ISIMA
Institut Supérieur d'Informatique
de Modélisation et de leurs Applications

Campus des Cézeaux
24 Av. des Landais
BP 10125
63173 Aubiere CEDEX

Rapport d'intégration d'application
Filière Génie Logiciel et Systèmes Informatiques

VALORISATION D'UNE BASE DE DONNÉES AVEC GOOGLE APP ENGINE ET GOOGLE WEB TOOLKIT

Étudiants : Antoine Colmard et Nicolas Prugne

Antoine Colmard et Nicolas Prugne : *Rapport d'intégration d'application,* Valorisation d'une base de données avec Google App Engine et Google Web Toolkit, Décembre 2014

TABLE DES FIGURES

TABLE DES MATIÈRES

1

INTRODUCTION

2

SERVEUR GWT

Pour rappel, le logiciel développé au cours de ce TP concerne la mise en œuvre d'un service de gestion de livres centralisé. Cette application est donc divisée en deux parties. D'un côté, se trouve une application cliente, celle que les utilisateurs finaux utiliseront et de l'autre se trouve une application serveur. Cette dernière doit offrir le support de tous les mécanismes de gestion des livres de manière centrale.

Ces deux outils ont vocation à s'exécuter sur des machines différentes mais doivent tout de même pouvoir dialoguer à travers le réseau. Le client doit pouvoir appeler des méthodes implémentées par le serveur, et ce dernier doit pouvoir lui retourner des résultats. Il s'agit la de mettre en place un système permettant au client de faire des RPC (Remote Procedure Call) sur le serveur.

Lorsque ce dernier recevra un de ces appels distant, il devra alors traiter la requête et la retourner au client. Le traitement implique bien entendu l'interaction avec une base de données implantées côté serveur.

C'est pourquoi, dans cette partie, il sera tout d'abord question de découvrir l'architecture de la base de données utilisées par le serveur. Puis, viendra ensuite une partie sur le support des RPC que propose GWT. Enfin, un exemple de mise en œuvre de ces RPC viendra compléter les explications de la partie précédente.

2.1 Gestion de la base de données

Par rapport au TP précédant, l'architecture de la base n'a pas du tout évolué, c'est pourquoi cette partie ne sera constituée que de rappels faisant référence à des explications données antérieurement.

Ainsi, la base s'appuie toujours sur le service NoSQL de Google appelé Datastore. L'interaction avec ce dernier ne s'effectue par directement avec l'API bas niveau que

propose Google, mais avec la bibliothèque Objectify. Cette bibliothèque Java permet en quelque sorte de faire de l'ORM avec le Datastore.

De ce fait, deux classes ont été modélisées (figure 1). Celles-ci sont rendues persistantes grâce à quelques annotations placées dans le code de leurs déclarations. Ainsi, une instance de type Livre ou de type Auteur peut être stockée dans le Datastore en quelques lignes grâce à l'API d'Objectify.

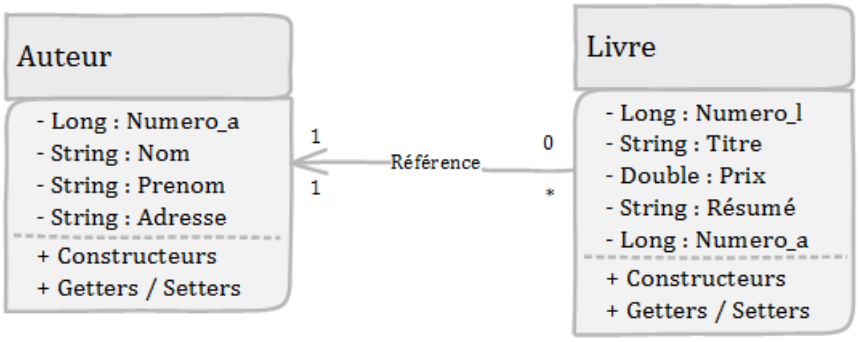

FIGURE 1: Classes métiers persistantes

Comme le NoSQL ne fournit pas de mécanisme permettant de lier deux types d'entités, à la manière des clefs étrangère en SQL, le lien entre les livres et les auteurs doit être géré manuellement (via du code) par les développeurs.

Une fois que le problème d'interaction avec la base est résolu, il faut prévoir toute la logique applicative que le serveur va devoir fournir aux clients. Cette logique repose sur un ensemble de méthodes exposées aux clients, et que ces derniers peuvent appeler de manière distantes. La partie suivante explique ce procédé dans le cadre d'une application GWT.

2.2 SUPPORT DES RPC AVEC GWT

Une application GWT est généralement consistuée de deux choses, une partie cliente et une partie serveur. La partie cliente va être exécutée via le navigateur web de l'utilisateur, tandis que la partie serveur, sera exécutée sur un serveur Java EE destiné à servir les clients.

Ainsi, pour communiquer, le client va devoir effectuer des requêtes HTTP sur le réseau pour accéder aux services offerts par le serveur. Le traitement de ces requêtes par le serveur s'effectue via une servlet, mais une servlet un peu particulière.

En effet, GWT fournit une classe spécifique à son support des RPC baptisée `RemoteServiceServle`
Lorsqu'un développeur souhaite implémenter la partie qui va traiter les RPC de son application, il doit commencer par créer une classe qui va hériter de ce composant.

Ensuite, pour que le client et le serveur s'entendent sur l'ensemble des méthodes exposées, il faut définir une première première interface que la servlet devra implémentée. Ici, elle est baptisée, `BiblioService` (figure 2).

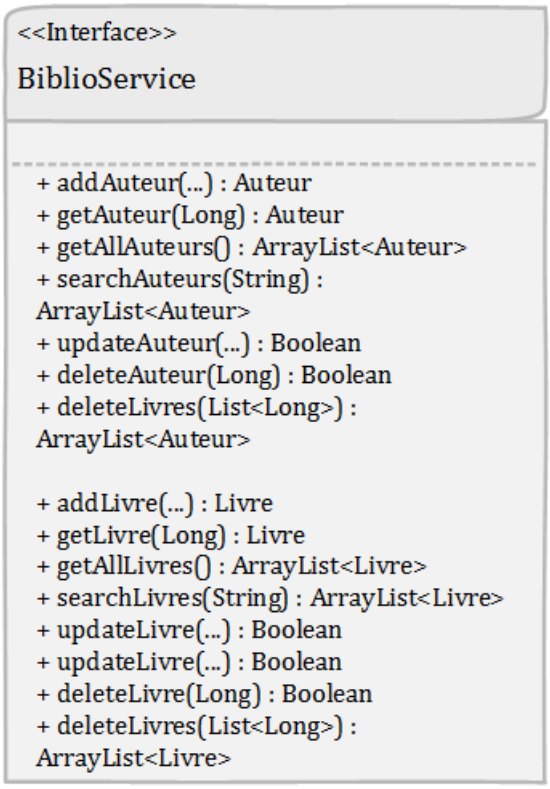

FIGURE 2: Interface BiblioService

Mais ce n'est pas tout, l'une des particularité de GWT est que le support des RPC est asynchrone, c'est-à-dire qu'un client qui fait appel à l'une des méthodes exposées par le serveur ne sera jamais bloqué par celle-ci. Par conséquent, il est nécessaire de définir une seconde interface dite asynchrone, qui reprend les prototypes des méthodes de `BiblioService` en les modifiant légèrement (figure 3).

