Programación Multimedia y Dispositivos Móviles

UD 13. HTTPS y servicios web

Javier Carrasco

Curso 2024 / 2025

HTTPS y servicios web

13. HTTPS y servicios web

Hay que ser consciente de que los dispositivos móviles se encuentran, la mayor parte del tiempo, conectados a Internet. Esto ofrece la posibilidad de acceder a información que no se encuentre de manera local al dispositivo, y trabajar con ella.

En este capítulo se tratará el uso del protocolo HTTPS, para evitar problemas en Android se evitará el HTTP, y la utilización de servicios web, concretamente los basados en REST.

Como se va a trabajar con Internet, lo primero que deberá hacerse es añadir el permiso para su uso en el *manifest*.

```
<uses-permission android:name="android.permission.INTERNET" />
```

También, algo muy recomendable es comprobar el estado de la conexión para saber si se tiene acceso o no (recuerda el capítulo 9). Por lo que se añadirá el siguiente permiso al *manifest*.

```
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />
```

Y se puede crear una clase con métodos de ayuda que contenga la siguiente función para comprobar el estado de la conexión y el tipo de la misma, vista anteriormente.

```
fun checkConnection(context: Context): Boolean {
    val cm = context.getSystemService(CONNECTIVITY_SERVICE) as ConnectivityManager
    val networkInfo = cm.activeNetwork

    if (networkInfo != null) {
        val activeNetwork = cm.getNetworkCapabilities(networkInfo)
        if (activeNetwork != null)
            return when {
                activeNetwork.hasTransport(NetworkCapabilities.TRANSPORT_WIFI) -> true
                actNetwork.hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR) -> true
                activeNetwork.hasTransport(NetworkCapabilities.TRANSPORT_ETHERNET) -> true
                else -> false
            }
    }
    return false
}
```

Ahora que ya está preparado lo básico, se comenzará por los más sencillo, descargar el contenido de una página web. Para ello se hará uso de la librería **OkHttp**.

13.1. OkHttp

La librería **OkHttp**[1] permite, básicamente, lanzar peticiones *https* y recibir la respuesta, pudiendo utilizarse de manera síncrona o asíncrona. El primer paso será añadir las siguientes dependencias al *gradle*.

1 OkHttp (https://square.github.io/okhttp/)

```
implementation 'com.squareup.okhttp3:okhttp:4.9.2'
implementation 'androidx.activity:activity-ktx:1.7.2'
```

La primera es la propia de OkHttp, la segunda permitirá realizar la tarea haciendo uso de *lifecycleScope* para crear corrutinas, ya se ha utilizado anteriormente.

Aprovechando el fichero *Utils.kt*, donde ya está el método para comprobar la conexión, se creará el siguiente método encargado de realizar la petición mediante *OkHttp*.

```
suspend fun downloadWebPage(url: String): String {
    return withContext(Dispatchers.IO) {
        try {
            val request = Request.Builder() // import okhttp3.Request
                .url(url)
                .build()
            val response = OkHttpClient().newCall(request).execute()
            response.body!!.string()
        } catch (e: IOException) {
            "ERROR: ${e.message}"
        }
    }
}
```

Este método se encarga de montar la petición a la URL que se pase por parámetro. Se hace de forma asíncrona, quedando a la espera hasta recibir la respuesta. Ésta se devolverá como un *String*, que es el tipo que se recibe. El método *onCreate* se encargará de lanzar la petición de la siguiente forma.

```
override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        if (checkConnection(this)) {
            var pageContent = ""
            lifecycleScope.launch {
                pageContent = downloadWebPage("https://www.javiercarrasco.es/")
                println(pageContent)
                binding.textView.text = pageContent
            }
        } else binding.textView.text = getString(R.string.NoConnection)
    }
```

De esta forma, haciendo uso de *lifecycleScope* para crear una corrutina se lanza la petición del método *suspend*.

Ahora bien, si prefieres no hacer uso de corrutinas en la UI, puedes optar por hacer uso de *ViewModel*. Para este caso no sería necesario crear un *factory*, ya que no es necesario pasar ningún parámetro.

```
class MainViewModel : ViewModel() {
    private var _pageContent: MutableLiveData<String> = MutableLiveData()
    val pageContent: MutableLiveData<String>
        get() = _pageContent

    init {
        viewModelScope.launch(Dispatchers.Main) {
            _pageContent.value = downloadWebPage("https://www.javiercarrasco.es/")
        }
    }
}
```

Se añade un constructor inicial (*init*) para que se lance la petición nada más crearse. Observa que se utilizará como tipo de datos para el contenido de la página el tipo *MutableLiveData*. Al utilizar este tipo de datos, es necesario hacer uso del método *value* para asignar los valores y, *pageContent* ahora es observable. El método *onCreate* de la clase principal quedará ahora así.

```
private val vm: MainViewModel by viewModels()

override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        if (checkConnection(this)) {
            vm.pageContent.observe(this) {
                binding.textView.text = it
            }
        } else binding.textView.text = getString(R.string.NoConnection)
    }
```

De esta forma, solo hay que observar la variable *pageContent*, de esta forma, cada vez que cambie el contenido se actualizará el *TextView*.

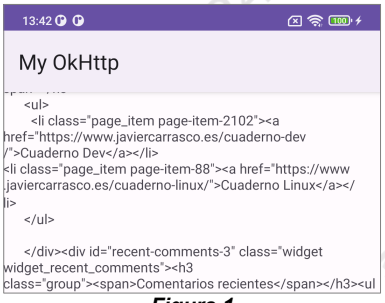

Figura 1

Ahora bien, cuando se utiliza este sistema de observación, es muy recomendable dejar de observar cuando se salga de la actividad que lo está haciendo, más bien, la actividad que contiene el componente que está observando.

Para evitar problemas con los observadores, una forma de solucionar esto sería crear un observador.

```
private val observer = Observer<String> {
    binding.textView.text = it
}
```

Cambiará ligeramente la forma de establecer el observador.

```
vm.pageContent.observe(this, observer)
```

Para eliminar el observador, bastará con sobrecargar el método *onDestroy()* y añadir la siguiente línea.

```
vm.pageContent.removeObserver(observer)
```

Con esta primera versión, puedes descargar el contenido de una página web, pero también se puede utilizar para descargar información en un JSON. Fíjate en esta segunda versión con *OkHttp*.

13.1.1. OkHttp y GSON

Para esta segunda versión, se realizará una petición contra una URL que devolverá información en formato JSON. Para poder *parsear* la información se utilizará la siguiente librería de GSON utilizada por Google. Añade, además de las dos añadidas anteriormente, la siguiente librería.

```
implementation 'com.google.code.gson:gson:2.9.0'
```

Ahora, al cambiar la URL que se pasa al método *downloadWebPage()* por la siguiente URL, *https://www.javiercarrasco.es/api/words/read*, verás que la información devuelta cambia, obteniéndose un JSON que habrá que *parsear* para poder interpretarlo.

```
data class Words (
    val idPalabra: String,
    val palabra: String,
    val definicion: String
){
    override fun toString(): String {
        return "$idPalabra - $palabra: ${
            if (definicion.length > 30)
                definicion.subSequence(0, 15)
            else definicion
        }...\n"
    }
}
```

El método de la clase principal se adaptará al uso de tipos *Flow*, como se verá a continuación en el *ViewModel*. Observa que se utiliza *lifecycleScope* y *repeatOnLifecycle* como se vio anteriormente, además de *collect* para obtener la información.

```
if (checkConnection(this)) {
    lifecycleScope.launch {
        repeatOnLifecycle(Lifecycle.State.STARTED) {
            vm.wordsList.collect {
                it.forEach {
                    binding.textView.append(it.toString())
                }
            }
        }
    }
} else binding.textView.text = getString(R.string.NoConnection)
```

La mayor modificación se encuentra en el *ViewModel*, donde se ha cambiado el tipo de la variable que recoge los datos, se utiliza *backing* para la variable y se aplica *Gson* para *parsear* la información recibida a través de la case *Words* creada.

```
class MainViewModel : ViewModel() {
    private var _wordsList: MutableStateFlow<List<Words>> = MutableStateFlow(emptyList())
    val wordsList: Flow<List<Words>>
        get() = _wordsList.asStateFlow()

    init {
        getWords()
    }

    fun getWords() {
        viewModelScope.launch(Dispatchers.Main) {
            val pageContent = downloadWebPage(
                "https://www.javiercarrasco.es/api/words/read"
            )
            if (!pageContent.isBlank()) {
                val gson = GsonBuilder().create()
                val wordsList = gson.fromJson(
                    pageContent,
                    Array<Words>::class.java).toList()

                _wordsList.value = wordsList
            }
        }
    }
}
```

Se ha creado el método *getWords()* para poder utilizarlo en cualquier momento para una actualización. Fíjate como en la instanciación de la variable *wordsList* se realiza la conversión de *pageContent*, que es un *String*, a una lista en base de la clase *Words*. La conversión puede hacerse a una lista, un mapa, etc, según sea necesarios.

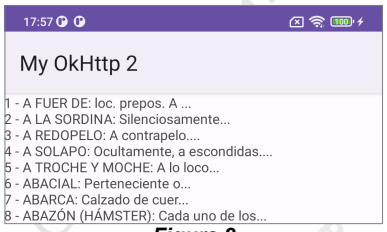

Figura 2

Si pruebas este código, verás que se muestra un listado de palabras, junto a su definición, en el *TextView* de la actividad.

Si en lugar de mostrar la información en un *TextView*, que cómo puedes ver, queda algo feo, se puede mostrar en un *RecyclerView*, por ejemplo. En esta tercera versión, se mostrará en un *Recycler*, pero haciendo uso de un *layout* de Android.

