Evidencia dia 04 - Semana 16

Leonardo Rodenas Escobar

Reflexión:

La clase de hoy estuvo bastante mejor que la de ayer, por fin pudimos ver como se hacen las pruebas de la API con MockWebServer e implementarlas. El resto del horario comprende seguir con el proyecto anterior o empezar el proyecto final de evaluación. Por mi parte la vacuna contra el COVID-19 que me puse hayer me pegó fuertísimo, así que presento acá mis apuntes de la clase a modo de evidencia.

Pruebas instrumentales con MockWbeServer:

- **Ojo** --> Como proyecto base para ejecutar las pruebas se utiliza el proyecto de la lista de imagenes de los perros que se estudió en clases.

Para comenzar a utilizar el proyecto e implementar las pruebas (la idea es probar la API, sin necesidad de internet) se inicia con la implementación de las dependencias necesarias, MockWebServer y Google Truth, estas son las siguientes:

```
//MockWebServer
androidTestImplementation "com.squareup.okhttp3:mockwebserver:3.12.13"
androidTestImplementation "com.google.truth:truth:1.1.3"
```

Luego, como vamos a trabajar las pruebas de forma local, en el cliente se cambia la BASE_URL, pues de esta forma como lo tenemos tiene una Url fija y no una que podamos pasarle como parámetro a la función, por ende se modifica el método de la siguiente manera:

```
//MÉTODO ORIGINAL (Notar el cambio en baseUrl(URL_BASE) con respecto al método de
abajo)

class MascotaClient {

    companion object{
        val URL_BASE = "https://dog.ceo/api/breed/"

        private val cliente = MascotaClient

        fun getCliente(): MascotaService{
            val retrofit =
Retrofit.Builder().baseUrl(URL_BASE).addConverterFactory(GsonConverterFactory.crea
te()).build()
            return retrofit.create(MascotaService::class.java)
        }
```

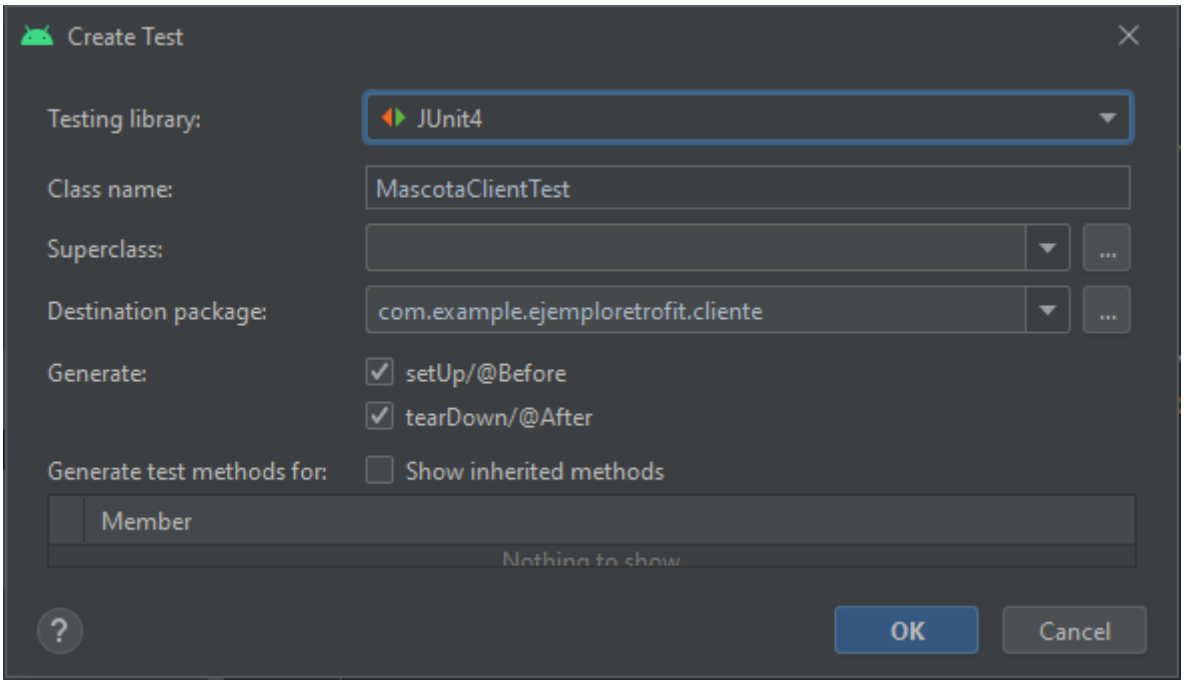
```
    }
}

//MÉTODO CON CAMBIO PARA LAS PRUEBAS


class MascotaClient {

    companion object{
        val URL_BASE = "https://dog.ceo/api/breed/"

        private val cliente = MascotaClient

        fun getCliente(url:String): MascotaService{
            val retrofit =
Retrofit.Builder().baseUrl(url).addConverterFactory(GsonConverterFactory.create())
.build()
            return retrofit.create(MascotaService::class.java)
        }
    }
}
```

Con este cambio lo que sucede es que donde se pide esta url cambiada (en el ViewModel) da un error, por tanto la "val URL_BASE" se hace constante (const) y se le pasa en la ubicación requerida al ViewModel.

```
//En el Cliente de Retrofit

const val URL_BASE = "https://dog.ceo/api/breed/"

//En ViewModel

class MascotaViewModel : ViewModel() {

    //cambio en getCliente(), se agrega (MascotaClient.URL_BASE)
    private val service = MascotaClient.getCliente(MascotaClient.URL_BASE)
    val mascota = MutableLiveData<Mascota>()
}
```

Con estos cambios hacemos que el método getCliente reciba la url, y se pueda probar con cualquiera de ellas, no solamente con la que se pone al principio, entonces sirve para probarlo de manera local.

Para seguir con la preparación de las pruebas, el siguiente paso es crear el test instrumental donde se realizarán las pruebas, por ende, desde el cliente (MascotaCliente) click derecho y lo de siempre:



Con esto se genera el Test (MascotaClientTest), con su @Before y @After y para completarlo le añadimos el @RunWith(AndroidJUnit4::class) y creamos un objeto de la clase MockWebServer, quedando como se ve a continuación:

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

    private var server = MockWebServer()

    @Before
    fun setUp() {

    }

    @After
    fun tearDown() {
    }
}
```

Luego, con el server ya creado, lo que se hace es configurarlo, por ende en el bloque de código de la función setUp, se le pasa lo siguiente:

```
 private val body = ""

 @Before
    fun setUp() {
        server.start(8080)
        server.enqueue(MockResponse().setResponseCode(200).setBody(body))
```

```
        server.url("test/raza/images")
    }
```

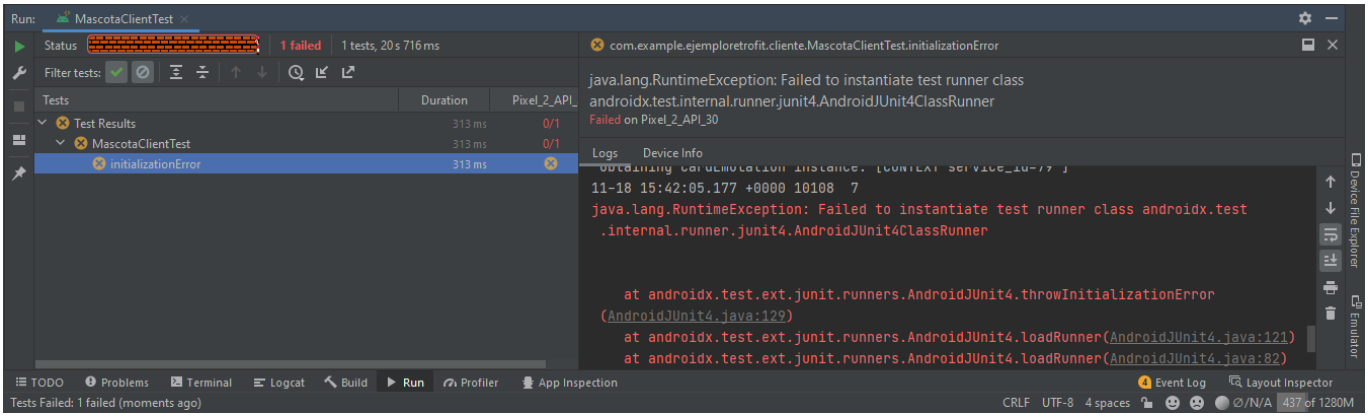
Notar que acá se configuran 3 cosas, estas son:

1. En la primera línea se comienza iniciando el server en el puerto 8080 (Puerto 8080: es el puerto alternativo al puerto 80 TCP para servidores web, normalmente se utiliza este puerto en pruebas).
2. En la segunda línea, este server iniciado se pone activo al ponerlo en "enqueue", este método recibe la respuesta y un código de programación (200 = Éxito/OK) y un body, esto es un poco más largo y complejo así que se describirá en detalle más adelante, por ahora solamente se le pasa una variable String vacía llamada body.
3. Finalmente, en la tercera línea se le da una ruta, la cual es "test/akita/images/", la razón de esto es que como estamos buscando de manera local, le damos la dirección entera (en este caso test = https://dog.ceo/api/breed/ y raza/images = la raza de prueba que vamos a buscar).

Con esto listo (a medias, recordar que falta lo del body) se configura la parte de @After, en donde sola y sencillamente se le dice al server que se cierre una vez ejecutadas las pruebas.

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

Hasta acá se puede probar el test, pero dará error pues sólo tenemos el antes y después, pero nada que probar (adjunto imagen para verlo, pero esto se soluciona creando los métodos a probar).



Como se puede ver (y dije arriba), la solución a esto es crear los métodos a probar, pero ¿Qué retornarán estos métodos?, una respuesta... ¿Qué trae la respuesta?, las imagenes de las razas que le pida... y ¿Dónde está eso si estoy trabajando de manera local?, cri-cri... cri-cri, jajajajaj. Para solucionar esto es que se crea el cuerpo de la respuesta (el body faltante de antes), a través de un archivo Json que contenga la respuesta de la raza a pedir.

Por consiguiente, para esto desde la carpeta app de la aplicación se crea un nuevo directorio (no folder, directory) llamado assets y dentro del cual se crea el archivo (file, no class) akita.json (que actuará de modelo para la respuesta) y hay pego el json de la respuesta de la API (copiado del navegador web) quedando de esta manera:

```
{
  "message": [
    "https:\/\/images.dog.ceo\/breeds\/akita\/512px-Ainu-Dog.jpg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/512px-Akita_inu.jpeg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/Akina_Inu_in_Riga_1.jpg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/Akita_Dog.jpg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/Akita_Inu_dog.jpg",

"https:\/\/images.dog.ceo\/breeds\/akita\/Akita_hiking_in_Shpella_e_Pellumbasit.jp
g",
    "https:\/\/images.dog.ceo\/breeds\/akita\/Akita_inu_blanc.jpg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/An_Akita_Inu_resting.jpg",
    "https:\/\/images.dog.ceo\/breeds\/akita\/Japaneseakita.jpg"
  ],
  "status": "success"
}
```

Ahora toca hacer que el MockwebServer se conecte a ese Json y traiga la respuesta local solicitada. Es así que para esto se crea un lector de archivos, partiendo de la base de crear en la carpeta de AndroidTest un "new-Kotlin Class-Kotlin Object" llamado FileReader, el cual codificado queda de la siguiente manera.

```
object FileReader {

    fun lectorJson(archivo:String):String
    {
        val input =
InstrumentationRegistry.getInstrumentation().targetContext.applicationContext.asse
ts.open(archivo)
        val builder = StringBuilder()
        val lector = InputStreamReader(input,"UTF-8")
        lector.readLines().forEach {
            builder.append(it)
        }
        return builder.toString()
    }
}
```

Notar que lo que hace el código de arriba es que el la primera línea crea el archivo, luego construye strings usando ese archivo, le sigue que el lector tenga un modo de lectura (UTF-8 en este caso) y que cada vez que

lea una línea, por cada una de ellas la agregue (ignorando guiones y espacios) y al final que esto que leyó lo retorne como uan String.

Con esto listo, se conforma el cuerpo de la respuesta haciendo el siguiente cambio en el body que teniamos antes.

```
//En MaccotaClienteTest, cambiar esto

private val body = ""

//Por eso

private val body = FileReader.lectorJson("akita.json")
```

Es así que despues de todo esto la respuesta logra quedar aislada y dentro del servidor local de pruebas, por ende no necesitamos conectarnos a internet para poder realizarla y recibir la respuesta a las peticiones de razas que realicemos y recién, ya podemos crear un método a probar.

Probando si la llamada a la API es exitosa:

Se parte creando el método como es habitual (con @Test), luego la función a probar con la nueva url, pero ¿Por qué es http://localhost:8080/test/?, la razon de esto es que vamos a probar nuestra url local, y como al inicio al confifurar el @Before le dimos "server.start(8080)", ese 8080 hace que el inicio de la url sea http://localhost:8080/ (fijo es así por defecto al ser puerto 8080) y el test, se añade porque el cliente trae el resto, por tanto no es necesario agregar los de raza/images. Finalmente esta primera línea se completa dandole la raza a buscar, en este caso "akita" (recordar que al poner akita, se cambia el bloque @Before a como sigue para que busque la palabra "akita" y no "raza"), todo esto queda así en código.

```
@Test
    fun test_apiSuccess(){
        val call =
MascotaClient.getCliente("http://localhost:8080/test/").getMascotas("akita")
}

// Cambiar esta parte, akita en vez de raza

server.url("test/raza/images") -- cambio a --> server.url("test/akita/images")
```

Con este cambio se agregan las siguientes dos líneas dentro de la función, quedand así el método:

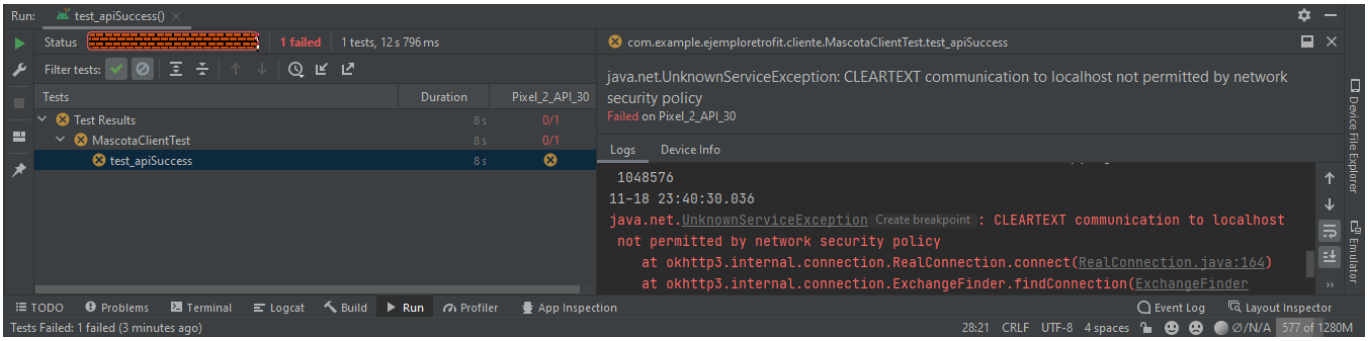
```
@Test
fun test_apiSuccess() {
```

```
    val call =
MascotaClient.getCliente("http://localhost:8080/test/").getMascotas("akita")
    var mascota = call.execute().body()
    Truth.assertThat(mascota!!.message[0].toString()).isEqualTo("
https://images.dog.ceo/breeds/akita/512px-Ainu-Dog.jpg")

}
```

Estas líneas agregadas sirven para:

- En la primera pedir una mascota, ejecutarla la acción y traer el cuerpo de la petición (la respuesta del archivo akita.json)
- En la segunda, hacer la aserción que comprueba que si la mascota en posición 0 (la primera) es igual al link que le pasamos como lo que esperamos (link que viene de la primera posición en el akita.json pero con los slash correctos para que no deban interpretarse), entonces el método es correcto.

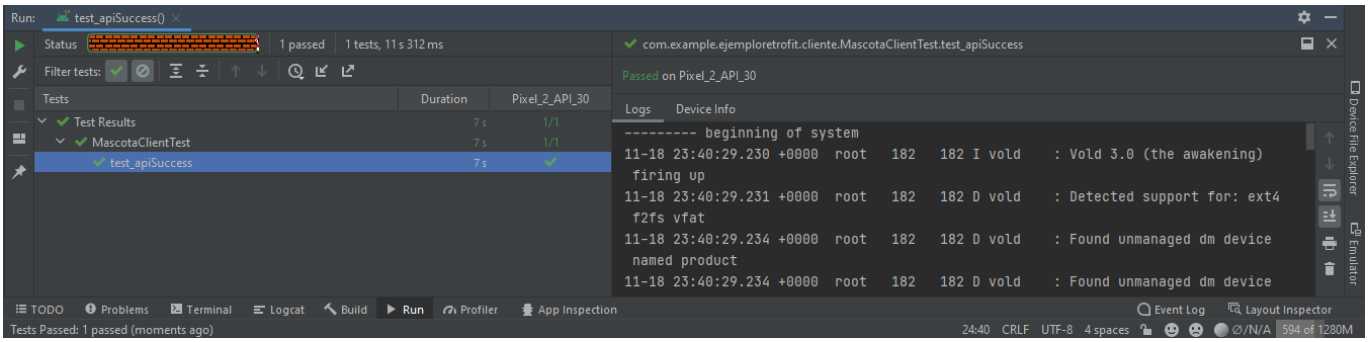
Así, el primer método esta listo, ejecutamos y... !!ERROR¡¡



Este error indica que que no tenemos permitido la conexión a direcciones no seguras (no https, nos intentamos concetar a una http, sin la s), por tanto la solución es sencilla, en el manifest, dentro de application se coloca la siguiente línea.

```
android:usesCleartextTraffic="true"
```

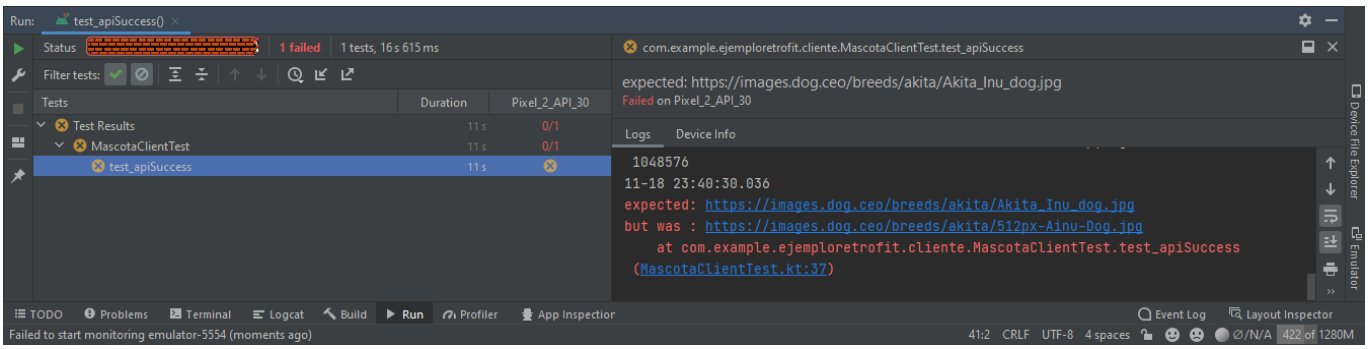
Con eso permitimos conexión a direcciones http y por ende podemos probar la aplicación, esta vez si dando un resultado exitoso para la prueba (lo esperado y lo recibido es en efecto lo mismo).



Ahora por probar la misma prueba, esperando un resultado erróneo (por tanto acá prueba fallida = éxito para nosotros), cambiamos lo esperado en el código (sólo como práctica) de modo que lo que nos entrega en posición 0, sea distinto a el link de la imagen que nos va a entregar.

```
@Test
fun test_apiSuccess() {

    val call =
MascotaClient.getCliente("http://localhost:8080/test/").getMascotas("akita")
    var mascota = call.execute().body()
    Truth.assertThat(mascota!!.message[0].toString()).isEqualTo("
https://images.dog.ceo/breeds/akita/512px-Ainu-Dog.jpg")

}
```

Cómo resultado la prueba falla y por tanto esto está correcto.



Hasta acá mi evidencia/resumen de hoy sobre el tema, muchas gracias.

Leonardo Rodenas Escobar :face_with_thermometer: