Socket.io

El Cliente

Una vez completado el Servidor, vamos a montar un pequeño **Cliente** en Android Studio usando **Kotlin**. Recuerda que siempre es posible 'convertir' Kotlin a Java automáticamente, luego no debería haber mucho problema para hacerlo en Java.

Recordemos la lógica detrás de las operaciones que queremos hacer.

Login	
Nombre del evento:	"onLogin"
Parámetros enviados	userName
Parámetros devueltos:	Alumno
Propósito:	El servidor devuelve el alumno de Base de Datos cuyo userName coincide con el solicitado.

GetAll	
Nombre del evento:	"onGetAll"
Parámetros enviados	-
Parámetros devueltos:	Listado de Alumnos
Propósito:	El servidor devuelve todos los alumnos de la Base de Datos.

Logout	
Nombre del evento:	"onLogout"
Parámetros enviados	userName
Parámetros devueltos:	-
Propósito:	El servidor toma nota de que el cliente se ha deslogueado.

Al igual que con el **Servidor Java**, lo primero que hacemos es crear un proyecto Android en Kotlin. A continuación, configuramos el **Gradle**.

- CompileSdk = 35
- MinSDK = 29
- TargetSdk = 35

Ignora el warning indicando que hay problemas en el TargetSdk, o soluciónalo diciendo al Android Studio que sabes lo que haces (probablemente). Ahora, añadimos al Gradle las dependencias que necesitamos_

- Socket.io.client: versión 2.1.1
- Engine.io.client: versión 2.1.0

Gson: la última disponible Gradle:

```
// SOCKET.IO
implementation(libs.socket.io.client)
implementation(libs.engine.io.client)

// GSON
implementation(libs.gson)
```

Libs.versions.toml

```
socket-io-client = { module = "io.socket:socket.io-client", version = "2.1.1" }
engine-io-client = { module = "io.socket:engine.io-client", version = "2.1.0" }
gson = { module = "com.google.code.gson:gson", version.ref = "gson" }
```

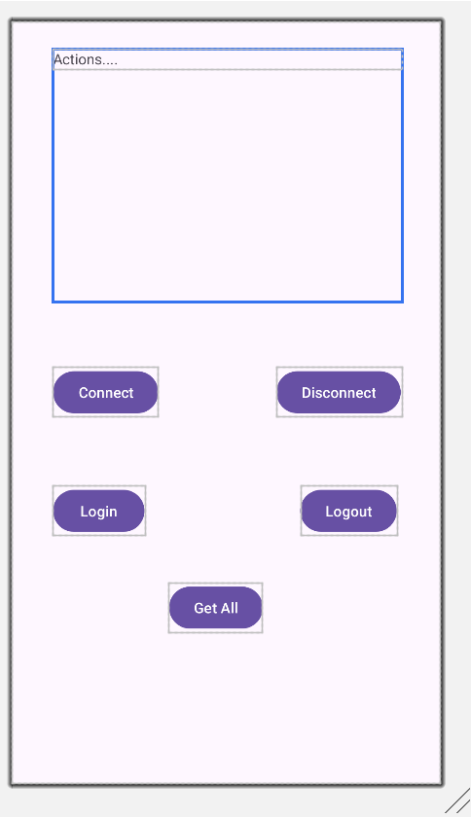
Por último, damos permisos de internet en el **Manifest**.

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

Y permitimos que los mensajes de texto vayan en claro por la red:

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

Con esto, ya tenemos el proyecto listo para montar la app.



El interfaz

Vamos a empezar por la ventana. Dado que es tan solo un ejemplo sobre la comunicación cliente-servidor, nuestra app va a limitarse a una serie de botones que lanzarán mensajes. Veremos las respuestas del Server porque aparecerán listadas tanto en un TextView como en las trazas de log.

Las funcionalidades de los botones son:

- Connect: Conecta el cliente con el Servidor
- Disconnect: Lo desconecta
- Login: Loguea al usuario en el Servidor.
- Logout: Lo desloguea
- GetAll: Solicita todos los alumnos de BBDD

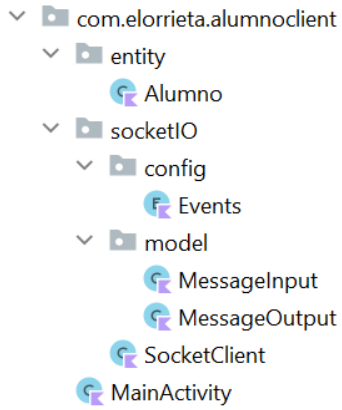
Varias cosas antes de seguir. **Primero**, en este ejemplo estamos haciendo el conectar/desconectar con sendos botones. Esto no tiene sentido. Tú tienes que ser el que decida cómo, cuándo y por qué un cliente se conecta al Servidor. Sí, es necesario que gestiones esto, pero para este ejemplo lo dejamos así. La app no va a funcionar ni a generar un error si no le das tú a conectar al comenzar a usarla, por mucho que le des a otros botones. Y si te funciona… es de casualidad.

Segundo. Te habrás dado cuenta de que **cada botón** se corresponde con uno de los **eventos del Server**. Los de conectar y desconectar son el onConnect () y onDisconnect (), mientras que los otros tres se corresponden con onLogin, onLogout y onGetAll.

El código

Generaremos la siguiente estructura de clases:

Si te fijas, son esencialmente las mismas clases que en el lado Servidor. De hecho, las clases **Alumno, MessageInput, MessageOutput** y **Events** son idénticas, no se ha cambiado una sola coma (salvo pasarlas a Kotlin).



SocketClient dispondrá de los eventos capturados por la app, y se encargará de procesar el envío y recepción de los mensajes. Por último, el **MainActivity** llamará al **SocketClient**

PASO 1 - Alumno y **Events** son idénticas a las del lado Server. **Alumno** se usará para convertir la información que llega

en formato JSON en los mensajes y manipularla más fácilmente, por ejemplo, mostrándola en un listado mediante un Adapter.

```
data class Alumno (private val id: Int, private val name: String, private val surname: String,
    private val pass: String, private val edad: Int)
```

Por su parte, Events contiene los nombres de los eventos.

```
enum class Events(val value: String) {
    ON_LOGIN ( value: "onLogin"),
    ON_GET_ALL ( value: "onGetAll"),
    ON_LOGOUT ( value: "onLogout"),
    ON_LOGIN_ANSWER ( value: "onLoginAnswer"),
    ON_GET_ALL_ANSWER ( value: "onGetAllAnswer");
}
```

Y **MessageInput** y **MessageOutput** son simplemente:

```
data class MessageInput (private val message: String)
```

PASO 2 – Vamos a añadir código al **SocketClient**. De la misma forma que hicimos con la clase **SocketIOModule** del Server. Comenzaremos por añadir al cliente todos los eventos que va a escuchar. Para eso, se añade el bloque **init** a la clase:

```
class SocketClient (private val activity: Activity) {

    // Server IP:Port
    private val ipPort = "http://10.0.2.2:3000"
    private val socket: Socket = IO.socket(ipPort)

    // For log purposes
    private var tag = "socket.io"

    // We add here ALL the events we are eager to LISTEN from the server
    init {
        // Event called when the socket connects
        socket.on(Socket.EVENT_CONNECT) {
            Log.d (tag,  msg: "Connected...")
        }

        // Event called when the socket disconnects
        socket.on(Socket.EVENT_DISCONNECT) {
            Log.d (tag,  msg: "Disconnected...")
        }
    }
}
```

Le hemos añadido por ahora dos eventos sencillos: **Connect** y **Disconnect**. Esto nos permitirá ver una traza de log en nuestro Android Studio cada vez que se conecte o desconecte el cliente al servidor. Vamos ahora con los **eventos** que hemos **diseñado nosotros** y que también añadimos al init:

```
// Event called when the socket gets an answer from a login attempt.
// We get the message and print it. Note: this event is called after
// We try to login
socket.on(Events.ON_LOGIN_ANSWER.value) { args ->

    // The response from the server is a JSON
    val response = args[0] as JSONObject

    // The answer should be like this:
    // {"id": 0, "name": "patata", "surname": "potato", "pass": "pass", "edad": 20}
    // We extract the field 'message'
    val message = response.getString( name: "message") as String

    // We parse the JSON into an Alumno because... ¯\_(ツ)_/¯
    var gson = Gson()
    var jsonObject = gson.fromJson(message, JsonObject::class.java)
    val id = jsonObject["id"].asInt
    val name = jsonObject["name"].asString
    val surname = jsonObject["surname"].asString
    val pass = jsonObject["pass"].asString
    val edad = jsonObject["edad"].asInt

    var alumno = Alumno (id, name, surname, pass, edad)

    // And... me list the Alumno in the list and in the Log
    activity.findViewById<TextView>(R.id.textView).append("\n" + alumno.toString())
    Log.d (tag,  msg: "The answer: $alumno")
}
```

Según nuestro propio diseño, el Server nos tiene que enviar como respuesta al login un objeto alumno. Nuestro Servidor es lo que hace: transforma el Alumno en JSON y lo envía dentro de un **MessageInput** al Cliente. Nosotros sólo tenemos que extraerlo, convertir el JSON de vuelta a un objeto Alumno, y utilizarlo.

En este caso hemos decidido parsear el JSON (que contiene un Alumno) a mano, aunque obviamente, podemos dejar a GSON que lo haga él sólo, como en hacemos con el **getAll**.

Por último, le añadimos la respuesta del **getAll** y hacemos lo mismo. En este caso

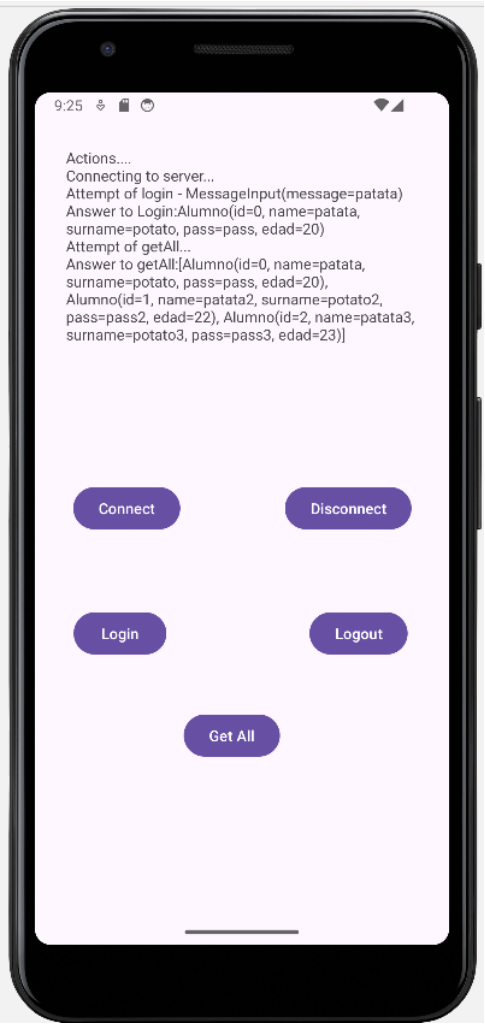
```
    socket.on(Events.ON_GET_ALL_ANSWER.value) { args ->

        // The response from the server is a JSON
        val response = args[0] as JSONObject

        // The answer should be like this:
        // [
        // {"id":0,"name":"patata","surname":"potato","pass":"pass","edad":20},
        // {"id":1,"name":"patata2","surname":"potato2","pass":"pass2","edad":22},
        // {"id":2,"name":"patata3","surname":"potato3","pass":"pass3","edad":23}
        // ]
        // We extract the field 'message'
        val message = response.getString( name: "message") as String

        // We parse the JSON. Note we use Alumno to parse the server response
        var gson = Gson()
        val itemType = object : TypeToken<List<Alumno>>() {}.type
        var list = gson.fromJson<List<Alumno>>(message, itemType)

        // The logging
        activity.findViewById<TextView>(R.id.textView).append("\nAnswer to getAll:$list")
        Log.d(tag,  msg: "Answer to getAll: $list")
    }
```

PASO 3 – Por ahora, nuestro Cliente simplemente escucha de forma pasiva al Server. Vamos a darle funcionalidad. Añadiremos fuera del init los siguientes métodos:

```
// This method is called when we want to establish a connection with the server
fun connect() {
    socket.connect()

    // Log traces
    activity.findViewById<TextView>(R.id.textView).append("\n" + "Connecting to server...")
    Log.d (tag,  msg: "Connecting to server...")
}

// This method is called when we want to disconnect from the server
fun disconnect() {
    socket.disconnect()

    // Log traces
    activity.findViewById<TextView>(R.id.textView).append("\n" + "Disconnecting from server...")
    Log.d (tag,  msg: "Disconnecting from server...")
}
```

Estos métodos conectarán y desconectarán nuestro cliente al servidor. Ahora, añadiremos también las tres operaciones básicas que hemos diseñado: **login**, **getAll** y **logout**.

```
// This method is called when we want to login. We get the userName,
// put in into an MessageOutput, and convert it into JSON to be sent
fun doLogin(userName: String) {
    val message = MessageInput(userName) // The server is expecting a MessageInput
    socket.emit(Events.ON_LOGIN.value, Gson().toJson(message))

    // Log traces
    activity.findViewById<TextView>(R.id.textView).append("\nAttempt of login - $message")
    Log.d (tag,  msg: "Attempt of login - $message")
}

// This method is called when we want to getAll the Alumno.
fun doGetAll() {
    socket.emit(Events.ON_GET_ALL.value)

    // Log traces
    activity.findViewById<TextView>(R.id.textView).append("\nAttempt of getAll...")
    Log.d (tag,  msg: "Attempt of getAll...")
}

// This method is called when we want to logout. We get the userName,
// put in into an MessageOutput, and convert it into JSON to be sent
fun doLogout(userName: String) {
    val message = MessageInput(userName) // The server is expecting a MessageInput
    socket.emit(Events.ON_LOGOUT.value, Gson().toJson(message))

    // Log traces
    activity.findViewById<TextView>(R.id.textView).append("\nAttempt of Logout - $message")
    Log.d (tag,  msg: "Attempt of logout - $message")
}
```



PASO 4 – Prueba la aplicación. Deberías de ver algo como en la imagen.

Cosas que recordar

Una serie de cosas sobre este ejemplo de Socket.io. Primero, no he capturado ninguna excepción ni en el Cliente ni en el Server. No lo he hecho por simplicidad. Eso deberías de programarlo tú.

Segundo. Si el Cliente no conecta con el Servidor, nadie te va a avisar del error. Lo mismo pasa si hay una excepción. La App va a quedarse 'tostada' esperando indefinidamente sin dar error. Lo conveniente es que tengas un Servidor y un Cliente con un montón de trazas de log y que hagas uso de paciencia.

Tercero. Antes de programar nada, haz lo que he hecho yo y diseña la comunicación y el paso de mensajes entre cliente y servidor. Improvisar es la receta para el desastre. Date cuenta de que hay código que deberás tener igual en ambas aplicaciones.

Cuarto: Aprende GitHub. En serio. Lo necesitas…