PROGRAMACIÓN MULTIMEDIA Y DISPOSITIVOS MÓVILES

TEMA 9:

PETICIONES HTTP: RETROFIT

"Una manera de hacer Europa". Cofinanciación a cargo del Programa Operativo del FSE 2014-2020 para Extremadura gastos de Ciclos Formativos de Grados Medio y Superior.

ÍNDICE

PMDM

PETICIONES HTTP: RETROFIT

1. API REST

"**Una manera de hacer Europa**". Cofinanciación a cargo del Programa Operativo del FSE 2014-2020 para Extremadura gastos de Ciclos Formativos de Grados Medio y Superior.

1.- API Rest

- Una *API REST* es una interfaz de comunicación que usa el protocolo HTTP para obtener datos o ejecutar operaciones sobre dichos datos.
- Se puede decir, por tanto, que una API REST es un intermediario entre una base de datos y un cliente (Aplicación Android, navegador web, etc).

1.- API Rest


ESQUEMA COMUNICACIÓN
CLIENTE-SERVIDOR
Servidor
Cliente Web
Cliente Android
API REST
BBDD

1.- API Rest

- Estas son algunas de las características de las APIs Rest:
  - Debe usar una arquitectura cliente-servidor.
  - Debe usar las operaciones estándar de los *verbos HTTP*.
  - Debe hacer uso de la *URL* como identificador único de los recursos.
  - Desacoplamiento del cliente-servidor: En el diseño de la API REST, las aplicaciones de cliente y servidor deben ser completamente independientes entre sí.

1.- API Rest

- Las API REST se comunican a través de solicitudes HTTP para ejecutar operaciones de base de datos estándar.
- Las operaciones básicas son crear (Create), leer (Read), modificar (Update) y borrar (Delete) y son también conocidas como *CRUD*.

PMDM

PETICIONES HTTP: RETROFIT

2. Peticiones HTTP

2.- Peticiones HTTP

- Las peticiones HTTP son mensajes enviados por un cliente para iniciar una acción en el servidor.
- Piensa en una petición HTTP como si tu navegador se conectara al servidor y le pidiera un recurso específico o le enviara datos.
- Hay distintos tipos de peticiones HTTP. Estos tipos dependen del *verbo* utilizado.

2.- Peticiones HTTP

- Los verbos más comunes son:
  - GET
  - POST
  - PUT
  - DELETE

2.- Peticiones HTTP

GET

- Una petición GET solicita al servidor una información o recurso concreto.
- Una de las principales características de una petición GET es que no debe causar efectos secundarios en el servidor: no deben producir nuevos registros, ni modificar los ya existentes.

2.- Peticiones HTTP

POST

- Este tipo de peticiones sirven para crear recursos nuevos en la base de datos.
- Cada llamada con POST debería, por tanto, producir un nuevo recurso.

2.- Peticiones HTTP

PUT

- Se utiliza para modificar un recurso existente en la base de datos.

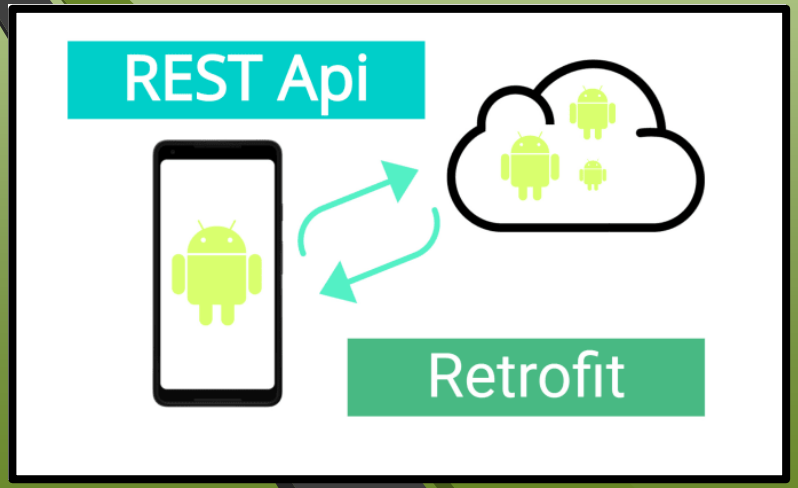
2.- Peticiones HTTP

DELETE

- Es el verbo que utilizamos para eliminar registros.
- Con este verbo, podríamos eliminar un único registro de la base de datos o, incluso, una tabla completa.

PMDM

PETICIONES HTTP: RETROFIT



3. Retrofit

3.- Retrofit

- *Retrofit* es una librería de Android que facilita la implementación de peticiones HTTP en nuestras aplicaciones.
- Provee de funcionalidades como la de añadir cabeceras personalizadas a las peticiones (añadiendo un token a la cabecera de las peticiones, por ejemplo).
- Retrofit permite incorporar "convertidores" que "castean" el resultado de las peticiones HTTP de manera automática.

3.- Retrofit

- Pasos para utilizar Retrofit en nuestra aplicación debermos realizar los siguientes pasos:

1. Configuración previa: Añadir dependencias al gradle.
2. Configuración previa: Añadir permisos en AndroidManifest.xml.
3. Crear los DATA CLASSES.
4. Crear la interfaz APIService.
5. Crear el ClienteRetrofit.
6. Realizar peticiones asíncronas en segundo plano (Corrutinas).

3.- Retrofit

Configuración previa

- Para poder utilizar la librería Retrofit en nuestra aplicación móvil, debemos añadir las siguientes dependencias al fichero *build.gradle.kts* de *Module:app* y sincronizar.

```
//Librería para Retrofit 2
implementation("com.squareup.retrofit2:retrofit:2.9.0")
//Librería GSON para el tratamiento y conversión de datos JSON
implementation("com.squareup.retrofit2:converter-gson:2.9.0")
//Librería OkHttp para simplificar la construcción de peticiones HTTP
implementation("com.squareup.okhttp3:okhttp:4.12.0")
//Librería para utilizar corrutinas en Kotlin (peticiones HTTP en segundo plano)
implementation("org.jetbrains.kotlinx:kotlinx-coroutines-android:1.7.3")
```

3.- Retrofit

Configuración previa

- Después añadiremos permisos de INTERNET en el *AndroidManifest.xml*.

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

- Además, debemos añadir dentro de la etiqueta <*application*> la propiedad *usesCleartextTraffic* a true, para habilitar el tráfico de texto simple.

```
<application
    android:usesCleartextTraffic="true"
```

3.- Retrofit

DATA CLASSES

- Estos DATA CLASSES será la convención que utilizaremos para traernos objetos de la base de datos y, con el apoyo de las anotaciones de Gson, facilitaremos el mapeo y manejo de variables dentro de nuestra aplicación.

3.- Retrofit

Data Class Kotlin Dirección

```kotlin
/**
 * Ejemplo de DATA CLASS para el objeto Direccion
 */
data class Direccion(

    @SerializedName("id")
    val id: Int,
    @SerializedName("localidad")
    val localidad: String,
    @SerializedName("provincia")
    val provincia: String,
    @SerializedName("direccion")
    val direccion: String,
    @SerializedName("codigo_postal")
    val codigo_postal: String

)
```

JSON (respuesta del servidor a la petición GET al endpoint "/dirección")

```json
{
  "id": 1,
  "localidad": "Plasencia",
  "provincia": "Cáceres",
  "direccion": "Av. de la Constitución, 12",
  "codigo_postal": "10600"
}
```

3.- Retrofit

DATA CLASSES

- Podemos usar el plugin **JSON to Kotlin Class** para crear los Data Classes que necesitamos en nuestras aplicaciones, el cual nos facilita la construcción de estas clases a partir del JSON recibido en las peticiones HTTP.

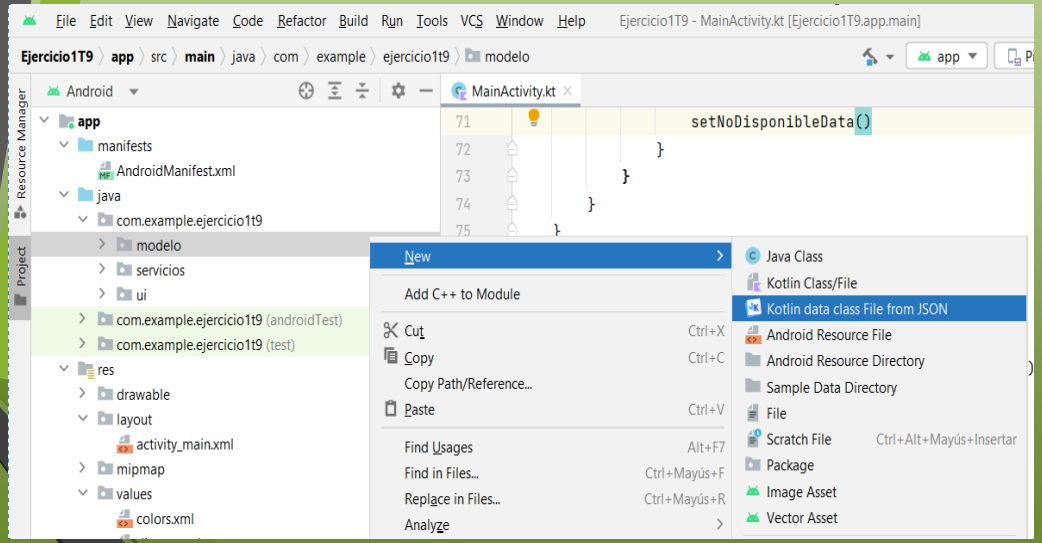
1. Instalar el plugin JSON to Kotlin Class

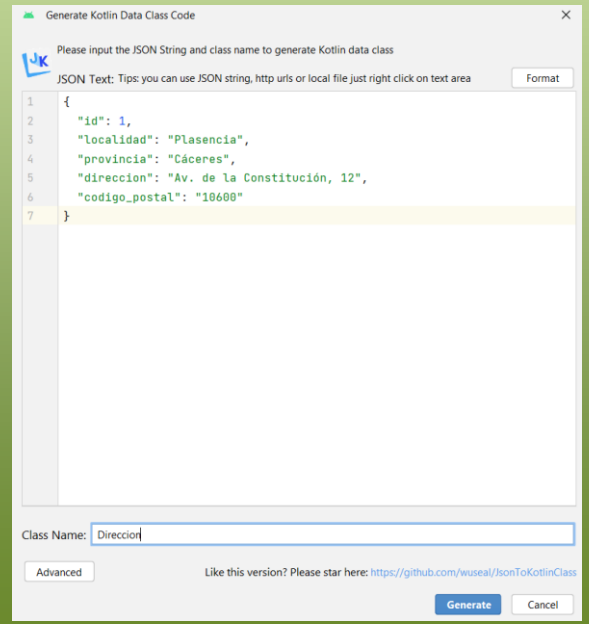
- Para poder utilizar el plugin debemos instalarlo previamente en Android Studio, desde Tools -> SDK Manager -> Plugins.

3.- Retrofit

DATA CLASSES

2.Utilización del plugin JSON to Kotlin Class





3.- Retrofit

Interfaz APIService

- En ella definiremos los métodos abstractos (*endpoints*).
- Cada endpoint va a representar una ruta específica de nuestra API.
- Para realizar los principales tipos de peticiones HTTP estándar como GET, POST, PUT y DELETE, Retrofit nos provee anotaciones para cada petición HTTP: @GET, @POST, @PUT y @DELETE.

3.- Retrofit

Interfaz APIService: Anotación

- Ya sabemos que necesitamos anotaciones para que Retrofit sepa el tipo de petición que queremos realizar.
- Adicionalmente, necesitaremos especificar la ruta relativa para cada uno de nuestros endpoints (por ejemplo, "players/<ID>") y no la ruta absoluta ("https://www.balldontlie.io/api/v1/players/<ID>").

3.- Retrofit

Interfaz APIService: Anotación

- Por tanto, si tuviésemos que definir un endpoint para obtener todos los jugadores ("players") de una API Rest, deberíamos especificar tanto la anotación como la ruta relativa antes del método abstracto.

```
interface APIService {
  @GET("players")
  (...) getAllPlayers() (...)
}
```

3.- Retrofit

Interfaz APIService: Retorno

- En cuanto al retorno de los métodos, Retrofit recoge todas las respuestas del servidor en objetos ***Response***. Es imprescindible declarar correctamente el objeto o conjunto de objetos que devuelve cada petición HTTP (por ejemplo, la respuesta de la petición HTTP al endpoint que devuelve todos los jugadores de una API Rest se debe mapear en un *List<Player>*).

```
@GET("players")
fun getAllPlayers(): Response <List<Player>>
```

3.- Retrofit

Interfaz APIService: Parámetros

- **@Body**. Utilizamos esta anotación en un parámetro cuando deseamos insertar o modificar un objeto en la base de datos (peticiones POST o PUT).

@POST ("players")

*fun insertPlayer (**@Body player: Player**): Response<Player>*

3.- Retrofit

Interfaz APIService: Parámetros

- **@Headers**. Se utiliza principalmente para agregar información de metadatos en una solicitud como, por ejemplo, para añadir un token a la cabecera de una petición.

  @Headers("Authorization")

  fun getAllPlayers (): Response<List<Player>>

- Si la petición es para insertar o modificar un objeto en la base de datos (peticiones POST o PUT) utilizamos esta anotación para indicar que el contenido de la petición HTTP lo enviamos en formato JSON.

  @Headers("Content-Type: application/json")

  @POST ("players")

  fun insertPlayer (@Body player: Player): Response<Player>

3.- Retrofit

Interfaz APIService: Parámetros

- **@Path**. Se utiliza como parámetro de consulta para poder filtrar los datos de una petición al servidor.

  *@GET("players**/{id}**")*

  *fun getPlayerPorId(**@Path("id") playerId: Int**): Response<Player>*

- **@Query**. Al igual que @Path, sirve para reemplazar con un valor parte de la ruta de la petición.

UNION EUROPEA
Fondo Social Europeo
"Una manera de hacer Europa"
"Una manera de hacer Europa". Cofinanciación a cargo del Programa Operativo del FSE 2014-2020 para Extremadura gastos de Ciclos Formativos de Grados Medio y Superior.

3.- Retrofit

ClienteRetrofit

- Será la encargada de instanciar un objeto Retrofit, haciendo uso del patrón *Singleton* para garantizar una única instancia. En esta clase se define la ruta base de la API (*BASE_URL*).

ClienteRetrofit.kt

```kotlin
package com.example.ejercicio1t9.servicios

import ...

object ClienteRetrofit {

    private val BASE_URL = "https://www.balldontlie.io/api/v1/"

    //Crea un objeto Retrofit que se utilizará para realizar peticiones a la API
    private fun getRetrofitInstance(): Retrofit {
        return Retrofit.Builder() Retrofit.Builder
            .baseUrl(BASE_URL) Retrofit.Builder
            .addConverterFactory(GsonConverterFactory.create())
            .build()
    }

    //Crea un objeto de la interfaz APIService (interfaz que contiene las diferentes peticiones a la API)
    //a partir del objeto Retrofit.
    //Se añade "by lazy" para que este objeto se cree únicamente cuando se invoque (solo se crea cuando se vaya a utilizar, si
    //se utiliza).
    val apiService: APIService by lazy {
        getRetrofitInstance().create(APIService::class.java)
    }

}
```

3.- Retrofit

Peticiones asíncronas en segundo plano (Corrutinas)

- Nos permiten ejecutar los endpoints sobre la API Rest.
- Es recomendable preguntar por el método ***isSuccessful*** para el objeto *Response*, garantizando así que la respuesta del servidor tiene un código entre 200 y 300 (respuesta ok).

3.- Retrofit

Peticiones asíncronas en segundo plano (Corrutinas)

```
//Lanza la corrutina en segundo plano
CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
    //Petición HTTP para obtener el jugador a través de su id
    val playerResponse: Response<Player> = apiService.getPlayer(playerId)
    //Tratamiento de los datos recuperados de la petición HTTP
    //El objeto Player que obtenemos de la petición HTTP debe ser nullable (puede ser que la petición
    //no nos devuelva ningún objeto con el id especificado).
    var player: Player? = null
    if(playerResponse.isSuccessful) {
        //Se obtiene el objeto con los datos del jugador del body del objeto Response
        player = playerResponse.body()
        //Rest of code
    } else if (playerResponse.code() == 404) {
        //Not found error
    } else if (playerResponse.code() == 500) {
        //Internal server error
    } else {
        //Other errors
    }
}
```

3.- Retrofit

Peticiones asíncronas en segundo plano (Corrutinas)

- El código que implementamos en una Corrutina se ejecuta en un hilo diferente al principal, hilo que se encarga de construir la interfaz de usuario (UIThread).
- Por tanto, toda la funcionalidad correspondiente al UIThread que se utilice dentro de una Corrutina se debe invocar en el hilo principal (por ejemplo, mostrar un Toast). Para ello utilizamos la función ***runOnUIThread***.

3.- Retrofit

Peticiones asíncronas en segundo plano (Corrutinas)

```
CoroutineScope(Dispatchers.IO).launch { this: CoroutineScope
    val playerResponse: Response<Player> = apiService.getPlayer(playerId)
    var player: Player? = null
    if(playerResponse.isSuccessful) {
        player = playerResponse.body()
        //Rest of code
    } else if (playerResponse.code() == 404) {
        showPlayerNotFoundError()
    } else if (playerResponse.code() == 500) {
        //Internal server error
    } else {
        //Other errors
    }
}
```

```
private fun showPlayerNotFoundError() {
    runOnUiThread {
        Toast.makeText(applicationContext, "No existe ningún jugador con el ID especificado", Toast.LENGTH_SHORT).show()
    }
}
```

3.- Retrofit

Peticiones asíncronas en segundo plano (Corrutinas)

- Además, toda función correspondiente a una petición HTTP que se invoque dentro de una Corrutina debe declararse con el modificador ***suspend***. Este modificador lo añadimos en la declaración de las funciones de la interfaz APIService. Ejemplo:

```
@GET("players")
suspend fun getAllPlayers(): Response <List<Player>>
```

PMDM

MODELO DE HILOS

4. Modelo-Vista-Controlador

4.- Modelo-Vista-Controlador

- El *Modelo-Vista-Controlador* es uno de los patrones de diseño más utilizados para estructurar un proyecto de desarrollo.
  - El **modelo** contiene la información con la que el sistema trabaja.
  - La **vista** presenta al usuario la información del modelo.
  - El **controlador** contiene la lógica de las aplicaciones. Qué hacer cuando se pulsa un botón, qué servicios contiene la aplicación, etc.

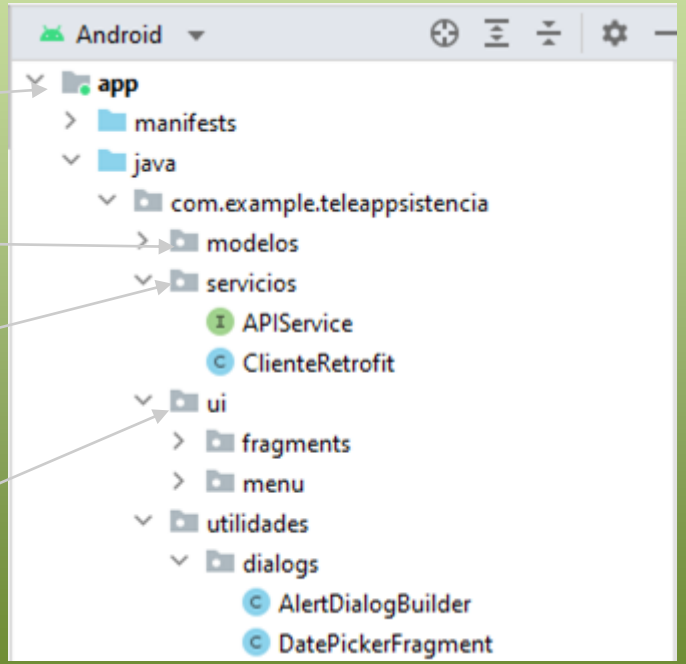
4.- Modelo-Vista-Controlador

Proyecto Android

Contiene todas las clases serializables para traer o enviar todos los datos a la API REST (**Modelo**)

Servicios. Contiene todo lo necesario para una correcta comunicación con la API Rest (**Controlador**)

Interfaz gráfica (**Vista**)



3.- Retrofit

Modelo

- **Paquete modelos**. Contiene todas las clases serializables para traer o enviar todos los datos a la API REST. Las clases serializables serán nuestros *DATA CLASSES*.

3.- Retrofit

Vista

- **Paquete UI (Interfaz gráfica)**. Paquete con todos los archivos relacionados con la interfaz gráfica.
  - Fragments.
  - Activities.

3.- Retrofit

Controlador

- **Paquete servicios**. Componente de la aplicación que puede realizar operaciones de larga ejecución en segundo plano y que no proporciona una interfaz de usuario.
  - APIService. Interface con todas las peticiones.
  - ClienteRetrofit. Crea una única instancia del cliente REST (de la APIService).

PMDM

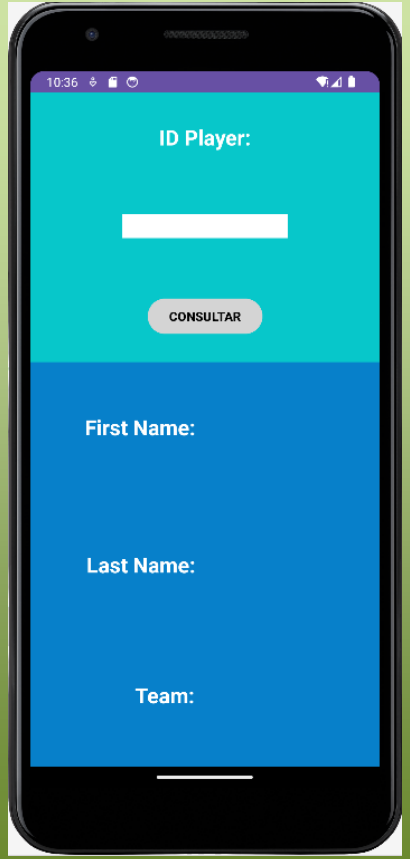
PETICIONES HTTP: RETROFIT

5. Ejercicios de consolidación

EJERCICIOS

- **Ejercicio 01:** La app estará dividida en un layout superior y otro layout inferior.
- Cuando el usuario pulse el botón *Consultar*, se hará una petición HTTP a la API Rest y cargará los datos del jugador en el layout inferior.
- Utiliza el MVC.

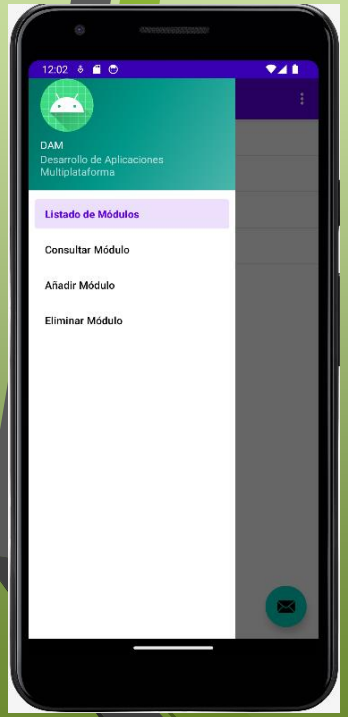
- API Rest: *https://www.balldontlie.io/home.html#introduction*

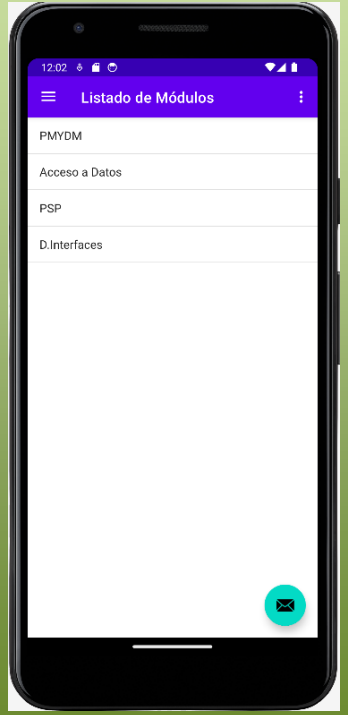


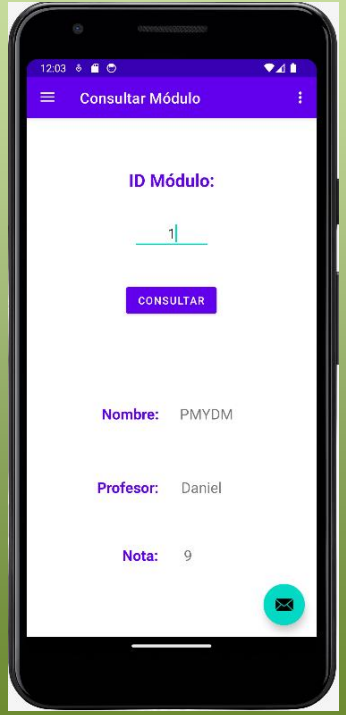
EJERCICIOS

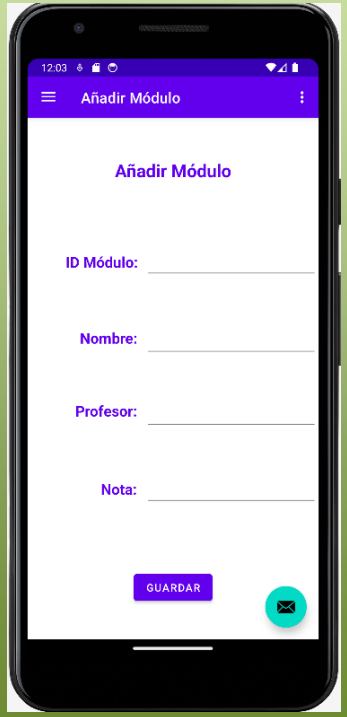
- **Ejercicio 02:** Crea una aplicación móvil para almacenar las calificaciones de los módulos del ciclo formativo de DAM.
- Para ello, deberás crear tu propia API Rest con JSON Server donde almacenarás la información del id del módulo, nombre del módulo, profesor que lo impartió y la nota que obtuviste.
- Tu aplicación debe tener una Navigation Drawer Activity con cuatro opciones de menú que permitan:
  - Consultar todos los módulos.
  - Consultar un módulo dado su id.
  - Añadir un nuevo módulo con todos sus datos.
  - Eliminar un módulo dado su id.

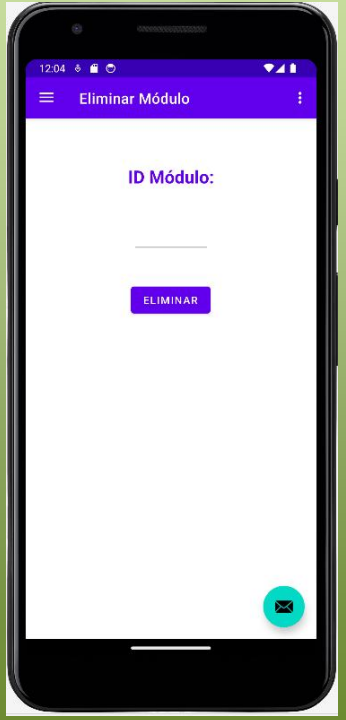
EJERCICIOS


12:02
DAM
Desarrollo de Aplicaciones Multiplataforma
Listado de Módulos
Consultar Módulo
Añadir Módulo
Eliminar Módulo


12:02
Listado de Módulos
PMYDM
Acceso a Datos
PSP
D.Interfaces


12:03
Consultar Módulo
ID Módulo:
1
CONSULTAR
Nombre: PMYDM
Profesor: Daniel
Nota: 9


12:03
Añadir Módulo
Añadir Módulo
ID Módulo:
Nombre:
Profesor:
Nota:
GUARDAR


12:04
Eliminar Módulo
ID Módulo:
ELIMINAR