**Sesión 3-03 Clase del 5 de diciembre**

En esta sesión realizaremos apps para consumir datos de un servicio web API Rest y procesar su información. Usaremos servicios que son gratuitos, de libre acceso y que no requieren autenticación (API Key, OAuth).

Además, usaremos una librería GSon para parsear documentos JSON y conexiones http.

Cuando hablamos de un servicio web no nos estamos refiriendo al servicio “normal” que nos proporciona un servidor web, en concreto a que nos devuelva un documento HTML o de otro tipo representable en un cliente navegador como una página web.

Con servicio web nos referimos a un servicio que nos ofrece un servidor web para proporcionarnos información en un documento con un standard de representación estandarizado (XML, JSON, otros) sobre los contenidos que se manejan en el sitio web que corresponda o, en su caso, permitirnos hacer operaciones de actualización de esos contenidos.

Cada web service dispone en servidor de varios métodos que responden a peticiones de los clientes. A estos métodos se los denomina en conjunto API Web Service.

Muchas plataformas web publican su API, es decir dan información a los desarrolladores de cómo acceder a sus métodos y cuál es el formato de los documentos de respuesta, por ejemplo, Youtube, Netflix, Facebook, Google Maps, Aemet, etc. Otras plataformas u otros sitios web tienen su API aunque no está publicada, lo que no tiene porqué significar que los desarrolladores no la puedan utilizar.

Todo servicio web:

• Debe utilizar un protocolo de transporte para la comunicación de las aplicaciones cliente con el servicio web, normalmente HTTP o HTTPS.

• Para desarrollarlos se ha utilizado una tecnología. Fundamentalmente, se usan dos tecnologías, SOAP y REST, aunque hay otras.

• Intercambia los mensajes (peticiones desde el cliente y respuestas al cliente) con uno o varios formatos de representación, principalmente JSON y XML.

Se habla de consumir un servicio web para indicar que una aplicación cliente obtiene datos de un servicio web y/o sube datos a un servicio web.

1.- Vamos a consumir la API **herokuapp (**<https://extinct-api.herokuapp.com/>). Esta API tiene un endpoint de tipo GET:

**/api/v1/animal/**

Cada vez que se hace una petición GET a ese endpoint, la API nos devuelve un JSON con información sobre un animal extinguido. El animal se obtiene de forma aleatoria de entre todos los registrados en la base de datos en la que consulta la API.

Esto es una captura de lo que devuelve la API en un momento a la solicitud GET:

<https://extinct-api.herokuapp.com/api/v1/animal/>

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Es importante entender la estructura que nos está devolviendo.

* Nos devuelve un documento **JSON** (todo el documento) que tiene un atributo status con valores de tipo String y un array **data.**
* Aunque **data** es un array siempre (para este endpoint), contiene un único documento con varios atributos de tipo String (**binomialName, commonName**, etc.)
* El atributo **imageSrc** contiene la url de un archivo de imagen con la foto del animal extinguido que se puede descargar.

2.- Descarga y abre el proyecto **Sesion3\_03\_Base**. La pantalla tiene los componentes necesarios para representar la imagen de un animal, el nombre común, la localización y el último registro del animal.

3.- Como vamos a manejar datos relativos a animales extintos obtenidos de la consulta de la API, debemos crear un modelo de datos para mapear los datos recibidos en formato JSON. Podemos crear en un package **model¸** los **data class** para representar los datos de un animal y para representar la información recibida en la respuesta de consulta a la API.

No se necesita representar en el modelo todos los datos que se reciben, pero si se necesita representar correctamente la estructura de toda la información recibida.

Podemos representar el **data class** de un animal, de esta forma:

data class Animal(  
 val binomialName: String="",  
 val commonName: String="",  
 val imageSrc: String="",  
 val lastRecord: String="",  
 val location: String="",  
 val shortDesc: String

)

Intencionadamente, he dejado sin representar en el modelo el atributo **wikiLink** (para verificar que se pueden mapear los datos del JSON sin representar todos sus atributos en el modelo). El data class de la respuesta, se podría representar de esta forma:

data class AnimalResponse(  
 val status: String,  
 val data: List<Animal>  
)

Aquí **status** podría no usars**e**. La lista de animales es necesaria ya que en la respuesta de la API viene un array (aunque contenga un solo animal).

Para crear los **data class** del modelo podrías usar el plugin **Json to Kotlin Class.**

* Instala el plugin.
* Elimina las clases anteriores.
* En package **model**, selecciona **New Kotlin Data Class From Json**. Pega la respuesta JSON de la api para obtener un animal de forma aleatoria.
* Finaliza y verás que se han creado los **data class** completos.

4.- Ahora modificamos lo necesario en el código para que se use **ViewModel**

Inicialmente, **sin consultar la API**, se cargarán los datos de un animal en un objeto **AnimalResponse.** Cuando se pulse el botón SIGUIENTE, se mostrarán los datos el animal:

Incluir librería para **viewmodel**:

*implementation*(*libs*.*androidx*.*runtime*.*livedata*)

Crear una clase para el **ViewModel**.

class AnimalViewModel(): ViewModel() { }

Declarar en la clase **AnimalViewModel** un **LiveData** para el animal que se vaya obteniendo de la API:

class AnimalViewModel: ViewModel() {  
 //LiveData para estado actual de Animal  
 private val \_animal=MutableLiveData<Animal?>(  
 Animal(imageSrc = "https://upload.wikimedia.org/wikipedia/commons/thumb/f/f7/Perru\_disparue.jpg/220px-Perru\_disparue.jpg",  
 ))  
 val animal:LiveData<Animal?> = \_animal

Como se puede ver el **live data** del animal tiene la información de un animal inicial (sólo la url de la imagen).

Añadir una función que carga los datos de un nuevo animal. Después programaremos que esa función la ejecute el botón SIGUIENTE. Esa función después la modificaremos para que cargue un animal obtenido de consumir la API.

fun getAnimalRandom(){  
 val newAnimal=Animal(  
 commonName = "Club-tailed glyptodont",  
 location = "South American Pampas",  
 imageSrc = "https://upload.wikimedia.org/wikipedia/commons/thumb/8/8b/Doedicurus.png/220px-Doedicurus.png" ,  
 lastRecord = "4765-4445 BCE"  
 )  
 \_animal.*value*=newAnimal  
}

Ahora, **HomeView** tiene que recibir el **ViewModel**, usar una variable **observador** del **livedata animal del ViewModel**. En los componentes **GlideImage y Text** asignaremos los valores del observador animal.

fun HomeView(viewModel:AnimalViewModel) {  
 val animal: Animal? by viewModel.animal.observeAsState()

GlideImage(  
 model = animal?.imageSrc,  
 contentDescription = "Imagen del animal extinto",  
 modifier = Modifier  
 .*fillMaxWidth*()  
 .*weight*(0.5f)  
)

Text(  
 text = "Último registro: ${animal?.lastRecord?:""}",  
 modifier = Modifier.*fillMaxWidth*(),  
 fontSize = 24.*sp*)

Programar el botón SIGUIENTE para que, al clicar, llame a la función **getAnimalRandom** del **ViewModel.**

// Botón "Siguiente"  
Button(  
 onClick = **{** viewModel.getAnimalRandom()**}**,

Por último, en **MainActivity** hay que construir el **ViewModel** y pasárselo a **HomeView.**

val viewModel by *viewModels*<AnimalViewModel>()  
HomeView(viewModel,service)

5.- **Utilización de Retrofit**. **Retrofit** es una librería para realizar procesos de consumo de una API web service.

* Incluimos dependencias de retrofit y del conversor JSON-Objetos en retrofit:

// Retrofit  
*implementation*("com.squareup.retrofit2:retrofit:2.9.0")  
*implementation*("com.squareup.retrofit2:converter-gson:2.9.0")

* Creamos una interface que contenga la estructura de las peticiones al servidor (a la API). Lo creamos en el **package data**. Declaramos una sola función anotada con GET para el endpoint animal. La función es de tipo **suspend** ya que deberá incluir una corrutina, La función debe devolver un objeto **AnimalResponse** (es lo que nos devuelve la API, aunque en formato JSON)

interface RetrofitService{  
 @GET("animal/")   
 suspend fun getRandomAnimal(): AnimalResponse  
}

Aunque no lo tiene la API, si tuviera un **endpoint** que recibe un id de un animal y devuelve el animal, declararíamos la otra función así:

interface RetrofitService {  
 @GET("animal/")  
 suspend fun getRandomAnimal(): AnimalResponse  
 // Otros métodos de la API si los necesitas  
 @GET("animal/{id}")  
 suspend fun getAnimalById(@Path("id") id: Int): AnimalResponse  
}

* A continuación de la declaración del interface, lo implementamos mediante un objeto:

object RetrofitServiceFactory{  
 fun makeRetrofitService():RetrofitService{  
 return Retrofit.Builder()  
 //url base de la api  
 .baseUrl("https://extinct-api.herokuapp.com/api/v1/")  
 //usar convertidor de JSON  
 .addConverterFactory(GsonConverterFactory.create())  
 //construir  
 .build().create(RetrofitService::class.*java*)  
 }  
}

* Ya está casi todo preparado. En el ViewModel construimos el servicio Retrofit. En la función **getRandomAnimal()** sustituimos código para que use el servicio Retrofit para obtener un **AnimalResponse** con la función del interface.

//Instancia del servicio retrofit  
val service= RetrofitServiceFactory.makeRetrofitService()

fun getAnimalRandom(){  
 //corrutina para obtener un animal random  
 *viewModelScope*.*launch* **{** try {  
 //llamada al servicio para obtener un animal random  
 val randomAnimalResponse = service.getRandomAnimal()  
 \_animal.*value* = randomAnimalResponse.data.get(0)  
 } catch (e: Exception) {  
 e.printStackTrace()  
 }  
 **}**}

Ya puede probar.

6.- Habrás apreciado un problema de funcionalidad. Al iniciarse la App no carga un animal inicial, sólo lo hace cuando clicamos SIGUIENTE. Hay una solución fácil. En un ViewModel podemos usar la función **init** (función que se ejecuta siempre que se construye un objeto de una clase que la tanga definida.

En nuestro caso la programamos para que también obtenga un Animal de la API.

//Inicializar con un animal random  
init {  
 getAnimalRandom()  
}