El primer paso será modificar la actividad principal para que en vez de un *TextView*, haya un *RecyclerView*. Hecho esto, se creará el siguiente adaptador que extenderá de ListAdapter, visto anteriormente, y que es mucho más sencillo de hacer y mantener. Se mostrará en cada elemento el número identificador de la palabra, y la palabra. Para ver la definición se pulsará sobre el elemento y se mostrará un diálogo con la definición completa.

```
class WordsRecyclerAdapter(private val onClickWords: (Words) -> Unit) :
    ListAdapter<Words, WordsViewHolder>(WordsDiffCallback()) {

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): WordsViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(android.R.layout.simple_list_item_2, parent, false)
        return WordsViewHolder(view)
    }

    override fun onBindViewHolder(holder: WordsViewHolder, position: Int) {
        val currentWord = getItem(position)
        holder.bind(currentWord, onClickWords)
    }
}

class WordsDiffCallback : DiffUtil.ItemCallback<Words>() {
    override fun areItemsTheSame(oldItem: Words, newItem: Words): Boolean {
        return oldItem.idPalabra == newItem.idPalabra
    }
    override fun areContentsTheSame(oldItem: Words, newItem: Words): Boolean {
        return oldItem == newItem
    }
}

class WordsViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    private val wordItemView: TextView = view.findViewById(android.R.id.text1)
    private val wordItemView2: TextView = view.findViewById(android.R.id.text2)

    fun bind(word: Words, onClickWords: (Words) -> Unit) {
        wordItemView.text = word.idPalabra
        wordItemView2.text = word.palabra
        itemView.setOnClickListener { onClickWords(word) }
    }
}
```

En *WordsViewHolder* se utiliza *findViewByID* para localizar los elementos que componen la lista, no puede hacerse *binding*. Ahora, en la case principal se creará el adaptador que se utilizará para controlar la pulsación sobre cada ítem.

```
private val adapter: WordsRecyclerAdapter by lazy {
    WordsRecyclerAdapter(onClickWords = { word ->
        MaterialAlertDialogBuilder(this).apply {
            setTitle(word.palabra)
            setMessage(word.definicion)
            setPositiveButton(getString(android.R.string.ok)) { dialog, _ ->
                dialog.dismiss()
            }
        }.show()
    })
}
```

En este caso se ha utilizado una inicialización ***lazy***, esto hará que el adaptador se cree únicamente cuando sea necesario, delegando la ejecución al momento necesario. Ha diferencia de *lateinit*, *lazy* se utiliza con variable inmutables. Ahora, el método *onCreate()* de la actividad principal quedará como se muestra.

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.recyclerWords.adapter = adapter

    if (checkConnection(this)) {
        lifecycleScope.launch {
            repeatOnLifecycle(Lifecycle.State.STARTED) {
                vm.wordsList.collect {
                    adapter.submitList(it)
                }
            }
        }
    } else {
        binding.recyclerWords.visibility = View.GONE
        val aviso = TextView(this)
        aviso.text = getString(R.string.NoConnection)
        binding.root.addView(aviso)
    }
}
```

La forma de obtener la información no cambia con respecto a la versión anterior, se sigue utilizando *lifecycleScope* y *repeatOnLifecycle*, junto con *collect* para recoger la información. La diferencia radica en que ahora, se hace un ***submit*** con la lista sobre el adaptador para cargar el *RecyclerView*.

Ahora, el resultado es algo más fácil de manejar por el usuario, y mostrará la definición de la palabra en un cuadro de diálogo. Evidentemente, puede mejorarse, como personalizar el *layout* del *Recycler*.

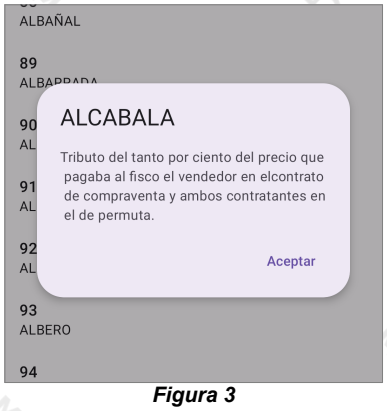

Figura 3

13.2. Retrofit

Retrofit[2] es otra librería que permite implementar peticiones HTTP en Android de una manera relativamente sencilla. Esta biblioteca permite el consumo de APIs, además se combinará con corrutinas y con el uso de *Flows*. Este último no es necesario, puede hacerse con *LiveData*, por ejemplo, puedes verlo en el proyecto MyRetrofit[3] que encontrarás en GitHub.

2 Retrofit (https://square.github.io/retrofit/)
3 MyRetrofit (https://github.com/KotlinStuff/DMKotlin-v2023/tree/main/CAP-13/MyRetrofit) sin *Flows*

En este proyecto se utilizará la API de pruebas *Fake Store API* (https://fakestoreapi.com/docs), revisa la documentación para ver los *endpoints* que ofrece y todas las opciones de las que dispone. La idea es simular una tienda virtual en la que se podrán listar los productos, todos o por categorías, ver su detalle y *logearse*.

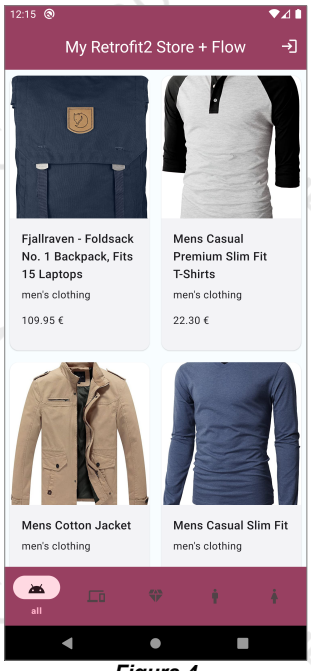

Figura 4

Para este proyecto se aplicará nuevamente *Clean Architecture* y el patrón MVVM, recuerda que puedes consultar el uso de esta estructura en el capítulo anterior.

El primer paso será añadir las siguientes dependencias al proyecto en el *Gradle (Module :app)*.

```
// Permite el uso de viewModelScope.
implementation 'androidx.activity:activity-ktx:1.7.2'

// Retrofit2
implementation "com.squareup.retrofit2:retrofit:2.9.0"
implementation "com.squareup.retrofit2:converter-gson:2.9.0"
```

Además, se añadirán las siguientes dos dependencias para facilitar tareas, como *Glide* para las imágenes, muy importantes para el objetivo final de la aplicación, y *SwipeRefresh*, para recargar los datos cuando sean necesarios.

```
// Glide
implementation 'com.github.bumptech.glide:glide:4.14.2'

// SwipeRefreshLayout
implementation "androidx.swiperefreshlayout:swiperefreshlayout:1.1.0"
```

Antes de continuar con el proyecto, un repaso sobre las API REST.

13.2.1. ¿Qué es una API REST?

Una API REST es un servicio que facilita una serie de mecanismos para obtener información de un cliente externo, generalmente una base de datos que nutra la aplicación. Las peticiones que pueden hacerse a una API REST son los siguientes:

- **GET**, devuelven información, puede pasársele parámetros a través de la URL, pero es poco segura.
- **POST**, similar a GET, pero más segura, los parámetros no se pasan en la URL.
- **PUT**, se utilizará para crear registros en la base de datos.
- **DELETE**, permite eliminar registros de la base de datos.

La información devuelva por una API REST estará por lo general en formato JSON. Observa una respuesta JSON de la API que se utilizará en el ejemplo.

```
//output
[
    {
        id:1,
        title:'...',
        price:'...',
        category:'...',
        description:'...',
        image:'...'
    },
    /*...*/
    {
        id:30,
        title:'...',
        price:'...',
        category:'...',
        description:'...',
        image:'...'
    }
]
```

Como norma general, para poder modificar el contenido mediante una API REST, será necesario algún tipo de autenticación, aunque muchas son utilizadas como consulta (GET) y no requieren de este sistema de seguridad.

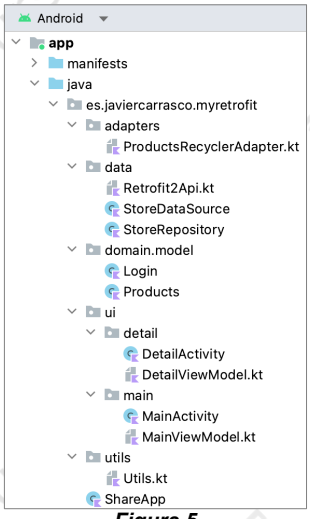

Figura 5

La API utilizada para este ejemplo permite emular acciones de borrado, modificación e inserción sin necesidad de autenticación, pero no se verá reflejado en la base de datos. En su lugar, la API devolverá la respuesta JSON, si todo ha ido bien, del elemento borrado, modificado o insertado.

De vuelta al proyecto, se reutilizará la clase `Utils.kt` que contiene el método ***checkConnection()*** utilizado en ejemplos anteriores, necesario para comprobar el estado de la conexión y saber cuando se pueden lanzar peticiones.

13.2.2. El Modelo

Se comenzará definiendo el modelo de la aplicación, según los datos recibidos desde la API REST. Se creará la clase `Products` para recoger la información de los productos, y la clase `Login` para realizar la identificación mediante *token*.

```
import android.os.Parcelable
import com.google.gson.annotations.SerializedName
import kotlinx.parcelize.Parcelize

@Parcelize
data class Login(@SerializedName("token") val token: String = "") : Parcelable
```

```
import android.os.Parcelable
import com.google.gson.annotations.SerializedName
import kotlinx.parcelize.Parcelize

@Parcelize
data class Products(
    @SerializedName("id") val id: Int = -1,
    @SerializedName("title") val title: String = "",
    @SerializedName("price") val price: Double = 0.0,
    @SerializedName("category") val category: String = "",
    @SerializedName("description") val description: String = "",
    @SerializedName("image") val image: String = ""
): Parcelable
```

13.2.3. Capa de datos

En esta parte se añadirá la configuración de **Retrofit2**, dónde se establece la URL de la API REST a consumir. Dentro del *package* ***data*** se creará la clase `Retrofit2Api.kt` con el siguiente código.

```
class Retrofit2Api {
    companion object {
        fun getRetrofit(): FakestoreApi {
            return Retrofit.Builder()
                .baseUrl("https://fakestoreapi.com/")
                .addConverterFactory(GsonConverterFactory.create())
                .build()
                .create(FakestoreApi::class.java)
        }
    }
}
```

Dentro de esta misma clase se creará la *interface* con los métodos que permitirán el acceso a la información de la API. Observa que desde la configuración se hace referencia a ella en el *create*.

```
interface FakestoreApi {
  @GET("products")
  suspend fun getProducts(): List<Products>

  @GET("products/categories")
  suspend fun getCategories(): List<String>

  @GET("products/category/{category}")
  suspend fun getProductsByCategory(@Path("category") category: String): List<Products>

  @GET("products/{id}")
  suspend fun getProductById(@Path("id") id: Int): Products?

  @FormUrlEncoded
  @POST("auth/login")
```

```
    suspend fun login(@Field("username") user: String, @Field("password") pass: String): Login
}
```

Como puedes ver, la *interface* contiene todos los métodos aplicando *suspend*, esto permitirá hacer uso de corrutinas. Las etiquetas identifican el tipo de operación (`@GET` , `@POST` , `@PUT` y `@DELETE`). Si observas el método *login*, verás que se pasa mediante POST, por lo que hay que utilizar la etiqueta `@FormUrlEncoded` para indicar el uso de campos en la cabecera, campos que se indican mediante el uso de `@Field` .

La cadena de texto que se indica como parámetro hace referencia a los *endpoints* de la API. Fíjate que la parte que está entre llaves se utiliza para sustituirla por el parámetro marcado como `@Path` , dato que se pasará en la llamada.

Una vez se dispone de la interface que da acceso a la fuente de datos, en este caso la API, ya pueden crearse los ***DataSources***. Recuerda que son una abstracción de la fuente de datos y, se crearán en base a la interface. Esto es muy útil cuando se utilicen más de una fuente de datos, por ejemplo una API y una base de datos *Room*. Siguiendo estas indicaciones, dentro del *package* ***data*** se creará la clase `StoreDataSource.kt` que contendrá el siguiente código.

```
class StoreDataSource {
    private val retrofit2Api = Retrofit2Api.getRetrofit()

    fun getProducts() = flow {
        emit(retrofit2Api.getProducts())
    }

    fun getCategories() = flow {
        emit(retrofit2Api.getCategories())
    }

    fun getProductsByCategory(category: String) = flow {
        emit(retrofit2Api.getProductsByCategory(category))
    }

    suspend fun getProductById(id: Int): Products? {
        return retrofit2Api.getProductById(id)
    }

    suspend fun login(user: String, pass: String): Login {
        return retrofit2Api.login(user, pass)
    }
}
```

Observa que se está haciendo uso de ***Flow***, lo que permite hacer uso de corrutinas y añadir programación reactiva. Básicamente reciben la información en vivo, pudiendo devolver más de un valor, además de trabajar de forma asíncrona.

Fíjate que los métodos que usan *Flow* ya no utilizan *suspend*, haciendo uso de ***emit***, lo que notificará la recepción de datos.

El siguiente paso será añadir el ***Repository***, muy similar al *DataSource* visto, pero es el punto en el que se decidirá que *DataSource* utilizar en caso de tener varias opciones como fuente de datos. En este caso, dentro del *package* ***data*** se creará la clase `StoreRepository.kt` que contendrá el siguiente código.

```kotlin
class StoreRepository(val dataSource: StoreDataSource) {

    fun fetchProducts(): Flow<List<Products>> {
        return dataSource.getProducts()
    }

    fun fetchCategories(): Flow<List<String>> {
        return dataSource.getCategories()
    }

    fun fetchProductsByCategory(category: String): Flow<List<Products>> {
        return dataSource.getProductsByCategory(category)
    }

    suspend fun fetchProductById(id: Int): Products? {
        return dataSource.getProductById(id)
    }

    suspend fun login(user: String, pass: String): Login {
        return dataSource.login(user, pass)
    }
}
```

Llegados a este punto, ya se dispondría de un acceso “total” a la API REST. Para este ejemplo se omitirá la capa de dominio, por no considerarse necesaria, pasando directamente a la capa de presentación.

13.2.4. Capa presentación

Llega el momento de recopilar la información de la API REST y presentarla al usuario. Se comenzará con la actividad principal, mostrando un listado completo de todos los elementos disponibles.

UI Principal

Además de los productos disponibles, se recogerán todas las categorías que tiene la API REST para montar una *BottomAppBar*.

Figura 6

```kotlin
class MainViewModel(private val storeRepository: StoreRepository) : ViewModel() {
    private var _products: Flow<List<Products>> = storeRepository.fetchProducts()
    val products: Flow<List<Products>>
        get() = _products

    var categories: Flow<List<String>> = storeRepository.fetchCategories()  ...
```

Además, se añadirán los siguientes métodos, el primero se utilizará para forzar la actualización de los productos, el segundo permitirá el filtrado por categorías.

```
    ...
    fun fetchProducts() {
        _products = storeRepository.fetchProducts()
    }

    fun fetchProductsByCategory(category: String) {
        _products = storeRepository.fetchProductsByCategory(category)
    }
}
```

Por último, como debe pasarse un repositorio por parámetro, deberá crearse un *Factory*.

```
@Suppress("UNCHECKED_CAST")
class MainViewModelFactory(private val storeRepository: StoreRepository) :
    ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return MainViewModel(storeRepository) as T
    }
}
```

Una vez definido el *ViewModel* para la actividad principal, deberá asociarse a esta, pasándole el repositorio que se utilizará.

