COMPTE-RENDU DE L'ÉTUDE DE CAS AIRLINE

Auteurs : Anthonin BONNEFOY
David DUPONCHEL

21 février 2009

Table des matières

1 Couche d'accès aux données

1.1 Base de données

Une base de données hsql a été utilisée pour ce TP pour plusieurs raisons. D'une part pour sa facilité d'utilisation en cours de développement et d'autre part, pour le déploiement rapide de l'application. La connexion se fait par l'interface Connector.

Une application de conversion a été réalisée pour récupérer les données du fichier xls.

1.2 Application de conversion, le XlsParser

Ce parseur se charge de lire le fichier xls et de remplir la base hsql. Ce parseur prend 2 paramètres, un fichier sql et le fichier xls.

1.2.1 Le script sql

Le script sql va être exécuté avant l'analyse et le parse du fichier xls. Ce script se charge d'initialiser la table correspondante au xls. Les tables créées doivent correspondre aux noms des feuilles. Les champs de ces colonnes doivent correspondre aux entêtes des colonnes.

1.2.2 Le fichier xls

Une fois la base initialisée avec le script sql, le fichier xls sera parsé. A chaque ligne, les cellules sont parsées et une requête d'insertion en base est écrite et exécutée.

Au final, une base de données hsql est créée avec toute les données du xls ajoutées dedans. Cette base de données est créée dans le répertoire db et peut être utilisée comme base de données pour l'application.

1.2.3 Inconvénients

Il y a actuellement plusieurs inconvénients à cette méthode. La première est l'absence de clé étrangère. Les insertions se faisant dans un ordre quelconque, il n'est pas évident d'avoir les tuples étrangers au moment de l'insertion. Cela pourrait se résoudre en ignorant les clés étrangères le temps du parse du fichier xls.

Le deuxième inconvénient est le fait de devoir supprimer les tables concernées avant d'effectuer les insertions, les problèmes de clés primaires étant délicat a résoudre. De plus, c'est la manière la plus simple pour être sûr d'être synchronisé par rapport au xls.

1.3 Représentation des tables

Le serveur inspecte la base de données et récupère des informations sur les tables présentes. Trois modèles sont utilisés pour représenter la base de données :

1.3.1 Table

Représente une table et contient le nom, le schéma et le type de la table.

1.3.2 TableColumn

Représente une colonne d'une table. Elle contient le nom, la table, le type de valeur et si la colonne est une clé primaire.

1.3.3 TableRow

Représente une ligne d'une table. Pour plus de facilité, les valeurs des champs sont considérées comme étant des String. Ce modèle lie donc à chaque colonne de la table un String pour une ligne.

1.4 Interaction avec les tables

L'interface AirlineDAO est un Data Access Object (DAO) permettant de récupérer toutes ces informations et d'exécuter des requêtes.

1.4.1 Consultation des tables

On a ainsi les méthodes
`[Map<String, Table> getTables()]` Récupère l'ensemble des tables.
`[List<TableColumn> getTablesColumns(Table table)]` Récupère l'ensemble des colonnes d'une table données.
`[List<TableRow> getTablesRows(Table table)]` Récupère l'ensemble des lignes d'une table.
Ces trois méthodes suffisent pour consulter l'intégralité des tables.

1.4.2 Exécution des requêtes

Les requêtes sont exécutées par deux méthodes
`[Set<TableRow> executeRequest(SelectRequest selectRequest) throws SQLException]` Cette méthode exécute une requête select et retourne le résultat.
`[void executeRequest(IRequest request) throws SQLException]` Cette méthode se contente d'exécuter la requête.
Dans les deux cas, une exception est renvoyée en cas d'erreur.
Les requêtes sont crées à partir de modèles Request. Nous allons voir comment sont élaborées ces Request.

1.5 La création des requêtes sql

La majeur partie du travail sur l'accès et la manipulation des données est de traiter les requêtes sql. Il était intéressant de développer une API permettant de générer des requêtes dynamiquement et présentant une couche d'abstraction au langage sql.
Pour cela, nous nous sommes inspirées de l'API Criteria de Hibernate. Une API similaire sera introduite dans la prochaine version de JPA. Cette API présente deux composant principaux, les Requests et les Restrictions.

1.5.1 Les Requests

Les Request représentent les requêtes a exécuter. Elles implémentent l'interface Request qui possède la méthode buildQuery(). Il suffit donc d'exécuter le résultat de cette méthode par le SGBD.
Le résultat de buildQuery est construit à partir du type de la Request et de ses informations. Par exemple, `DropTableRequest` contient simplement la table à supprimer. Elle créera donc une requête de type `DROP TABLE table`. Les autres requêtes traitent des cas plus complexes comme le choix des colonnes ou la modification des champs. Les clauses `WHERE` des requêtes sont gérés par des Restrictions.

1.5.2 Les Restrictions

Les Restrictions sont des contraintes qui symbolisent les clauses `WHERE`. On peut ainsi écrire différentes contraintes :

- Les contraintes simples. Il s'agit soit de la simple comparaison entre une colonne et une valeur ou entre deux colonnes pour les jointures.
- Les contraintes conditionnelles. Une Restriction peut lier deux autre Restrictions avec les conditions OR ou AND.

On obtient au final une Restriction qui regroupe une arborescence de Restriction. Il suffit d'ajouter restriction.toString() à la fin de la request pour rajouter les clauses WHERE.

Ce système de restriction est utilisé pour les requêtes de sélection, de suppression et de mise à jour.

1.6 L'enregistrement des transactions

Les transactions sont enregistrées dans la table Transaction. Elles sont enregistrées grâce au AirlineManager. Les Jsp et servlet utilisent le manager pour communiquer avec le serveur et ce manager appelle le DAO pour répondre aux requêtes. C'est au moment de l'exécution de requêtes que la transaction est enregistrée par le manager.

2 Interface utilisateur

2.1 Architecture

Notre projet a été architecturé en respectant l'architecture standard java :

```
src
 |--test  // tests unitaires
 |--main
 |    |--java    // code source java
 |    |--webapp  // fichiers jsp
```

2.1.1 Servlet et jsp

Toute la partie traitement des requêtes web et envoi de la réponse au navigateur utilise une servlet comme contrôleur et des fichiers jsp comme vue. Le fichier web.xml permet de spécifier quelle servlet est appelée pour quelle url. Ce fichier défini également les filtres à appliquer, les éventuels listeners, les fichiers à afficher par défaut, ou encore les pages d'erreurs à afficher.

Afin de centraliser les éléments récurrents des pages (entête du html, menu de la page), des fichiers header.jsp et footer.jsp ont été utilisés.

Voici le déroulement d'une requête :

1. le navigateur envoie une requête
2. les filtres vérifient la validité de la demande
3. la servlet (contrôleur) interprète la requête et la traduit en appels au DAO (modèle)
4. la servlet envoie les résultats à une page jsp (vue) qui les affiche

2.1.2 Filtre AdminFilter

Pour vérifier si un utilisateur est identifié par le site quand il demande une action de modification, un filtre est utilisé. Ce filtre est actif sur tout le site et redirige l'utilisateur vers la page de login s'il n'a pas les droits nécessaires.

Le mot de passe est stocké dans un fichier de propriétés, AuthDAO.properties. Par défaut, le login est admin et le mot de passe est adminadmin.

2.1.3 Injection des dépendances dans les servlets

L'injection de dépendances avec Guice ne peut pas être faite de façon classique, car les instances des servlets sont créées par le serveur applicatif. Afin de palier ce problème, nous utilisons le `ServletContext`. Cette interface définie un ensemble de méthodes utilisables par les servlets pour communiquer avec son conteneur. Il existe un seul contexte par application web (et un seul par JVM par application web dans le cas d'une application web distribuée). Lorsque le contexte est initialisé, un injecteur est créé et stocké dans le contexte de la servlet. Lorsqu'une servlet est instanciée, elle utilise ce contexte pour récupérer l'injecteur et injecter ses dépendances.

2.1.4 Respect des standards et pérennité de airline

Garantir la pérennité d'une application web et son comportement dans un navigateur est crucial. C'est dans ce but que nous avons fait le choix d'utiliser des technologies récentes et ouvertes, ainsi qu'un encodage universel.

Les pages générées sont en XHTML 1.1 avec des feuilles de style CSS. Toutes les pages et les feuilles de style ont été validées pour le validateur du W3C, organisme qui définit notamment les

standards ouverts du web. Les navigateurs évoluant vers toujours plus de respect des standards, les respecter présente un énorme avantage pour la pérennité de cette application.

Pour éviter tout problème d'encodage et simplifier l'internationalisation de notre application, cette dernière utilise exclusivement l'encodage unicode.

Le design du site a été pensé en gardant l'esprit d'ouverture et l'amour du libre qui a animé le reste du travail. Les images utilisées sont soit sous licence LGPL et issues du monde libre, soit sont sous une licence Creative Commons permettant la modification de l'image. Ainsi, nous sommes sûr d'utiliser des images en respectant l'auteur et les conditions d'utilisations. A moins de commercialiser airline (autorisé par sa licence libre Apache), l'utilisation de ces images est parfaitement légale.

2.2 Select From Where

Cette partie permet à l'utilisateur d'effectuer une requête SELECT sur la table de son choix. Elle consiste en une seule page présentant un formulaire qui affiche les informations demandées. Dès que l'utilisateur modifie un champ du formulaire, un court code javascript le détecte et recharge la page, affichant ainsi les nouvelles informations. La première étape est de choisir la table, puis le ou les champ(s) à afficher. Enfin, l'utilisateur spécifie une contrainte avec le champ where.

Puisque ce formulaire ne modifie pas l'état de la base de données (on la consulte juste), on utilise la méthode GET au lieu de la méthode POST, conformément aux recommandations du W3C.

2.3 Tables

2.3.1 Url rewriting

La réécriture d'URL permet d'interpréter une url et de la modifier à la volée. Cela permet d'avoir des urls plus courtes et plus compréhensibles. Exemple de réécriture d'url avec un serveur Apache :
une url `localhost/airline/user/martin/showprofile` avec le filtre
`RewriteRule ^airline/user/([a-zA-Z]+)/([a-z]+)$ airline/userServlet?login=$1&action=$2`
deviendra `localhost/airline/userServlet?login=martin&action=showprofile`.
Dans le code côté serveur, tout sera comme si l'utilisateur avait appelé
`localhost/airline/user?login=martin&action=showprofile`
L'utilisateur, de son point de vue, interagira avec la page
`localhost/airline/user/martin/showprofile`
beaucoup plus propre et parlante.

Cependant, cette technique n'est pas implémentée de base dans le système de servlet/jsp. Pour pouvoir utiliser cette technique, nous sommes donc passé par un filtre qui analyse les appels à la partie admin/table et analyse l'url appelée pour en déduire le contexte, les actions, et les entités sélectionnées.

2.3.2 Notions de contexte, action

La notion de verbe (action à effectuer) existe déjà avec HTTP (définie par la RFC 2616). Seulement, les sites webs utilisent (les navigateurs également) exclusivement les verbes HTTP GET et POST. Ne pouvant pas utiliser les autres verbes (HEAD, PUT, DELETE, etc) on ajoute l'action demandée à la fin de l'url (par défaut show). On définie également une notion de contexte (aucun rapport avec le contexte de la servlet) pour définir ce sur quoi porte l'action. En ajoutant la réécriture d'url à ces notions, on aboutit à ça :

- /table/add : ajoute une table
- /table/FOOBAR : affiche (show par défaut) la table foobar
- /table/FOOBAR/field/ID/edit : modifie la colonne id de la table foobar
- /table/FOOBAR/row/4/delete : supprime la 4ème ligne de la table foobar

La méthode GET est utilisée pour afficher le détail de l'opération ou faire l'affichage demandé, tandis que la méthode POST est utilisée pour effectuer l'action demandée si cela modifie le serveur.

Ce résultat permet de profiter d'url très propres et parlantes pour l'utilisateur, mais permettra également de faciliter le développement d'un service web à partir du code existant. En effet, un web service RESTful utilise ces notions : l'action correspond a la méthode HTTP, et la notion de contexte correspond a une ressource web. Tout est donc prêt ou presque pour ajouter ces fonctionnalités.

Concrètement, voici ce qu'il se passe lorsqu'une ressource est appelée :

- la requête passe par le filtre de réécriture d'url. L'action, le contexte, la table visée, etc... sont extraites et stockés dans la requête. Si l'url est incohérente, une page d'erreur est aussitôt envoyée.
- la requête passe dans un second filtre, vérifiant la validité des informations : existence de la table, existence du champ, etc. En cas de problème, l'erreur est affichée à l'utilisateur.
- la servlet est finalement appelée : elle dispose de toutes les informations nécessaires, et elle est sûre d'être dans un cas cohérent et vérifié.

2.3.3 Polymorphisme

Pour éviter l'utilisation de nombreux blocs switch/case dans le code (il y a 2 méthodes HTTP, 4 contextes et 4 actions possibles, soit virtuellement 32 cas possibles) le polymorphisme a été utilisé. Nous avons créé des classes spécialisées chacune dans un cas spécifique et implémentant toutes la même interface. Ces classes contiennent le code nécessaire à la vérification du contexte (méthode checkContext) et à l'exécution de l'action (méthodes post et get). Dès lors, le filtre vérifiant le contexte a juste à appeler la méthode checkContext. De même, une fois dans la servlet, cette dernière a juste à appeler les méthodes get et post. Au final, le code est regroupé par type d'action et par type d'entités, et le code dans les filtres/servlet a été grandement simplifié.

2.3.4 Actions possibles

L'application web propose de nombreuses possibilités : un utilisateur non identifié va pouvoir voir toutes les tables, leurs détails et leurs contenus. S'il est identifié comme administrateur, il pourra ajouter/supprimer des tables, ajouter/modifier/supprimer des champs dans une table, ajouter/modifier/supprimer une entrée dans une table.

3 Limitations et problèmes rencontrés

3.1 Les problèmes

3.1.1 Typage des Objets récupérés

Il est très difficile de gérer la généricité en partant des résultats de la requête select. Il a fallut gérer les valeurs en tant qu'objet ce qui n'est pas forcément judicieux. Mais le fait de traiter les tables de manière dynamique oblige à procéder de cette façon.

Il aurait été plus facile de traiter ce problème avec un langage dynamique. Dans ce cas là, il aurait été envisageable de créer à la volée les classes correspondantes aux tables.

3.1.2 Construction des requêtes

Le langage SQL n'est pas réputé pour sa simplicité d'utilisation. Les SGBD sont relationnels et le java est un langage objet, il y a donc toujours un certain nombre de difficultés pour la correspondance entre un tuple SQL et sa représentation en Java. La création des query à demandé un travail conséquent et les possibilités offertes par le programme sont limitées.

3.1.3 Le traitement des mdb et xls

La transcription du fichier mdb en script SQL n'est pas aisée. En effet, il n'est pas possible d'exporter la base et le format mdb n'est pas facilement exploitable. De plus, les champs de la base mdb et les entêtes du xls ne correspondent pas. Au final, la table créée correspond à une vue du xls.

3.1.4 Servlets

Un problème rencontré lors du développement avec des servlets/jsp a été les notions flous que nous avions concernant ces technologies. En particulier, on peut citer l'utilisation du ServletContext, l'utilisation d'un servlet comme contrôleur et d'une jsp comme vue, ou la façon de transmettre des données à la vue.

3.1.5 Passage de paramètres "typesafe"

Lorsque l'on passe des paramètres d'une servlet à une jsp, on passe soit par la session soit par la requête. Or dans les deux cas, on récupère un objet de type Object à partir d'une chaîne de caractères. Ces deux points posent problème : si le développeur fait une faute de frappe ou si un autre développeur utilise le même nom comme "clef", retrouver la source du bug peut devenir cauchemardesque. Le transtypage obligatoire peut également poser problème : on suppose connaître à l'avance le type d'objet récupéré. Si ce n'est pas le cas, l'application récupèrera une exception, et aucun objet exploitable.

3.1.6 Composants jsp

Le dernier point problématique est l'absence de composants près à l'emploi, comme des formulaires, etc.

3.2 Les limitations

3.2.1 Caractères non ASCII

L'utilisation de lettres entre a et z ne pose ancun problème. L'utilisation de caractères accentués va poser problème avec l'encodage et la base de données. La requête SQL a alors de grandes chances

d'échouer, comme avec l'utilisation de ' ou de \.

3.2.2 Requêtes sql limitées

Les requêtes sql réalisables sont limitées par l'implémentation et l'interface que nous avons réalisées. Il n'est pas possible de faire des jointures entre les tables par exemple.

3.2.3 La sécurité

La sécurité de l'application laisse à désirer. Il n'y a aucune protection sur les requêtes sql et n'importe quel utilisateur peut faire une attaque par injection sql. Par exemple, il peut taper dans la clause WHERE.

```
';DROP TABLE APPAREL;
```

Cela aura pour effet d'exécuter un DROP sur la table APPAREL.

Annexes

Utilisation

Le lancement de l'application se fait en 2 temps.

Récupération des données du xls

La première étape facultative consiste à extraire les informations du xls. Pour cela, il faut lancer le jar xlsParser.jar avec script.sql et AirLineData.xls en paramètre.

```
java -jar xlsParser.jar script.sql AirLineData.xls
```

Un repertoire db sera créé contenant le résultat.

Lancement du serveur avec l'application

L'application peut être lancé simplement en exécutant le jar glassfish-web.jar avec le war de l'application en paramètre.

```
java -jar glassfish-web.jar airline.war
```

Si le répertoire db est présent, la base présente sera utilisée. Sinon, il créera sa propre base.

Le serveur est alors accessible sur l'url http ://localhost :8888/airline/