Consumo de API REST

Imágenes en Servidores (Parte II)

{desafío} latam_

Construir una aplicación Android que consume un servicio REST actualizando la interfaz de usuario, acorde al lenguaje Kotlin y a la librería Retrofit.

- Unidad 1:
  Acceso a datos en Android
- Unidad 2:
  Consumo de API REST
- Unidad 3:
  Testing
- Unidad 4:
  Distribución del aplicativo Android

{desafío} latam_

¿Qué aprenderás en esta sesión?

- *Authoken, jwt, token-bearer*
- *Añadir el authtoken a un header usando Retrofit*

¿Cuál es el rol del token en un proceso de autenticación?

{desafío} latam_

/* Authoken, jwt, token-bearer */

Authoken

- Authoken es un token de autenticación que se usa en Retrofit para Android para autenticar las solicitudes realizadas a una API. Retrofit permite a los desarrolladores agregar autenticación fácilmente a sus solicitudes de API al incluir una Autorización en los headers de solicitud.

- El servidor puede generar el token y enviarlo al cliente, que luego puede incluirlo en solicitudes posteriores a la API para demostrar que el cliente ha sido autenticado.

¿Cómo usar Autothoken en Retrofit?

En este ejemplo, el objeto Retrofit se crea con una URL base, una fábrica de convertidores Gson y una interfaz `YourApiService` personalizada que define los puntos finales para su API. Se pasa una autorización como parámetro al método `getData()` del servicio. El método devuelve un objeto Call que se pone en cola para realizar la solicitud. El método enqueue toma dos devoluciones de llamada, una para manejar una respuesta exitosa y otra para manejar errores.

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://your-api-base-url.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val service = retrofit.create(YourApiService::class.java)

val authToken = "YOUR_AUTH_TOKEN"

val request = service.getData(authToken)
request.enqueue(object : Callback<Data> {
    override fun onResponse(call: Call<Data>, response:
Response<Data>) {
        // do something with the response
    }
    override fun onFailure(call: Call<Data>, t: Throwable) {
        // handle the error
    }
})
```

¿Cómo usar Autothoken en Retrofit?

Deberá reemplazar `YOUR_AUTH_TOKEN` con su token real y ajustar la URL base, la interfaz del servicio y el nombre del método de acuerdo con su API.

Tenga en cuenta que la Autorización debe mantenerse privada y segura, y debe aprobarse con cada solicitud que requiera autenticación.

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://your-api-base-url.com")
    .addConverterFactory(GsonConverterFactory.create())
    .build()

val service = retrofit.create(YourApiService::class.java)

val authToken = "YOUR_AUTH_TOKEN"

val request = service.getData(authToken)
request.enqueue(object : Callback<Data> {
    override fun onResponse(call: Call<Data>, response:
Response<Data>) {
        // do something with the response
    }
    override fun onFailure(call: Call<Data>, t: Throwable) {
        // handle the error
    }
})
```

JWT

- JSON Web Token (JWT) es un estándar abierto (RFC 7519) que define una forma compacta y autónoma de transmitir información de forma segura entre las partes como un objeto JSON. Esta información se puede verificar y confiar porque está firmada digitalmente. JWT se usa a menudo para autenticar a los usuarios.

- JWT generalmente consta de tres partes: un encabezado, una carga útil y una firma. El encabezado y la carga útil son cadenas JSON codificadas en Base64Url que contienen información sobre el token, como su tipo y las notificaciones que contiene. La firma se usa para verificar que el remitente del JWT es quien dice ser y para garantizar que el mensaje no haya cambiado en el camino.

JWT

- La carga útil de un JWT suele contener notificaciones, que son declaraciones sobre una entidad (normalmente, el usuario) y metadatos adicionales. Las notificaciones se representan como un objeto JSON que contiene pares clave-valor. Algunos reclamos comunes incluyen nombre, correo electrónico y sub (asunto) que se refiere al usuario que se está autenticando.

- JWT se usa a menudo en combinación con los protocolos OAuth2 y OpenID Connect para proporcionar autenticación y autorización para las API.

- JWT no tiene estado y no requiere almacenar ninguna información en el lado del servidor, por eso es útil para el inicio de sesión único (SSO) y la autenticación sin estado.

Ejemplo JWT

```
import com.auth0.jwt.JWT
import com.auth0.jwt.algorithms.Algorithm

val secret = "yoursecretkey"
val algorithm = Algorithm.HMAC256(secret)
val claims = mapOf("name" to "John Doe", "email" to "johndoe@example.com")
val jwt = JWT.create().withClaims(claims).sign(algorithm)
```

(claims).sign(algorithm)

Este ejemplo usa la biblioteca com.auth0.jwt para crear un JWT. La variable secreta contiene la clave secreta utilizada para firmar el token. La variable del algoritmo se establece en HMAC256, que es un algoritmo de firma simétrica.
La variable de reclamaciones es un mapa de pares clave-valor que se incluirán en la carga útil del token. En este ejemplo, la carga útil contiene el nombre y el correo electrónico del usuario.

Ejemplo JWT

- El método `JWT.create()` crea un nuevo constructor JWT y el método `withClaims(claims)` se utiliza para establecer las reclamaciones del token. El método `sign(algorithm)` se usa luego para firmar el token usando el algoritmo especificado y la clave secreta.

- La variable jwt resultante contiene el JWT codificado, que se puede enviar al cliente y pasar al servidor para la autenticación.

- Necesitarás usar tu propia clave secreta y ajustar los reclamos de acuerdo con sus necesidades, también puede considerar otros algoritmos de firma como RSA o ECDSA.

- Tenga en cuenta que es importante mantener la clave secreta privada y segura, y debe intercambiarse de forma segura.

- Además, puede usar bibliotecas como `io.jsonwebtoken` para crear y validar JWT, depende de usted decidir qué biblioteca desea usar.

Token-bearer

En Retrofit para Android, un "portador de tokens" es un tipo de autenticación que se utiliza para autenticar solicitudes de API. Es una forma de pasar un token de autenticación en los encabezados de la solicitud de API, que el servidor puede usar para autenticar al usuario.

La autenticación del portador del token funciona al incluir un encabezado de autorización en la solicitud de API, con el valor establecido en Bearer <YOUR_TOKEN>. <YOUR_TOKEN> es el token real que utiliza el servidor para autenticar al usuario.

Token-bearer

Puede usar la anotación @Header para agregar el encabezado de Autorización a sus métodos de interfaz Retrofit. Aquí hay un ejemplo:

```
interface YourApiService {
    @GET("/data")
    fun getData(@Header("Authorization") authToken: String): Call<Data>
}
```

En este ejemplo, el método getData tiene un parámetro adicional authToken que se anota con @Header("Autorización"). Esto le dice a Retrofit que incluya un encabezado de Autorización con el valor de authToken en la solicitud de API.

Token-bearer

También puede usar un interceptor para agregar los encabezados a todas las solicitudes realizadas por Retrofit, aquí hay un ejemplo:

```
val client = OkHttpClient.Builder()
    .addInterceptor { chain ->
        val newRequest = chain.request().newBuilder()
            .addHeader("Authorization", "Bearer $authToken")
            .build()
        chain.proceed(newRequest)
    }
    .build()
val retrofit = Retrofit.Builder()
    .baseUrl("https://your-api-base-url.com")
    .client(client)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Deberás reemplazar el authToken con su token real y ajustar la URL base de acuerdo con tu API. **Ten en cuenta** que el portador del token debe mantenerse privado y seguro, debe pasarse con cada solicitud que requiera autenticación.

/* Añadir el authtoken a un header usando Retrofit */

Interceptor

- Para saber como agregar un authtoken, o cualquier otro elemento, en un header usando Retrofit, primero debemos saber que es un Interceptor.

- Un interceptor en Retrofit es una interfaz que le permite interceptar y modificar solicitudes y respuestas HTTP.

- Cuando se realiza una solicitud mediante Retrofit, primero pasa por los interceptores antes de enviarse al servidor. De manera similar, cuando se recibe una respuesta del servidor, primero pasa por los interceptores antes de regresar al código de llamada.

- Los interceptores se utilizan para modificar solicitudes y respuestas de varias maneras, como agregar encabezados, registrar, almacenar en caché o reintentar solicitudes en caso de errores de red. Proporcionan una forma de modificar solicitudes y respuestas en todas las llamadas de API en una sola ubicación, en lugar de repetir el mismo código en cada método de API.

Añadir el authtoken a un header usando Retrofit

Ejemplo de Interceptor

Aquí hay un ejemplo de un interceptor que agrega un header personalizado a las solicitudes:

```
val myInterceptor = object : Interceptor {
    override fun intercept(chain: Interceptor.Chain): Response {
        val request = chain.request().newBuilder()
            .addHeader("My-Header", "HEADER_VALUE")
            .build()
        return chain.proceed(request)
    }
}
```

En este ejemplo, el método de intercepción crea una nueva solicitud con un encabezado personalizado "My-Header" y un valor "HEADER_VALUE". Luego continúa con la cadena llamando a chain.proceed(request), que envía la solicitud modificada al servidor y devuelve la respuesta.

Añadir el authtoken a un header usando Retrofit

Ejemplo de Interceptor

Luego puedes agregar el interceptor a la instancia de `OkHttpClient` utilizada por Retrofit, así:

```
val okHttpClient = OkHttpClient.Builder()
    .addInterceptor(myInterceptor)
    .build()
```

Con esta configuración, cada solicitud realizada mediante Retrofit incluirá el encabezado "`My-Header`" con el valor "`HEADER_VALUE`".

Demostración "Token-bearer"

Token-bearer

Crea una nueva instancia de la clase `OkHttpClient` con un Interceptor que agregue el token de autenticación a cada solicitud:

```
val authToken = "YOUR_AUTH_TOKEN_HERE"

val okHttpClient = OkHttpClient.Builder()
    .addInterceptor { chain ->
        val originalRequest = chain.request()
        val newRequest = originalRequest.newBuilder()
            .header("Authorization", "Bearer $authToken")
            .build()
        chain.proceed(newRequest)
    }
    .build()
```

Token-bearer

Cree una nueva instancia de la clase Retrofit, pasando la instancia de `OkHttpClient` y la URL base de su API:

```
val retrofit = Retrofit.Builder()
    .baseUrl("https://example.com/api/")
    .client(okHttpClient)
    .addConverterFactory(GsonConverterFactory.create())
    .build()
```

Token-bearer

Defina una interfaz que describa sus métodos API usando anotaciones Retrofit. Aquí hay un ejemplo:

```
interface MyApiService {
    @GET("users/{id}")
    suspend fun getUser(@Path("id") userId: Int): User
}
```

Utiliza la instancia de Retrofit para crear una instancia del servicio API:

```
val apiService = retrofit.create(MyApiService::class.java)
```

Token-bearer

Llama a los métodos de API en la instancia de servicio como lo harías con cualquier otra interfaz. El token de autenticación se agregará automáticamente a los encabezados de solicitud:

```
val user = apiService.getUser(123)
```

Ahora solo nos falta probar, para esto usaremos un Unit Test.

Para realizar pruebas, puede usar una biblioteca como MockWebServer para simular las respuestas del servidor. Aquí hay un ejemplo:

```
@Test
fun testGetUser() = runBlocking {
    val mockWebServer = MockWebServer()
    mockWebServer.start()

    val apiService = Retrofit.Builder()
        .baseUrl(mockWebServer.url("/"))
        .client(okHttpClient)
        .addConverterFactory(GsonConverterFactory.create())
        .build()
        .create(MyApiService::class.java)

    val expectedUser = User(id = 123, name = "John Doe")
    val responseBody = ResponseBody.create(MediaType.parse("application/json"),
Gson().toJson(expectedUser))
    val mockResponse = MockResponse()
        .setResponseCode(200)
        .setBody(responseBody.string())
    mockWebServer.enqueue(mockResponse)

    val actualUser = apiService.getUser(123)

    assertEquals(expectedUser, actualUser)

    mockWebServer.shutdown()
}
```

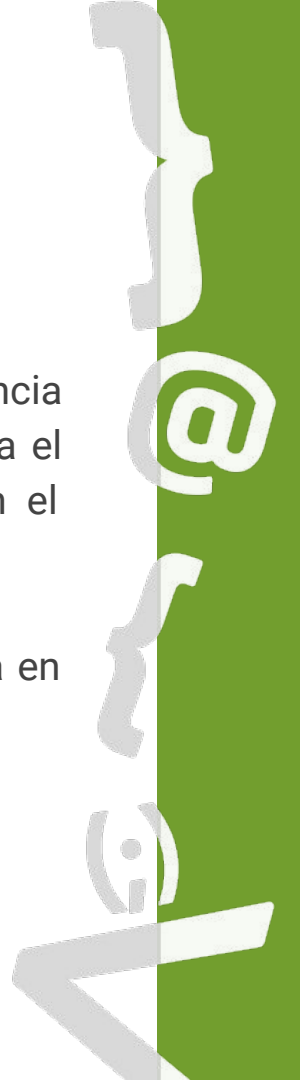
Token-bearer / Unit Test

Token-bearer / Unit Test

Explicación

Esta prueba crea un servidor simulado usando `MockWebServer`, configura una instancia Retrofit con la URL del servidor simulado, pone en cola una respuesta simulada para el método `getUser`, llama al método y afirma que el usuario devuelto coincide con el usuario esperado.

Ten en cuenta que la función `runBlocking` se usa para ejecutar el código de prueba en un contexto coroutine.



¿Cuáles son los beneficios de usar un proceso de autenticación basado en token en comparación con la autenticación tradicional de nombre de usuario/contraseña?

Próxima sesión...

- *Ventajas y desventajas: Glide, Fresco, Picasso*
- *Cómo usar Picasso*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales