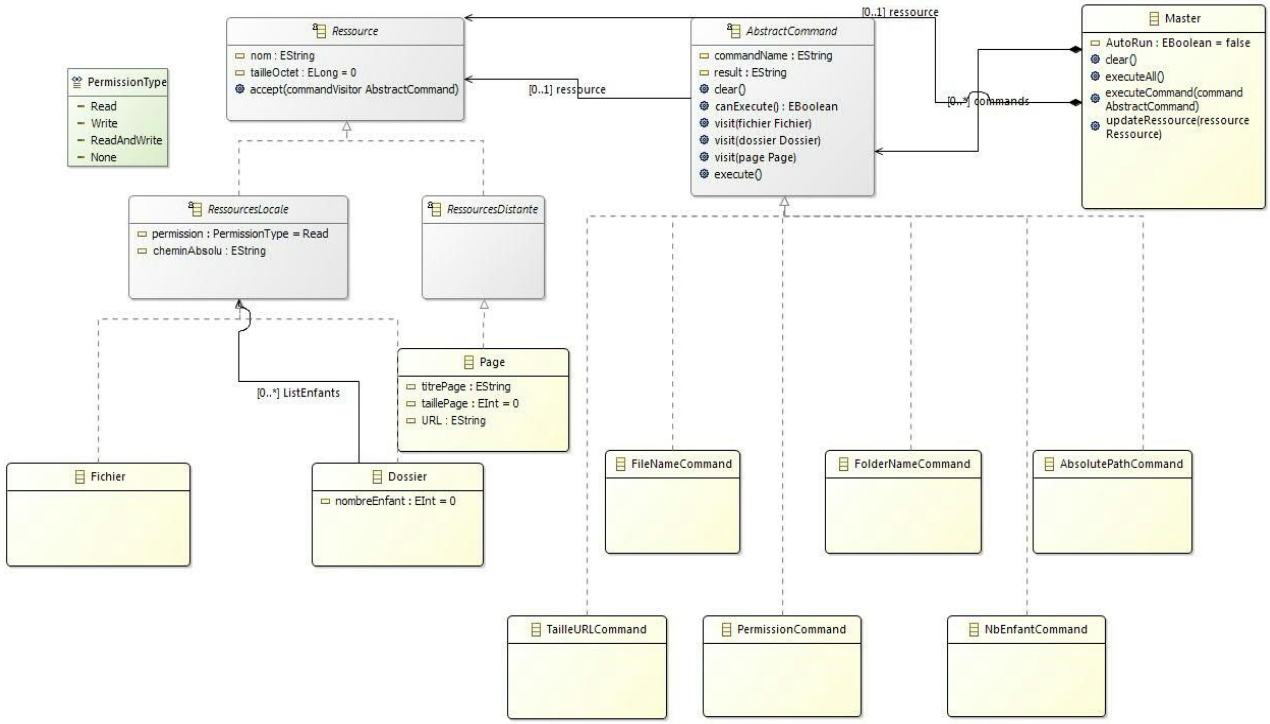
Guide du développeur
LOG8430 - TP3

Serveur Web

Auteurs:
Mathieu Laprise (1639528)
Julien Bergeron (1611134)
Mathias Varinot (1627457)
Alexandre St-Onge (1623576)

P1 Modèle de données

Nous avons utilisé le plug-in editor pour créer un fichier XML de notre modèle du TP2 avec des commandes, des pages, des fichiers et des dossiers. Ensuite, nous l'avons inséré dans le gabarit. À titre de rappel, voici notre modèle du TP2.



P2 Serveur web de fichier REST

Le serveur web a été implémenté à partir du gabarit fourni pour le tp. Il a été possible de créer un serveur web avec très peu de lignes de code grâce aux librairies fournies. La classe Activator se charge du démarage du serveur avec osgi et du chargement du modèle créé précédemment. La classe EMFHandler est utilisé comme handler du serveur pour gérer les requêtes REST. EMFHandler dérive de la classe abstraite AbstractHandler et implémenta la

méthode *handle* qui s'occupe de recevoir et de répondre au requête. C'est dans cette méthode que ce fait l'analyse de la requête pour d'abord déterminer si elle provient d'un usager valide. Si l'usager et le mot de passe est valide, la méthode s'occupe ensuite de la requête http. Dépendemment de si la requête reçu est un PUT ou un GET, le serveur effectue les modifications appropriées sur le modèle ou renvoie tout simplement l'information demandé.

P3 Serveur d'authentification

Le serveur d'authentification est très simple. L'utilisateur n'a pas connaissance que nous avons un serveur d'authentification. Il intéragit seulement avec le serveur de fichier. Nous avons modifié les requêtes précédentes du point P2 pour que l'utilisateur envoie à la fin de chaque requête deux paramètres supplémentaires : son nom et son mot de passe. Avant d'envoyer un fichier, le serveur web de fichier envoie une requête GET HTTP au serveur d'authentification en passant le nom d'utilisateur et le mot de passe en paramètre dans la requête(?user=XXX&pwd=YYY). C'est la classe AuthHandler du serveur d'authentification, avec sa méthode handle, qui s'occupe de recevoir et de valider l'authentification. Si on reçoit un réponse HTTP OK Advenant une réponse positive du serveur d'authentification, le serveur de fichier envoie à l'utilisateur ce qu'il souhaite. Sinon, il lui dit de réessayer. Vous pouvez vous référer au guide utilisateur pour plus de détails sur la syntaxe des requêtes.

P4 Ajout de ressources avec un JSON

Le serveur permet la modification du modèle par l'envoie de JSON représentant les ressources à ajouter. La librairie JSON-java ainsi qu'une classe helper nous permet de réaliser ce travail. Le serveur détecte qu'une requête PUT est envoyé et après avoir authentifié l'utilisateur, le JSON reçu est transformer en objet ressource compatible avec le modèle. Cette nouvel objet est en fait un EObject qui est peut être tout simplement ajouté au modèle existant.

P5 Retour des ressources en fonction du propriétaire

Pour ce tp, nous avons choisi d'implémenter la gestion des ressources de manière simple: seul le propriétaire de la ressource peut y accèder. Lorsqu'une requête GET est reçu, on vérifie si la ou les ressources demandées appartiennent bien au client. Si c'est le cas, les ressources lui sont envoyés, sinon rien ne lui est envoyé. Cela a nécessité, au niveau de l'implémentation, d'ajouter un attribut propriétaire au modèle EMF qui est une chaine de caractère indiquant le nom de l'usager.

Déploiement de l’application dans un environnement réel (L5)

Afin de pouvoir utiliser notre application dans un environnement réel, certaines modifications doivent être apportées. Tout d’abord, il faudrait repenser la façon dont on gère les utilisateurs de notre système. Il est évident que de garder une liste des utilisateurs avec leur mot de passe dans un fichier texte est loin d’être judicieux. Idéalement, les utilisateurs seraient sauvegardés dans une base donnée et un hash de leur mot de passe serait utilisé pour faire l’authentification. En effet, c’est complètement irresponsable de stocker les mots de passes en clairs. N’importe quel employé de l’entreprise ou pirate qui a accès au fichier voit les mots de passes des utilisateurs qui ont probablement été réutilisés sur d’autres sites comme celui de sa banque. En stockant un hash, il est difficile de faire le lien entre le mot de passe utilisé et le haché. Cela limite donc les conséquences si un vol de la base de donnée survient. Le choix de l’algorithme de hashage est un choix de conception très important. Tous les algorithmes de cryptage ne sont pas égaux. MD5 ou DES seraient de très mauvais choix et n’offriraient pas plus de sécurité qu’un mot de passe stocké en lair. Il ne faut pas non plus inventer notre propre fonction de hashage, il faut utiliser des solutions qui ont fait leurs preuves avec une haute entropie de clé. On devrait privilégier RSA 2048 bits ou Blowfish avec un salt aléatoire.

L’utilisation d’une base de donnée ou d’un autre système semblable nous permettrait de faire plus facilement des changements en lien avec les usagers, comme leur attribuer des permissions supplémentaire, changer les mots de passe et créer des nouveaux usagers. On pourrait même aller jusqu’à stocker toutes les méta-données dans une base de données No-SQL comme MongoDB si la performance à grande échelle est importante pour notre système de fichier. Elle permet de stocker directement des données sous forme d’objets JSON, ce qui serait pratique dans notre cas. La base de donnée devrait être bien protégée. Selon les principes de sécurité, les privilèges de chaque employé de l’entreprise devraient être minimums sur la base de données. Seul ceux qui en ont vraiment besoin devraient avoir des permissions élevées.

Dépendamment du genre d’environnement, il serait pertinent de vérifier si les connections avec les clients sont sécurisé (protocole https). Si elles ne le sont pas, une encryption des données avant des les envoyer au clients serait souhaitable. Évidemment, il faudrait protéger le serveur contre tous les dangers qui touchent les serveurs habituellement.

Il faudrait aussi faire une meilleure gestion de la concurrence. Un serveur doit pouvoir répondre à des requêtes provenant de multiples clients au même moment grâce au multithreading. Cependant, par exemple, il pourrait survenir des situations inattendues lorsque deux client essaient de modifier le même fichier au même moment. Des tests de concurrence seraient donc nécessaires afin de vérifier assidument si certaines structures de données requièrent des

verrous (mutex) afin de garder la cohérence des données dans un environnement hautement concurrent comme un gestionnaire de fichier. Ensuite, il faudrait faire attention aux interblocages.

Finalement, il est clair qu'un système centralisé comme celui-ci ne serait pas très résistant aux pannes, un élément majeur des systèmes Internet. Qu'arrive-t-il si le serveur d'authentification ou le serveur principal tombe en panne ? Plus rien ne fonctionne. En entreprise, il faudrait assurer la redondance. Par exemple, on pourrait avoir 5 serveurs de fichier qui communiquent ensemble pour synchroniser les données de manière à assurer la redondance. Si le serveur d'authentification tombe en panne, les serveurs de fichier se contacteront afin d'élir celui qui deviendra le nouveau serveur d'authentification. Un des serveurs de fichier prendra le relais pour faire l'authentification. Aussi, si un serveur de fichier est trop occupé par ses multiples requêtes, il pourrait transférer à un autre serveur de fichier la requête du client pour obtenir de meilleures performances.