TD6 - Les Appels Réseau

Une application android a souvent besoin d’une source de données externe, par exemple un site web avec lequel elle va échanger des données.

Prenons l’exemple de l’application youtube, qui va interagir avec les serveurs de google afin de récupérer une liste de vidéos.

Pour ceux qui l’ignorent, l’application n’accède pas directement aux données via des requêtes SQL mais par des appels vers des APIs (Application Programming Interface).

Pour vous expliquer simplement, le serveur met à disposition une liste d’url par lesquels les applications vont pouvoir accéder aux données.

Le format des données échangées se fait le plus souvent en JSON ou en XML.

On dit alors que le serveur fournit un webservice, dans notre cas une API REST.

## [Retrofit](https://square.github.io/retrofit/)

Tout d’abord, nous allons utiliser dans ce TP cette librairie très utilisée sur Android. Retrofit permet très simplement et rapidement d’effectuer des appels HTTP de manière synchrone ou asynchrone.

Comme pour toute librairie, il est nécessaire de l’importer dans notre projet

Pour cela il vous suffit d’ajouter la ligne suivante à vos dépendances :

| implementation 'com.squareup.retrofit2:retrofit:2.9.0' implementation 'com.squareup.retrofit2:converter-gson:2.9.0' |
| --- |

Nous allons dans ce TD interagir avec l’API de github qui a comme avantage de proposer une version publique, ne nécessitant pas d’authentification.

Retrofit se base sur une interface qui décrit notre API REST:

| **public** **interface** **GithubService** {   **public** **static** **final** String ENDPOINT = "https://api.github.com";   @GET("/users/{user}/repos")  List<Repo> **listRepos**(@Path("user") String user);   @GET("/search/repositories")  List<Repo> **searchRepos**(@Query("q") String query); } |
| --- |

Cette interface décrit un webservice contenant 2 méthodes :

* listRepos affiche la liste des dépôts d’un utilisateur
* searchRepos recherche des dépôts en fonction de mots clés

Regardons maintenant les informations associées à ces fonctions :

En premier lieu, nous devons définir l’url du webservice à appeler :

| **public** **static** **final** String ENDPOINT = "https://api.github.com"; |
| --- |

Pour chaque fonction, ajouter une annotation définissant la méthode d’appel :

| @GET("/users/{user}/repos")  listRepos(@Path("user") String user); |
| --- |

Les principales méthodes suivantes sont disponibles :

* @GET (Récupèration de l’info)
* @POST (Création de l’info)
* @PUT (Modification de l’info)
* @DELETE (Suppression de l’info)

Puis l’url de la méthode à appeler :

| @GET("/users/{user}/repos") List **listRepos**(@Path("user") String user); |
| --- |

Cette url viendra s’ajouter à notre ENDPOINT, il faut donc éviter de mettre l’url complète

Il nous faut ensuite la possibilité de paramétrer nos appels.

Pour cela, deux méthodes nous sont offertes :

## Injecter des arguments à notre url

| @GET("/users/{user}/repos") List **listRepos**(@Path("user") String user); |
| --- |

Par exemple, pour l’utilisateur adrienbusin, l’url suivante sera construite : <https://api.github.com/users/adrienbusin/repos>

## Ajouter des arguments en paramètre de notre url

| @GET("/search/repositories") List **searchRepos**(@Query("q") String query); |
| --- |

Ainsi, avec le paramètre picasso, l’url suivante sera construite : <https://api.github.com/search/repositories?q=picasso>

## Envoyer des paramètres de formulaire (en POST)

| POST("/search/repositories") List **searchRepos**(@Field("q") String query); |
| --- |

## Interpréter le retour du webservice

Retrofit facilite la lecture des données renvoyées par le webservice. Pour cela, il retourne directement le résultat dans des objets Java (ou Kotlin).

Par exemple, dans le cas de searchRepos, vous avez pu remarquer que l’objet List avant le nom de la méthode. Ici, Retrofit va nous retourner le résultat automatiquement sous forme d’une ArrayList de Repos.

NB: La désérialisation de l’objet est réalisée par GSON (bibliothèque Java qui peut être utilisée pour convertir des objets Java en leur représentation JSON), le nom des attributs privés doivent donc être identiques aux champs de l’objet JSON afin d’être injectés.

Il ne reste qu’à définir l’objet Repo, cliquez sur l’url suivante : <https://api.github.com/users/adrienbusin/repos>. La page affichée contient le retour du webservice. Nous allons retenir pour l’instant que les attributs id, name, full\_name et html\_url, ce qui donne l’objet :

| **public** **class** **Repo** {  **private** **int** id;  **private** String name;  **private** String full\_name;  **private** String html\_url;    //getters & setters à générer avec android studio } |
| --- |

NB : Si vous désirez renommer les variables vous pouvez utiliser l’annotation

| @SerializedName("full\_name") **private** String fullName; |
| --- |

## Réaliser nos appels

Il nous faut maintenant fournir notre interface à un objet nommé Retrofit.Builder. C’est lui qui va nous retourner une implémentation de notre webservice :

| GithubService service = **new** Retrofit.Builder()  .baseUrl(GithubService.ENDPOINT)  .addConverterFactory(GsonConverterFactory.create())  .build()  .create(GithubService.class); |
| --- |

Nous pouvons ensuite réaliser nos appels de webservice en utilisant l’objet créé par le Retrofit.Builder, pour cela Retrofit nous offre plusieurs possibilités : synchrone ou asynchrone.

Pour la majorité des appels que nous effectuerons, nous utiliserons les appels asynchrone pour éviter de bloquer l’interface utilisateur..

L’appel asynchrone peut être effectué « depuis » le thread principal de façon assez simple.

Le retour du webservice s’effectue par un callback, un objet dont les méthodes seront appelées suite à la réception de données du webservice :

| **public class MainActivity extends AppCompatActivity {   @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);   GithubService githubService = new Retrofit.Builder()  .baseUrl(GithubService.ENDPOINT)  .addConverterFactory(GsonConverterFactory.create())  .build()  .create(GithubService.class);   githubService.listRepos("adrienbusin").enqueue(new Callback<List<Repo>>() {  @Override  public void onResponse(Call<List<Repo>> call, Response<List<Repo>> response) {  afficherRepos(response.body());  }   @Override  public void onFailure(Call<List<Repo>> call, Throwable t) {   }  });  }   public void afficherRepos(List<Repo> repos) {  Toast.makeText(this,"nombre de dépots : "+repos.size(), Toast.LENGTH\_SHORT).show();  }  }** |
| --- |

Pour effectuer les appels de cette façon, il est nécessaire de modifier notre interface GithubService afin d’ajouter les callbacks :

| **public** **interface** **GithubService** {  ...  @GET("/users/{user}/repos") Call<List<Repo>> listRepos(@Path("user") String user);  ... } |
| --- |

## Exercice:

Ajouter un second appel réseau (searchRepos) et afficher une liste de repositories dans un RecyclerView. Vous utiliserez une première activité pour demander à l’utilisateur de saisir un nom de repo à rechercher. Puis dans une seconde activité vous afficherez la liste des repo reçu par l’appel réseau.