
@Test
    fun stateFlowEmitsValues() = testScope.runBlockingTest {
        // Crea un MutableStateFlow
        val stateFlow = MutableStateFlow(listOf(1, 2, 3))

        // Captura los valores emitidos
        val values = stateFlow.collect {
            // No hacer nada aquí
        }

        // Verifica los valores emitidos por el StateFlow
        assertThat(values).isEqualTo(listOf(1, 2, 3))
    }
```

En este ejemplo, la prueba `stateFlowEmitsValues` crea un `MutableStateFlow` con una lista inicial de enteros y recopila los valores emitidos por StateFlow.

Finalmente, utiliza la función assertThat de la librería Truth para verificar que los valores emitidos por el StateFlow son iguales a la lista inicial.

¿Qué tipo de restricción podrías encontrar al ejecutar tests instrumentados en el emulador de Android?

Próxima sesión...

- *Desafío*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales