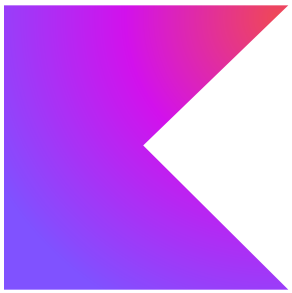

Kotlin

IUT Clermont Auvergne

Février 2024

Kotlin Retrofit

Consommation d'API REST

- On dispose d'un service web qui expose des données à travers une API REST
- On veut récupérer ces données dans un programme en Kotlin

Problèmes :

1. Comment envoyer une requête HTTP ?
2. Comment récupérer les données sous forme Objet ?
3. Comment gérer le non blocage de l'application pendant les attentes dues aux communications réseau ?

Requêtes HTTP en Kotlin

- On utilise les classes Java classiques

```kotlin
import java.io.BufferedReader
import java.io.InputStreamReader
import java.net.HttpURLConnection
import java.net.URI

fun requestUsers(): List<User> {
    val url = URI("https://reqres.in/api/users").toURL()

    val connection = url.openConnection() as HttpURLConnection
    connection.requestMethod = "GET"

    BufferedReader(InputStreamReader(connection.inputStream)).use { reader ->
        val response = StringBuilder()
        reader.forEachLine(response::append)
        // parser la réponse pour créer les objets adéquats
        return parseUsers(response.toString())
    }
}
```

Parser une réponse en JSON

```
import org.json.JSONObject

data class User(val id: Int, val email: String, val firstName: String,
                val lastName: String, val avatarURL: String)

fun parseUsers(response: String): List<User> {
    val users = mutableListOf<User>()
    val jsonUsers = JSONObject(response).getJSONArray("data")
    for (user in jsonUsers) {
        with(user as JSONObject) {
            users.add(User(getInt("id"),
                           getString("email"),
                           getString("first_name"),
                           getString("last_name"),
                           getString("avatar")))
        }
    }
    return users
}
```

- Pénible (surtout s'il y a de nombreux attributs)

Attendre sans bloquer le reste de l'application

- On peut utiliser des threads

```
fun main() {
    Thread {
        println(requestUsers())
    }.start()
    println("Pendant ce temps, je compte…")
    repeat(10) {
        println(it)
        Thread.sleep(100)
    }
}
```

- Quid du traitement d'erreur ? de l'annulation d'une requête ? des problèmes de concurrences ? de la synchronisation avec l'UI ?

Consommation d'API REST

- Cas d'utilisation classique, mais rébarbatif
- Tout le temps le même type de code

- On s'autorise quelques bibliothèques qui font plutôt consensus

1. Pour les requêtes HTTP à l'API REST : *Retrofit*
2. Couplé à la *sérialisation Kotlin* pour la conversion JSON
3. Retrofit peut être non bloquant, pour cela il utilise en interne un pool de threads (néanmoins, nous utiliserons plus tard les *coroutines* Kotlin qui offrent plus de flexibilité et d'efficacité)

La sérialisation en Kotlin

- Offerte par la bibliothèque d'extension `kotlinx.serialization`
- Plusieurs sérialisations possibles : JSON, Protobuf (pour les plus connus) de base et d'autres en option provenant de la communauté (CSV, XML, YAML, ...)
- Basé sur un système d'annotation des classes
- Du code est généré automatiquement (le *serializer*) à la compilation (grâce à un plugin du compilateur)
- Ça n'utilise pas *l'introspection* à l'exécution (c'est statique et type-safe)

- Nous nous limiterons à la sérialisation JSON founie par le package `kotlinx.serialization.json`

La sérialisation en Kotlin

- Les types de base (`Int` , `Float` , `Boolean` , etc. et `String`) sont sérialisables
- Les autres types (composites) sont sérialisables s'ils sont composés d'attributs sérialisables et que leur type est annoté `@Serializable`

```
import kotlinx.serialization.Serializable

@Serializable
data class User(val id: Int, val email: String,
                val firstName: String, val lastName: String,
                val avatarURL: String)
```

- Nombreuses classes de la bibliothèque standard de Kotlin sont sérialisables, notamment les collections

Exemple en JSON

```
import kotlinx.serialization.encodeToString
import kotlinx.serialization.json.Json

fun main() {
  val user = Json.decodeFromString<User>("""{"id":1,
    "email":"george.bluth@reqres.in",
    "firstName":"George",
    "lastName":"Bluth",
    "avatarURL":"https://reqres.in/img/faces/1-image.jpg"}""")

  println(user)
}
// Affiche

User(id=1, email=george.bluth@reqres.in,
     firstName=George, lastName=Bluth,
     avatarURL=https://reqres.in/img/faces/1-image.jpg)
```

Mapping JSON / objet

- Et si l'API renvoie une réponse JSON où nom des valeurs $\neq$ nom des attributs de la classe ?
- On utilise l'annotation `@SerialName`

```
data class User(val id: Int, val email: String,
    @SerialName("first_name") val firstName: String,
    @SerialName("last_name")  val lastName: String,
    @SerialName("avatar")     val avatarURL: String
)
…
val user = Json.decodeFromString<User>("""{"id":1,
    "email":"george.bluth@reqres.in",
    "first_name":"George",
    "last_name":"Bluth",
    "avatar":"https://reqres.in/img/faces/1-image.jpg"}""")

// Même résultat que l'exemple précédent
```

Ignorer des infos

- Et si l'API retourne une réponse JSON avec plein d'éléments inutiles ?
- On personnalise le parser

```
data class User(val id: Int,
    @SerialName("first_name") val firstName: String,
    @SerialName("last_name")  val lastName: String,
)

private val json = Json { ignoreUnknownKeys = true }

fun main() {
  val user = Json.decodeFromString<User>("""{"id":1,
      "email":"george.bluth@reqres.in",
      "first_name":"George",
      "last_name":"Bluth",
      "avatar":"https://reqres.in/img/faces/1-image.jpg"}""")
  println(user)
}
// Affiche
User(id=1, firstName=George, lastName=Bluth)
```

Aller plus loin

- Il est possible de personnaliser le parser plus encore (pretty printing, transformations JSON, etc.) : cf. Kotlin Serialization Guide : JSON features
- Il est aussi possible de faire ses propres *serializers* si ceux générés par défaut ne conviennent pas : cf. Kotlin Serialization Guide : Serializers
- Bref : lisez la documentation pour les fonctionnalités avancées qui dépassent le cadre de cette introduction

Retrofit

- Bibliothèque développée par *Square*
- Beaucoup utilisée par les dév. Android
- Permet de créer un client type-safe pour consommer des API REST en quelques lignes de code

- S'appuie sur :
  1. OkHttp (de Square aussi) pour les requêtes HTTP
  2. De *converters* pour déserialiser les réponses : plusieurs sont disponibles (Moshi, Gson, Jackson, etc.) nous utiliserons évidemment `kotlinx.serialization.json`

Fonctionnement

- Comme pour la sérialisation : repose sur l'utilisation d'annotations

1. Une interface déclare des méthodes annotées qui correspondent aux requêtes à l'API
2. Des classes sérialisables qui recevront les données sont créées relativement aux réponses JSON renvoyées par l'API
3. Un builder permet de configurer et d'instancier une implémentation de l'interface

Préparation des classes sérialisables

```
{
  "page": 2,
  "per_page": 6,
  "total": 12,
  "total_pages": 2,
  "data": [
    {
      "id": 7,
      "email": "michael.lawson@reqres.in",
      "first_name": "Michael",
      "last_name": "Lawson",
      "avatar": "https://reqres.in/img/faces/7-image.jpg"
    },
    {
      "id": 8,
      "email": "lindsay.ferguson@reqres.in",
      "first_name": "Lindsay",
      "last_name": "Ferguson",
      "avatar": "https://reqres.in/img/faces/8-image.jpg"
    },
    …
  ],
  "support": {
      "url": "https://reqres.in/#support-heading",
      "text": "To keep ReqRes free, contributions
               towards server costs are appreciated!"
  }
}
```

```
@Serializable
data class UsersResponse(
    val page: Int,
    @SerialName("total_page")
    val total: Int,
    @SerialName("data")
    val users: List<User>
)

@Serializable
data class User(
    val id: Int,
    val email: String,
    @SerialName("first_name")
    val firstName: String,
    @SerialName("last_name")
    val lastName: String,
    @SerialName("avatar")
    val avatarURL: String
)

@Serializable
data class Data<T>(val data: T)
```

Mise en place de l'interface des requêtes

```
const val BASE_URL="https://reqres.in/api/"

interface ReqResService {
  @GET("users")
  fun getUsers(): Call<UsersResponse>

  …
}
```

- Chaque verbe HTTP correspond à une annotation (`@GET` , `@POST` , `@PUT` , `@DELETE` , etc.)
- On passe en paramètre le endpoint de la requête (pas de `/` au début)
- La fonction retourne un `Call<T>` où `T` est la classe utilisée pour désérialiser la réponse

Mise en place de l'interface des requêtes

- Possibilité de faire des requêtes dynamiques

```
interface ReqResService {
  @GET("users/{id}")
  fun getUser(@Path("id") id: Int): Call<Data<User>>
}
```

1. Placeholders dans le chemin (`{xxx}`) : seront remplacés par la valeur du paramètre annoté `@Path("xxx")`

`getUser(2)` fera une requête GET au endpoint https://reqres.in/api/users/2

Mise en place de l'interface des requêtes

- Possibilité de faire des requêtes dynamiques

```
interface ReqResService {
  @GET("users")
  fun getUsersPage(@Query("page") pageNum: Int): Call<UsersResponse>
}
```

2. Les paramètres de requête annotés `@Query("xxx") val` (ou `@QueryMap opt: Map<String, String>`) seront ajoutés à fin la fin du endpoint : `users?xxx=val` (ou `users?xxx=opt[xxx]&yyy=opt[yyy]`)

`getUsersPage(2)` fera une requête GET au endpoint https://reqres.in/api/users?page=2

Mise en place de l'interface des requêtes

- Possibilité de faire des requêtes dynamiques

```
interface ReqResService {
  @POST("users")
  fun createUser(@Body user: User): Call<User>
}
```

3. On peut utiliser l'annotation `@Body` pour sérialiser les paramètres dans le payload de la requête

Mise en place de l'interface des requêtes

- Certaines API nécessitent l'envoi d'informations dans l'entête de la requête
- Spécification du mimetype, token d'authentification, etc.

```
interface ReqResService {
  @Headers({
      "Accept: application/jwt",
      "User-Agent: WebKit"
  })
  @POST("register")
  fun (@Body user: User): Call<Token>

  @PUT("user/{id}")
  fun udpate(@Path("id") id: Int, @Body user: User,
             @Header("Authorization") token: String): Call<User>
}
```

Configuration du client

- On utilise un builder pour paramétrer le client
- On spécifie l'URL de base de l'API (finir par `/`)
- On spécifie le *converter* pour la sérisalisation (utiliser `retrofit2-kotlinx-serialization-converter`, de Jake Wharton, pour la sérialisation Kotlin)

```
const val REQRES_BASE_URL="https://reqres.in/api/"

private val json = Json { ignoreUnknownKeys = true }

private val reqresService = Retrofit.Builder()
    .baseUrl(REQRES_BASE_URL)
    .addConverterFactory(json.asConverterFactory(
                        MediaType.get("application/json")))
    .build()
```

Instanciation et utilisation du client

```
fun main() {
  val reqresClient = reqresService.create<ReqResService>()

  // Requête synchrone (bloquante)
  val respUsersSync = reqresClient.getUsers().execute()
  respUsersSync.body()?.let {
      // Utilisation du résultat
  }

  //Requête asynchrone (non-bloquante)
  val respUsers = reqresClient.getUsers().enqueue(
    object: Callback<UsersResponse> {
      override fun onResponse(call: Call<UsersResponse>,
                              resp: Response<UsersResponse>) {
        // La requête a réussi
      }
      override fun onFailure(call: Call<UsersResponse>, error: Throwable) {
        // La requête a échoué
      }
    }
  )
}
```