SYM - Labo 2

Auteurs : Benoît Schopfer, Antoine Rochat et Jérémie Châtillon

3) Manipulations :

Nous avons concu notre code de sorte que chaque manipulation soit le plus similaire possible. Chaque manipulation possède un fragment dédié ainsi qu'un *SymComManager* gérant leur communication avec le serveur.

Remarque globale :

Modifier l'orientation du téléphone provoque la recréation du layout... Ceci engendre la perte des données entrées dans les champs et, pour le fragment GraphQL, relance la requête pour obtenir l'ensemble des auteurs... Ceci n'est évidemment pas optimal! Il aurait fallu, pour chaque fragment, enregistrer l'état actuel du fragment lorsqu'il est détruit, de sorte à pouvoir le reconstruire à sa prochaine création. Pour le fragment GraphQL, il aurait fallu enregistrer le résultat de la première requête afin de pouvoir le recharger à la recréation du layout. Nous n'avons cependant pas jugé utile de faire tout cela dans le cadre de ce laboratoire...

3.1) Transmission asynchrone :

La transmission asynchrone est gérée dans la classe *AsyncSymComManager* utilisée par le fragment *AsyncSendFragment*. À la création du fragment, on crée une instance d'*AsyncSymComManager* qui sera chargée de gérer la transmission avec le serveur. On passe à cette instance un *CommunicationEventListener* définissant ce qui devra être fait lorsque l'instance recevra la réponse du serveur, à savoir afficher cette réponse dans le champs prévu à cet effet. Enfin, on définit l'action du bouton qui est d'envoyer une requête au serveur contenant le texte du *EditText* du fragment.

Pour envoyer la requête, on utilise la méthode *sendRequest()* définie dans l'*AsyncSymComManager*. Cette méthode définit les headers HTTP à utiliser pour la requête puis crée et démarre une *AsyncTask* définie dans la classe *MyAsyncTask*.

Cette tâche asynchrone travaille sur un thread à part du thread UI. Elle utilise le singleton de la classe *SymComManager* afin de créer et envoyer la requête au serveur à l'aide de la librairie *OKHTTP*, puis elle attend la réponse du serveur. Lorsqu'elle la reçoit, elle repasse sur le thread UI et utilise le *CommunicationEventListener* reçu à sa création pour traiter la réponse et modifier l'UI en conséquence.

Comme demandé, nous avons travaillé avec des requêtes *OKHTTP* synchrones. L'asynchronisme est créé par l'utilisation d'une *AsyncTask*. Cette façon de faire n'est pas très intelligente puisque *OKHTTP* est capable de gérer tout seul cet asynchronisme en utilisant la méthode *enqueue()* et en lui donnant le callback, ce qui éviterait l'usine à gaz de l'*AsyncTask*...

3.2) Transmission différée :

La transmission différée est réalisée dans la classe *DelayedSymComManager* utilisée par le fragment *DelayedSendFragment*. Le fonctionnement du fragment est identique à celui de la transmission asynchrone excepté qu'on utilise ici un *DelayedSymComManager*.

Lorsqu'il est créé, ce *DelayedSymComManager* crée un thread qui tourne en continu. Ce thread vérifie qu'une connexion internet est disponible et, si c'est le cas, envoie au serveur toutes les requêtes en attente présentes dans l'*ArrayList waitingRequests* à l'aide de la méthode *sendRequest()*, puis s'endors pendant 10 secondes avant de recommencer sa tâche. Si la connexion internet n'est pas disponible, le thread se rendors immédiatement.

Lorsque la méthode *sendRequest()* du *DelayedSymComManager* est appelée, elle commence par vérifier qu'une connexion internet est disponible. Si c'est le cas, elle envoie sans autres la requête reçue. Dans le cas contraire, elle enregistre la requête dans l'*ArrayList waitingRequests* qui sera traitée par le thread lorsqu'une connexion internet sera à nouveau disponible.

Limitations :

Les limites de cette façon de faire sont assez évidentes.

La première est que les requêtes en attente ne sont enregistrées qu'en mémoire... Ainsi, si l'application est fermée ou le téléphone s'éteint, toutes les requêtes en attente sont perdues.

Une solution à ce problème serait d'enregistrer les requêtes en attente dans la mémoire permanente de l'appareil lorsque l'application est quittée afin de conserver les requêtes en attente. On pourrait alors récupérer les requêtes en attente au prochain lancement de l'application et les traiter alors.

Une autre limitation de la méthode et l'utilisation d'un thread afin de traiter les requêtes en attente. Cette façon de faire est tout sauf optimisée et élégante mais à l'avantage d'être simple à mettre en place. Pour faire mieux, il faudrait utiliser un JobScheduler permettant de planifier des actions lorsque certaines conditions sont remplies (batterie suffisante, connexion internet disponible, ...). Le système s'occupe alors de lancer l'opération que lrosque les conditions requises sont remplies ce qui est bien meilleur pour la batterie qu'un thread tournant en boucle et endormi manuellement pendant 10 secondes!

3.3) Transmission d'objets :

3.3.1) Format JSON :

La transmission JSON est réalisée dans la classe *JSONObjectSymComManager* utilisée par le fragment *JsonObjectSendFragment*. Le fonctionnement du fragment est identique à celui de la transmission asynchrone excepté qu'on utilise ici un *JsonObjectSymComManager* transformant l'objet reçu en un Json avant de l'envoyer au serveur.

3.3.2) Format XML :

La transmission XML est réalisée dans la classe *XmlObjectSymComManager* utilisée par le fragment *XmlObjectSendFragment*. Le fonctionnement du fragment est identique à celui de la transmission JSON excepté qu'on utilise ici un *XmlObjectSymComManager* transformant l'objet reçu en un XML avant de l'envoyer au serveur.

3.3.3) GraphQL - Format JSON :

Cette activité est gérée par le fragment *GraphQLSendFragment* et la transmission avec le serveur est gérée par la classe *GraphQLObjectSymComManager*.

À la création du fragment, la méthode *populateAuthorsSpinner()* est appelée. Celle-ci utilise le *GraphQLObjectSymComManager* pour faire un appel à l'API GraphQL et récupérer l'id, le nom et le prénom de tous les auteurs. Lorsqu'ell reçoit la réponse, elle affiche la liste dans le spinner et sélectionne le premier auteur de la liste.

Cette sélection (automatique) a pour effet d'appeler la méthode *populateAllPostByAuthor()* qui fait un nouvel appel à l'API GraphQL afin de récupérer tous les posts de cet auteur et de les afficher dans le RecyclerView du fragment.

Les appels à l'API GraphQL sont très simples. Dans chacune des deux méthodes citées, on défini le Query à utiliser, puis on appelle la méthode sendRequest de la classe *JsonObjectSymComManager()* dont *GraphQLObjectSymComManager* hérite. Autrement dit, le Query GraphQL est envoyé par JSON au serveur GraphQL, et la réponse reçue est également au format JSON.

3.4) Transmission compressée :

La transmission compressée est réalisée dans la classe *CompressedSymComManager* utilisée par le fragment *CompressedSendFragment*. Le fonctionnement du fragment est identique à celui de la transmission JSON excepté qu'on utilise ici un *CompressedSymComManager*. Le contenu de la requête est compressé dans le *SymComManager* qui reçoit le paramètre *enableCompression* valant *true* et la réponse est décompressée dans la classe *MyAsyncTask* avant qu'elle ne revienne sur le thread UI pour mettre à jour le fragment.

4) Réponses au questions :

4.1) Traitement des erreurs :

```
Les interfaces AsyncSendRequest et CommunicationEventListener utilisées au point 3.1 restent très (et certainement trop) simples pour être utilisables dans une vraie application : que se passe-t-il si le serveur n'est pas joignable dans l'immédiat ou s'il retourne un code HTTP d'erreur ? Veuillez proposer une nouvelle version, mieux adaptée, de ces deux interfaces pour vous aider à illustrer votre réponse.
```

Si le serveur n'est pas joignable dans l'immédiat, la requête va tout simplement se perdre dans la nature... Une erreur est levée dans le thread de la *MyAsyncTask* et directement catchée. Comme cela se passe sur un thread à part, nous ne pouvons pas directement mettre l'UI à jour. Si on ne fait rien, on ne va donc rien voir s'afficher! Une erreur se sera produite, mais il n'y aura aucun retour utilisteur... Pour régler ce problème, nous avons fait en sorte d'enregistrer l'erreur (pour la transmettre au thread UI), d'annuler la tâche asynchrone, puis d'afficher l'erreur dans le fragment à la place de la réponse.

Une meilleure solution serait d'ajouter une méthode *handleError()* à l'interface *CommunicationEventListener*. Ainsi, on pourrait définire précisément que faire lorsqu'une erreur de communication se produit et il suffirait d'appeler cette méthode afin de traiter l'erreur comme désiré.

4.2) Authentification :

```
Si une authentification par le serveur est requise, peut-on utiliser un protocole asynchrone ?
```

Oui, un protocole asynchrone est parfaitement utilisable même si le serveur requiert une authentification. Cela peut toutefois poser un problème : que peut faire un utilisateur durant le temps d'attente de son authentification? Il y a deux solutions. Soit on l'oblige à attendre que le serveur ait confirmé la validité de son authentification (ce qui revient plus ou moins à rendre synchrone une requête asynchrone...), soit on l'autorise à accéder au contenu protégé par l'authentification mais de manière limitée uniquement.

On peut par exemple imaginer une application permettant d'écrire et envoyer des messages mais demandant une authentification pour les envoyer. L'utilisateur pourrait alors s'authentifier, commencer à rédiger son message durant la requête asynchrone d'authentification mais devrait attendre d'être authentifié par le serveur (donc d'avoir reçu la réponse de la requête asynchrone) pour pouvoir envoyer son message.

```
Quelles seraient les restrictions ?
```

Les restrictions seraient une application ne proposant que du contenu nécessitant une authentification. Par exemple, on est obligé d'être identifié pour pouvoir voir le contenu qui se trouve sur Facebook. Dans ce cas, utiliser un protocole asynchrone pour la page de login est inutile (mais reste possible), puisqu'on est obligé d'attendre la réponse du serveur pour pouvoir continuer...

```
Peut-on utiliser une transmission différée ?
```

On peut également utiliser une transmission différée mais on se heurte aux même limitations que pour une requête asynchrone.

4.3) Threads concurrents :

```
Lors de l'utilisation de protocoles asynchrones, c'est généralement deux threads différents qui se préoccupent de la préparation, de l'envoi, de la réception et du traitement des données. Quels problèmes cela peut-il poser ?
```

- Un des problèmes qui peut survenir est que seul le thread UI peut modifier la vue. Si d'autres threads pouvaient le faire, on pourrait avoir des problèmes de concurrence. Résultat, si on a beaucoup de threads annexes qui sont lancés, il peut y avoir des

problèmes de performances lorsqu'ils tenteront tous de mettre la vue à jour en même temps...

- On peut aussi avoir des problèmes d'affichage incorrect par rapport aux résultats de la requête... Imaginons par exemple une interface avec un bouton incrémentant un compteur situé sur un serveur, ainsi qu'un textView affichant la valeur du compteur retournée par le serveur. Cliquer sur le bouton lance un nouveau thread qui effectue la requête et attend le résultat. Lorsqu'il reçoit le résultat, il me le texteView à jour. Imaginons maintenant que l'utilisateur spam le bouton. De nombreux threads sont créés et de nombreuses requêtes sont envoyées au serveur. Le problème est alors qu'on a aucun moyen de savoir quelle réponse correspond à quelle requête... Autrement dit, on peut parfaitement imaginer que le thread de la requête 1 se fasse préempter juste avant d'afficher le résultat et que tous les autres threads lui passent devant... Puis que finalement, il affiche le résultat 1, malgré qu'il y a eu bien plus de requêtes et que le compteur du serveur ne vale pas du tout 1...

4.4) Écriture différée :

```
Lorsque l'on implémente l'écriture différée, il arrive que l'on ait soudainement plusieurs transmissions en attente qui deviennent possibles simultanément. Comment implémenter proprement cette situation (sans réalisation pratique) ? Voici deux possibilités :
    - Effectuer une connexion par transmission différée
    - Multiplexer toutes les connexions vers un même serveur en une seule connexion de transport. Dans ce dernier cas, comment implémenter le protocole applicatif, quels avantages peut-on espérer de ce multiplexage, et surtout, comment doit-on planifier les réponses du serveur lorsque ces dernières s'avèrent nécessaires ?

Comparer les deux techniques (et éventuellement d'autres que vous pourriez imaginer) et discuter des avantages et inconvénients respectifs.
```

Imaginons une situation qui se produit fréquemment dans le cadre d'une application. Disons qu'un utilisateur effectue les 3 actions suivantes dans l'ordre donné:

1. L'utilisateur se crée un compte.
2. L'utilisateur modifie ses informations.
3. L'utilisateur supprime son compte.

Dans le cas d'une écriture différée, cela veut dire que chacune de ces 3 requêtes est enregistrée dans une base de données locale. Lorsqu'une connexion avec le serveur est disponible, un thread à part s'occupe alors de les envoyer. À ce moment, on a les 2 possibilités ci-dessous :

Effectuer une connexion par transmission différée :

Effectuer une nouvelle connexion pour chaque requête permet d'alléger le poids des requêtes. Ainsi, si l'une d'elle échoue, seule celle-ci sera renvoyée... En revanche, dans notre cas d'utilisation, cela pose un problème majeur. Comment définir l'ordre des requêtes et comment s'assurer que la requête de création arrive avant celle de modification et que la requête de suppression arrive bien en dernier? Un autre ordre poserait immédiatiement problème...

Multiplexer toutes les connexions vers un même serveur en une seule connexion de transport :

Effectuer une seule connexion et envoyer toutes les requêtes d'un coup permet de régler le problème d'ordre des requêtes. On peut alors spécifier l'ordre dans lequel le serveur doit traiter les requêtes et donc s'assurer que tout se déroule dans le bon sens.

En revanche, mettre plusieurs requêtes en une seule peut alourdir considérablement la requête finale... Si un problème de transmission arrive, il faut alors retransmettre l'ensemble des requêtes... Cette taille plus importante peut également poser des problèmes évident de performances dans le cas d'une mauvaise connexion (faible débit)... Si le payload fait plusieurs Mo mais que la connexion est lente, l'envoi risque de durer longtemps voir de ne jamais aboutir... Dans ce cas, le nombre de requêtes en attente n'aura de cesse d'augmenter, alourdissant le payload à envoyer au serveur et accentuant encore le problème... On entre alors dans un cercle vicieux sans fin.

4.5) Transmission d'objets :

```
a. Quel inconvénient y a-t-il à utiliser une infrastructure de type REST/JSON n'offrant aucun service de validation (DTD, XML-schéma, WSDL) par rapport à une infrastructure comme SOAP offrant ces possibilités ? Est-ce qu'il y a en revanche des avantages que vous pouvez citer ?
```

Les inconvénients sont nombreux :

- Un client peut envoyer des données invalides au serveur. Dans un tel cas, le comportement le plus probable du serveur serait d'ignorer l'objet reçu et de retourner une erreur signalant le problème au client.
- Dans la même idée, le client ne peut pas vérifier la validité de son objet avant de l'envoyer au serveur. Il est obligé de le transmettre pour savoir si l'objet est valide ou non pour le serveur. Cela peut poser problème si l'objet transmis est d'une taille conséquente... Dans ce cas, devoir faire plusieurs aller-retours entre le client et le serveur pour valider l'objet peut prendre un temps considérable qu'un service de validation permettrait d'éviter.

Ces inconvénients peuvent aussi être des avantages :

- Un serveur fonctionnant avec une base de données schemaless pourrait par exemple accepter tout type d'objets, quelque soit leur structure et les enregistrer...
- En supposant que le client connaît la structure d'objet que supporte le serveur, cela évite de faire des requêtes de validation au serveur. Comme le client sait ce que le serveur attend, il n'a qu'à créer l'objet correspondant et l'envoyer.

```
b. L’utilisation d’un mécanisme comme Protocol Buffers8 est-elle compatible avec une architecture basée sur HTTP ? Veuillez discuter des éventuels avantages ou limitations par rapport à un protocole basé sur JSON ou XML ?
```

Oui, l'utilisation d'un mécanisme comme Protocol Buffers est compatible avec une architecture basée sur HTTP. Le grand avantage de cette technologie est de permettre de sérialiser des données de façon plus flexible, plus efficace et plus légère que le permet le JSON ou le XML. De plus, Protocol Buffers permet une validation des données comme le permet la DTD en XML. En revanche, l'un des principal inconvénient de cette technologie est de rendre les données totalement illisible pour un humain. En effet, les données sont sérializées sous forme binaire, ce qui les rend illisibles. La validation ""à l'oeil" est donc totalement impossible, contrairement au XML ou au JSON.

```
c. Par rapport à l’API GraphQL mise à disposition pour ce laboratoire. Avez-vous constaté des points qui pourraient être améliorés pour une utilisation mobile ? Veuillez en discuter, vous pouvez élargir votre réflexion à une problématique plus large que la manipulation effectuée.
```

Oui, un des points qui pourrait être améliorer est de paginer les réponses du serveur. Au lieu de retourner l'ensemble des auteurs, il serait plus judicieux de retourner les 10 ou 20 premiers. Si le client souhaite afficher les auteurs suivants, il ferait alors une nouvelle requête au serveur pour les charger. Il faudrait faire de même pour les posts d'un auteur et n'en renvoyer qu'une partie au lieu de la totalité.

Cela permettrait d'accélérer le chargement en diminuant considérablement la taille des données transmises. D'autant plus que si les données deviennent vraiment très conséquentes (par exemple plusieurs milliers de posts avec un long texte pour un seul auteur), le chargement pourrait durer très longtemps et même faire crasher l'application client pour consommation trop importante de mémoire...!

De manière générale, il faut toujours paginer les données du côté du serveur afin de ne pas surcharger le client.

Transmission compressée :

```
Quel gain peut-on constater en moyenne sur des fichiers texte (xml et json sont aussi du texte) en utilisant de la compression du point 3.4 ? Vous comparerez vos résultats par rapport au gain théorique d’une compression DEFLATE, vous enverrez aussi plusieurs tailles de contenu pour comparer.
```

Pour calculer le gain moyen obtenu avec la compression deflate, nous avons rempli les deux textfields de notre fragment *CompressedSendFragment* avec des caractères alphanumériques aléatoires de différentes longueurs.

Lors de l'envoi, nous avons vérifié la longueur du texte avant et après la compression. Nous avons fait cette manipulation 10 fois afin d'avoir un minimum de données pour pouvoir faire une moyenne.

Nous avons obtenu les données suivantes :

longueur requête avant compression	**longueur requête après compression**	**gain**
1665	1279	~23.18 %
957	752	~21.42 %
5673	4287	~24.43 %
2337	1783	~23.71 %
6553	4945	~24.54 %
4717	3571	~24.30 %
6429	4852	~24.53 %
5345	4041	~24.40 %
4289	3244	~24.36 %
377	318	~15.65 %

Le gain moyen obtenu est donc d'environ 23.05 %, soit presque un quart! On remarque que plus le texte est long, plus le gain est important.

En même temps que cette analyse des données envoyées, nous avons observé les données reçues dans la réponse car la réponse aussi est compresée. Nous avons donc étudié la longueur du body de la réponse reçue avant et après la décompression. Nous avons obtenu les données suivantes:

longueur réponse avant décompression	longueur réponse après décompression	gain
1754	2500	~29.84 %
1214	1792	~32.25 %
4781	6508	~26.54 %
2265	3172	~28.59 %
4539	7388	~38.56 %
4059	5552	~26.89 %
5352	7264	~26.32 %
4532	6180	~26.67 %
3737	5124	~27.07 %
765	1212	~36.88 %

Pour les réponses, on obtient un gain d'environ 29.96 %, soit encore mieux qu'à l'envoi. L'une des explications les plus probable à cette différence de gain entre la compression de la requête et celle de la réponse est qu'on ne connaît pas l'algorithme de compression utilisé par le serveur. Il est probable que le serveur utilise un algorithme plus performant que celui que nous utilisons pour compresser la requête...