IFT3912 - Développement et maintenance, Architecture
Groupe1

Nicolas Lavoie Drapeau Alexandre Mathon-Roy Luc-Antoine Girardin William Tchoudi
Lionnel Lemogo

7 mars 2014

Avant-propos

Comme nous en avions convenus lors de la partie précédente, nous avons séparé le travail de sorte que Lionel et Alexandre ont travaillé sur le site web et Nicolas, William et Luc-Antoine ont travaillé sur le pont. Il ne faut donc pas s'alarmer si vous voyez seulement trois noms dans les champs "@author".

0.1 Les modules :

Dans les sections suivantes, les rôles des différents modules seront décrits brièvement. Pour une description plus poussée des classes et de leurs méthodes, voir la Javadoc.

0.1.1 Module client

Le client est l'utilisateur. Il visualise le site web remis pour le premier livrable. Ce site web contient des formulaires (forms) qui seront utilisés pour communiquer les requêtes de l'usager au serveur http, sous format JSON, qui les enverra ensuite au **pont**.

0.1.2 Module pont

Le module qui s'occupe d'établir le lien de communication entre le **client** (par le biais du serveur http) et la **base de données**, de gérer les permissions des utilisateurs ainsi que de traiter l'information qui passe entre les deux autres modules. Cette information est sous format JSON quand elle vient du client et sous forme de *ResultSet* quand elle vient de la DB. Le pont contiendra les méthodes appropriées pour transformer un *ResultSet* en String JSON.

Ce module est composé des packages suivant :

ca.diro

Contient la logique d'exécution du serveur.

Main
: Initialise le serveur, son moyen de gérer la connexion et l'authentification des utilisateurs. Les champs et méthodes ne requièrent pas vraiment d'explication ici ; *main* gère les arguments si applicables et *initSecureServer* s'occupe de lancer le serveur avec les méthodes de sécurité que nous aurons décidées.

RequestHandler
: Toutes les requêtes passeront d'abord par ce *AbstractHandler*, pour s'assurer que le format de la requête est conforme à celui utilisé par le client et le pont. On peut le voir comme un "dispatcher" ou encore comme un "sanitizer", qui filtre les requêtes malicieuses ou erronées. Comme il ne contient qu'une méthode publique héritée de *AbstractHandler*, nous ne justifierons pas ses méthodes.

ca.diro.UserHandlingUtils

Contient la logique de traitement des actions.

ActionPermissionsException
: Une implémentation de *Exception*, elle est lancée si un utilisateur n'a pas les permissions requises pour accéder à une certaine action. Par exemple, un utilisateur qui essaye d'effacer un événement alors que sa session est expirée. Elle a été jugée utile pour avoir un moyen d'informer le client qu'une requête n'a pas pu être exécutée.

UserPermissions
: Un enum qui sert à décrire les différents types de permissions requises pour différentes actions. Pourquoi avoir utilisé un enum ? Comme les types d'accès permis étaient déjà définis pour le projet, il ne semblait pas être nécessaire de créer une interface ou une classe abstraite qui implémente *Comparable* et d'utiliser une classe par permission. Les types de permissions sont définis comme suit :

  UNAUTHORIZED
  : Équivalente au "null", c'est une permission qui est impossible à remplir, même par un administrateur.

  ADMIN
  : C'est la permission absolue, qui est donnée seulement aux utilisateurs qui administrent le site web.

EVENT_OWNER
C'est la permission requise pour modifier un événement. On pourrait aussi rajouter des modérateurs.

COMMENT_OWNER
C'est la permission requise pour modifier ou effacer un commentaire. Elle vient après *EVENT_OWNER* pour qu'on puisse décider que le créateur d'un événement peut retirer tous les commentaires sur son event s'il le désire.

PREMIUM
Une petite blague, un état élevé de permission pour certains utilisateurs. En même temps, pourrait servir pour créer des modérateurs.

LOGGED_USER
Un utilisateur qui a été authentifié peut effectuer l'action qui requiert cette permission.

REGISTERED_USER
Un utilisateur qui possède un compte peut effectuer cet action. (Par exemple, la connexion à son compte.)

NONE
La permission universelle pour les utilisateurs qui ne sont pas connectés.

UserPermissionsHandler
Les requêtes qui sont acceptées et qui requièrent des permissions d'accès spécifiques sont passées à ce *Handler*. Il jugera si l'utilisateur a les permissions requises pour faire l'appel ou encore s'il doit propager la requête vers *OwnerPermissionsHandler*.

OwnerPermissionsHandler
Les requêtes qui requièrent des permissions encore plus particulières, c'est-à-dire les permissions de "Ownership" sont envoyées à ce *Handler*.

ca.diro.UserHandlingUtils.Actions

Contient la spécification des actions possibles, donne l'information sur leur permission requises et utilise une méthode pour appeler la commande attachée à cette action. Notez que la vérification des permissions doit être faite à l'extérieur de cette classe. Ce package contient les classes :

IAction
L'interface pour les actions effectuées dans le pont, en réponse à une requête du client. Elle est utile si nous devons ajouter d'autres types d'actions que les "*UserAction*", par exemple si nous désirons ajouter des modérateurs. On pourrait alors avoir une classe abstraite "*AdminAction*" qui retourne toujours "*UserPermission.ADMIN*" comme permission requise mais dont chaque classe doit implémenter "*performAuthorizedAction*".

Les méthodes définies sont :

getTarget()
Retourne l'ID de la cible de cette action.

getCaller()
Retourne l'ID de l'appelant de cette action.

getRequiredUserPermission()
Retourne la permission requise pour effectuer cette action.

performAuthorizedAction(String request)
Effectue l'action avec l'information donnée dans *request*. Le nom sert a indiquer qu'elle doit être d'abord autorisée par le pont.

UserAction
La classe abstraite qui doit être implémentée par les actions qui sont demandées par le client. Elles doivent être associées à une *Command* de la base de données initialisée dans le constructeur. Il sera possible de changer la description de la commande dans le future, lorsque les détails d'implémentations seront décidés. Comme cette classe hérite de "IAction", la description des méthodes publiques ne sera pas réécrite.

Le reste des *Action*s héritent de *UserAction* et ne font que définir leur *Command* spécifique ainsi que de redéfinir *getRequiredUserPermission()* avec la permission appropriée. Nous n'avons pas utilisé de constante pour ce champ puisqu'il a été jugé plus générique d'utiliser une méthode. (Mais il serait toujours possible d'utiliser une constante qui est retournée dans la méthode.) La même solution avait été envisagée pour la *Command* associée à une action, mais il était impossible de retourner une *Command* d'un certain type sans connaître la *String* de la requête.

Notez qu'il reste plusieurs actions à implémenter, mais comme elles suivaient toutes le modèle de *UserAction*, seulement quelques unes ont été données pour montrer un exemple d'implémentation.

Module base de données

Le module qui s'occupe de gérer la base de donnée - sa création, les accès et requêtes. Il reçoit les requêtes du client qui ont été validées et autorisées par le **pont**. Le module de base de données retourne des objets *ResultSet* au pont qui les transformera en réponse de format JSON pour les renvoyer au client.

ca.diro.DataBase

Contient les classes qui gèrent les appels à la base de données.

DataBase
: Cette classe s'occupe de la création, de l'exécution de commandes et de la gestion de la base de données.

ca.diro.DataBase.Command

Contient les spécifications des commandes de base de données possibles. Comme la plupart des *Commands* ne font que remplir les fonctionnalités spécifiées pour le projet et que leurs noms sont assez descriptifs, elle ne seront pas énumérées ici. Leur but est de fournir un moyen facile d'invoquer des requêtes SQL. Ces requêtes sont ensuite enveloppées par un objet "UserAction" qui gère leurs permissions.

Command
: La classe qui doit être héritée par les commandes d'accès à la DB. Les méthodes requise pour l'exécution d'une requête SQL y sont définies.

0.2 Les librairies externes

Nous avons utilisé, en plus de "Jetty" qui nous était donné, les librairies suivantes :

org.json
: Pour gérer les requêtes et réponses dans le pont.

h2
: Qui sera la base de données que nous utiliserons.