```
class MainActivity : AppCompatActivity() {
    private lateinit var binding: ActivityMainBinding

    private val vm: MainViewModel by viewModels {
        val storeDataSource = StoreDataSource()
        val storeRepository = StoreRepository(storeDataSource)
        MainViewModelFactory(storeRepository)
    }
    ...
```

Seguidamente se crea el adaptador para el *RecyclerView* que mostrará los elementos obtenidos de la API REST.

```
private val adapter: ProductsRecyclerAdapter by lazy {
    ProductsRecyclerAdapter(
        onClickProduct = { product ->
            DetailActivity.navigateToDetail(this@MainActivity, prodId = product.id)
        },
        onClickDelete = { product ->
            Toast.makeText(
                this@MainActivity,
                "Deleted ${product.title}", Toast.LENGTH_SHORT
            ).show()
        }
```

```
    )
}
```

Observa que el adaptador se crea haciendo uso de la propiedad ***lazy***, de esta forma, el código especificado en la lambda se ejecutará cuando sea realmente necesario. Sin entrar en detalle, la clase `ProductsRecyclerAdapter` extiende de la clase *ListAdapter*, visto en el capítulo 11. Ya en el método *onCreate* se configura el listado y se cargan los datos.

```
binding.recyclerProducts.layoutManager = GridLayoutManager(this, 2)
binding.recyclerProducts.adapter = adapter

if (checkConnection(this)) {
    collectCategories()
    collectProducts()
}
```

Los métodos encargados de la recolección de la información, mediante el uso de *Flows*, quedarán de la siguiente forma. Para las categorías:

```
private fun collectCategories() {
    lifecycleScope.launch {
        vm.categories
            .catch {
                println("ERROR: $it")
            }
            .collect { it: List<String>
                binding.bottomNavigation.menu.apply {
                    this.findItem(R.id.itemBottom2).title = it[0]
                    this.findItem(R.id.itemBottom3).title = it[1]
                    this.findItem(R.id.itemBottom4).title = it[2]
                    this.findItem(R.id.itemBottom5).title = it[3]
                }
            }
    }
}
```

Se accede la propiedad *categories* del *ViewModel* desde la corrutina, además, se deberá controlar la posibilidad de error con la sección *catch*, y si todo va bien, en la recolección se monta la *BottomAppBar*. La recolección de los productos será como se muestra:

```
private fun collectProducts() {
    lifecycleScope.launch {
        vm.products
            .catch {
                println("ERROR: $it")
            }
            .collect { it: List<Products>
                adapter.submitList(it)
            }
    }
}
```

Al igual que para las categorías, se controla un posible error en la sección *catch*. Para este caso, como la variable *products* es un *Flow* de una lista de productos, el *collect* obtiene la lista que actualizará el adaptador del *RecyclerView*.

Para filtrar por categorías se hace uso de una *BottomAppBar*, en este caso, se añade la funcionalidad en el método *onStart* para asegurarse que la vista está creada. También se añade al método la funcionalidad del *Swipe Refresh* tal como se vio en el captítulo 5.

```
override fun onStart() {
    super.onStart()

    binding.swipeRefresh.setOnRefreshListener {
        adapter.submitList(emptyList())

        if (checkConnection(this)) {
            collectCategories()
            collectProducts()
        }
        binding.swipeRefresh.isRefreshing = false
    }

    binding.bottomNavigation.setOnItemSelectedListener { item ->
        adapter.submitList(emptyList())
        when (item.itemId) {
            R.id.itemBottom1 -> { // All
                vm.fetchProducts()
            }
            else -> vm.fetchProductsByCategory(item.title.toString())
        }
        collectProducts()
        true
    }
}
```

UI Detalle

La actividad detalle se encargará de mostrar el elemento que se seleccione en el listado. Para ello, deberá pasarse al detalle el identificador del producto en cuestión. En primer lugar, se creará el *ViewModel* para la actividad.

```
class DetailViewModel(private val storeRepository: StoreRepository, private val prodId: Int)
 : ViewModel() {
    private val _state = MutableStateFlow(Products())
    val state: StateFlow<Products> = _state.asStateFlow()

    init {
        viewModelScope.launch {
            val product = storeRepository.fetchProductById(prodId)
            if (product != null)
                _state.value = product
```

```
        }
    }
}
```

En este caso, únicamente se utiliza el constructor *init* para obtener directamente el elemento en la creación del objeto y se actualiza la variable *_state* con el producto obtenido. Fíjate que en este caso, se utiliza las clases *MutableStateFlow* y *StateFlow*[4], son un flujo observable que contiene los estados que emiten las actualizaciones del estado actual y nuevas para los recolectores. En el constructor se utiliza la propiedad *value* para actualizar esos estados.

Por último, al igual que en la actividad anterior, deberá pasarse el repositorio, pero también el identificador del producto como parámetros, deberá crearse nuevamente un *Factory* para este *ViewModel*.

```
@Suppress("UNCHECKED_CAST")
class DetailViewModelFactory(
    private val storeRepository: StoreRepository,
    private val prodId: Int
) : ViewModelProvider.Factory {

    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return DetailViewModel(storeRepository, prodId) as T
    }
}
```

Una vez definido el *ViewModel* para la actividad detalle, deberá asociarse a ésta, pasándole el repositorio que se utilizará y el identificador del producto.

```
class DetailActivity : AppCompatActivity() {
    private lateinit var binding: ActivityDetailBinding

    private val vm: DetailViewModel by viewModels {
        val storeDataSource = StoreDataSource()
        val storeRepository = StoreRepository(storeDataSource)
        val prodId = intent.getIntExtra(PRODUCT_ID, -1)
        DetailViewModelFactory(storeRepository, prodId)
    }
    ...
```

A diferencia de la creación de la variable *vm* de la actividad principal, se añade el identificador, que se obtiene del *intent*. A continuación, se crea un *companion object* para crear el método encargado de lanzar la actividad.

```
companion object {
    const val PRODUCT_ID = "PRODUCT_ID"

    fun navigateToDetail(activity: AppCompatActivity, prodId: Int = -1) {
        val intent = Intent(activity, DetailActivity::class.java).apply {
            putExtra(PRODUCT_ID, prodId)
```

4 *StateFlow* (https://developer.android.com/kotlin/flow/stateflow-and-sharedflow)

```
        }
        activity.startActivity(
            intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TOP),
            ActivityOptions.makeSceneTransitionAnimation(activity).toBundle()
        )
    }
}
```

Ya sólo quedaría recolectar el dato en el método *onCreate* de la clase. Ten en cuenta que no es necesario obtener el identificador del producto aquí, ya que se obtiene en la declaración de la variable *vm*.

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    lifecycleScope.launch {
        repeatOnLifecycle(Lifecycle.State.STARTED) {
            vm.state.collect { it: Products
                binding.title.text = it.title
                binding.description.text = it.description
                binding.category.text = it.category
                binding.price.text = getString(R.string.txtPrecio, it.price)

                binding.imageView.contentDescription = it.title
                Glide.with(this@DetailActivity)
                    .load(it.image)
                    .into(binding.imageView)
            }
        }
    }
}
```

En este punto ya se recupera una lista de categorías, una lista de productos y, con la actividad detalle se obtiene un único elemento, dónde se muestra la descripción del producto mostrado.

Figura 7

Obtener un token

El siguiente paso será añadir el token de autenticación que proporciona esta API REST. Desde la *AppBar* se añadirá una opción de menú que mostrará un diálogo para solicitar al usuario los datos de autenticación.

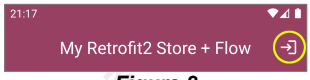

Figura 8

Si analizas la documentación de la *Fake Store API* (https://fakestoreapi.com/docs), puedes obtener datos de autenticación, por ejemplo, *username* “johnd” y *password* “m38rmF$”.

La idea es validarse contra la base de datos a través del *endpoint* que facilita la API, para ello se ha creado un cuadro de diálogo personalizado en el que se solicitarán los datos de autenticación, también se controla que no se dejen los campos vacíos.

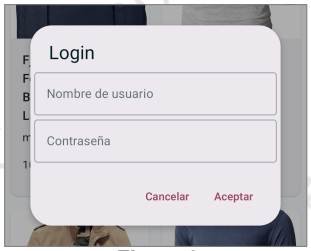

Figura 9

Para almacenar el *token* recibido se utilizará la clase *SharePreferences* vista en el capítulo 8, con una ligera modificación, en este caso la clase Preferences se creará como una *inner class* para que todo quede en la misma clase *ShareApp*.

```
class ShareApp : Application() {
    private val PREFS_NAME = "es.javiercarrasco.myretrofit"
    private val SHARED_NAME = "token"

    companion object {
        lateinit var preferences: Preferences
            private set
    }

    override fun onCreate() {
        super.onCreate()
        preferences = Preferences()
    }

    inner class Preferences() {
        val prefs = applicationContext.getSharedPreferences(
            PREFS_NAME,
            MODE_PRIVATE
        )

        var token: String
            get() = prefs.getString(SHARED_NAME, Login().token) ?: Login().token
            set(value) = prefs.edit().putString(SHARED_NAME, value).apply()
    }
}
```

Recuerda indicar en el *manifest* la existencia de esta clase para poder almacenar las preferencias.

```
...
<application
    android:name=".ShareApp"
    ...
```

Al tratarse de una opción de menú deberá crearse el recurso XML necesario para mostrarlo. Además, como se hace uso de una *MaterialToolbar*, no añadida a través del estilo, en el método *onCreate* se añade una llamada al método ***createMenu()*** , justo después del *setContentView*, que se detallará a continuación.

```
@SuppressLint("NotifyDataSetChanged")
private fun createMenu() {
    // Se añade el menú a la Toolbar persosnalizada.
    binding.mToolbar.inflateMenu(R.menu.action_bar_menu)

    stateLogin()

    binding.mToolbar.setOnMenuItemClickListener {
        when (it.itemId) {
            R.id.item_login -> {
                if (it.title == getString(R.string.txtLogin)) {
                    showLoginDialog()
                } else {
                    ShareApp.preferences.token = ""
                    stateLogin()
                    binding.recyclerProducts.adapter?.notifyDataSetChanged()

                    Toast.makeText(
                        this@MainActivity, "You are logged out", Toast.LENGTH_SHORT
                    ).show()
                }
                true
            }

            else -> false
        }
    }
}
```

Como puedes observar, este método se encarga de añadir el menú a la *MaterialToolbar*, seguidamente se comprueba el estado del *login* (se detalla a continuación) y se añade el *listener* para controlar la pulsación sobre los elementos del menú. Concretamente, se comprueba el texto que contiene la opción del menú, si el texto es *"login"* se muestra el cuadro de diálogo, en caso contrario, ya se está *logeado*, por lo que se hará un *logout*.

El método *stateLogin* se utiliza simplemente para comprobar si hay, o no, *token* para mostrar un icono en la opción de menú u otro, además de cambiar el texto que se utiliza para la comprobación al pulsar la opción del menú.

```
private fun stateLogin() {
    if (ShareApp.preferences.token.isBlank()) {
        binding.mToolbar.menu.findItem(R.id.item_login)
            .setIcon(R.drawable.ic_login).title = getString(R.string.txtLogin)
    } else {
        binding.mToolbar.menu.findItem(R.id.item_login)
            .setIcon(R.drawable.ic_logout).title = getString(R.string.txtLogout)
    }
}
```

El método ***showLoginDialog*** se encargará de mostrar el cuadro de diálogo para la autenticación, el control de campos vacíos y solicitará el *token* con los datos facilitados por el usuario.

```
@SuppressLint("NotifyDataSetChanged")
private fun showLoginDialog() {
    val bindCustomDialog = LoginLayoutBinding.inflate(layoutInflater)

    val dialogLogin = MaterialAlertDialogBuilder(this).apply {
        setView(bindCustomDialog.root)
        setTitle(R.string.txtLogin)
        setPositiveButton(android.R.string.ok, null)
        setNegativeButton(android.R.string.cancel) { dialog, _ ->
            dialog.cancel()
        }
    }.create()

    dialogLogin.setOnShowListener {
        dialogLogin.getButton(AlertDialog.BUTTON_POSITIVE).setOnClickListener {
            val user = bindCustomDialog.etUsername.text.toString().trim()
            val pass = bindCustomDialog.etPassword.text.toString().trim()

            if (user.isNotEmpty() && pass.isNotEmpty()) {
                vm.getLogin(this, user, pass)
                lifecycleScope.launch {
                    vm.token.collect { login ->
                        ShareApp.preferences.token = login.token
                        stateLogin()
                        if (login.token != "") {
                            binding.recyclerProducts.adapter?.notifyDataSetChanged()
                            Toast.makeText(
                                this@MainActivity,
                                "You are logged\n\nToken ${login.token}",
                                Toast.LENGTH_SHORT
                            ).show()
                        }
                    }
                }
                dialogLogin.dismiss()
            } else {
                bindCustomDialog.etUsername.error = getString(R.string.txtErrorDialog)
                bindCustomDialog.etPassword.error = getString(R.string.txtErrorDialog)
            }
        }
    }

    dialogLogin.show()
}
```

Se muestra el *token* en un *Toast*, evidentemente esto no debe hacerse.

Tras introducir los datos de *logeo* correctos, se mostrará el *toast* con el *token* recibido. Además de mostrar el *token*, observa que se refrescará el listado, apareciendo así una papelera en cada uno de los *items* mostrados. Esto se consigue a través de la actualización del *adapter* y forzar así la comprobación del *login*.

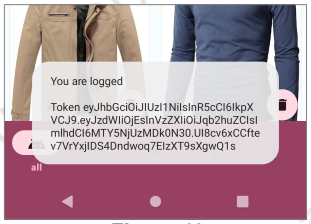

Figura 10

En la clase del *adapter*, se deberá controlar la visibilidad de la papelera haciendo uso de la siguiente comprobación.

```
if (ShareApp.preferences.token.isNotBlank())
    bind.btnDelete.visibility = View.VISIBLE
else bind.btnDelete.visibility = View.GONE
```

Eliminar un elemento

Si recuerdas la declaración del *adapter*, el segundo parámetro es *onClickDelete*, dónde se muestra un *toast*.

```
...
onClickDelete = { product ->
    Toast.makeText(
        this@MainActivity,
        "Deleted ${product.title}", Toast.LENGTH_SHORT
    ).show()
}
...
```

Recuerda que la API que se está utilizando para las pruebas no permite el borrado, pero si la petición se hace correctamente, ésta devolverá el *item* “eliminado” como respuesta correcta.

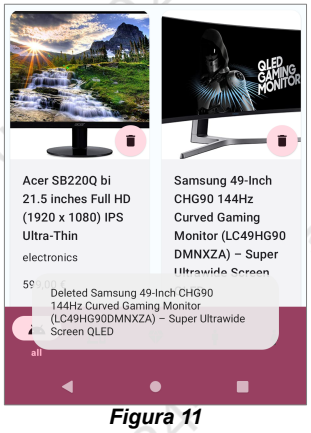

Figura 11