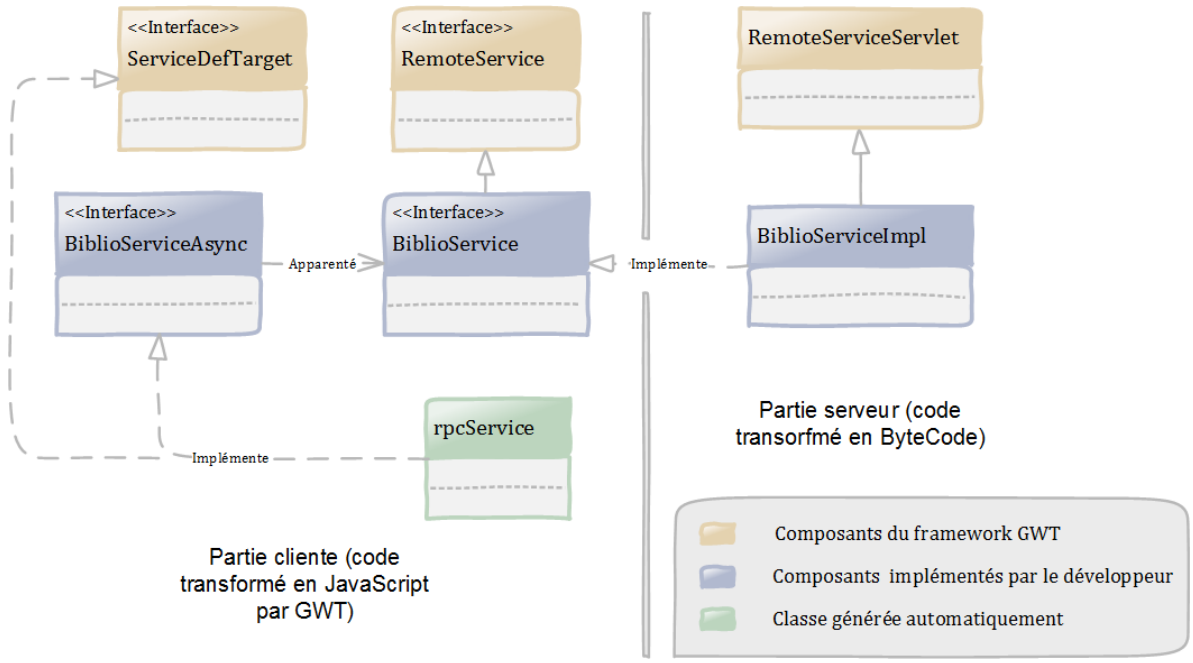

FIGURE 3: Architecture RPC

Dans l'implémentation réelle de l'application, la frontière de communication entre le client et le serveur est définie par l'interface `BiblioService`. A partir de cette dernière, l'interface asynchrone qu'utilise le client peut elle aussi être définie. Elle porte le nom de `BiblioServiceAsync`. Côté serveur, l'implémentation de la remote servlet est réalisée par la classe `BilbioServiceImpl` qui au passage implémente aussi l'interface `BiblioService`.

Seules ces trois classes sont nécessaires à la mise en place de RPC avec le framework GWT. A partir de ce moment, il ne reste plus qu'à lancer le serveur Java EE en chargeant la servlet codée précédemment, et côté client, il suffit d'instancier une classe qui fera office de proxy avec le serveur, baptisée ici `rpcService`. Cette classe n'a pas besoin d'être implémentée, elle est directement créée à la volée par le framework en se basant sur les méthodes de l'interface `BiblioService`.

2.3 WORKFLOW DU MÉCANISME DES RPC AVEC GWT

La partie précédente a permis de révéler l'architecture pour le support des RPC dans sa globalité, mais elle n'explique en rien le déroulement d'un de ces appels.

Tout d'abord, il faut préciser un point important du système. Le client et le serveur vont bien entendu devoir s'échanger des données afin de réaliser leurs tâches respectives. Dans le TP précédant, cet échange été réalisé grâce au protocole SOAP. Ici, il n'en n'est rien. GWT permet aux deux applications, cliente et serveur, de s'échanger directement des objets Java. La seule contrainte imposée étant que ces objets soient sérialisable.

Dans le cadre de cette application, les principaux objets que vont s'échanger les deux parties du système vont être des objets de type Livre ou de type Auteur. Voilà pourquoi ces deux classes doivent être déclarées sérialisable, à l'aide d'une annotation placée dans le code de leurs déclarations respectives. Dès lors, même un échange d'une liste de livres ou d'auteurs ne pose pas de problèmes car celui-ci s'effectuera grâce à une instance d'`ArrayList` elle même sérialisable.

Une fois ce problème résolu, le flux de données qui circule entre le client et le serveur est très simple à comprendre. Tout d'abord, le client va envoyer une requête au serveur par l'intermédiaire du composant `rpcService` vu précédemment. Ce dernier, va transmettre la requête HTTP au serveur et rendre la main au client. De cette manière, l'interface du client ne sera jamais gelée en attendant la réponse du serveur. Pendant ce temps là, la classe `BiblioServiceImpl` va se charger d'effectuer la requête, comme par exemple, charger la liste des auteurs présents dans la base de données. Une fois le calcule de la requête terminée, il va retourner un `ArrayList` d'auteurs que le framework va se charger de sérialisabler et de renvoyer au client (figure 4).

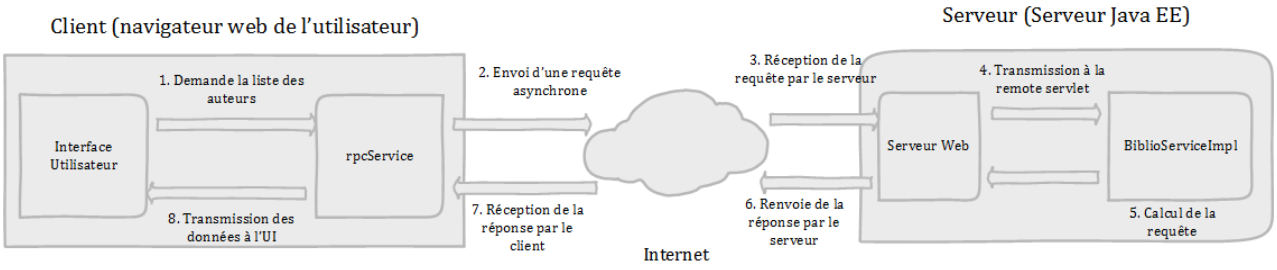

FIGURE 4: Demande de la liste des auteurs via le mécanisme des RPC

Comme le composant `rpcService` implémente l'interface asynchrone de `BiblioService`, il est capable, lorsqu'une réponse en provenance du serveur arrive, de transmettre les

informations contenues dans cette réponse au client. L'interface du client n'a plus qu'à être rafraichie, et les données en provenance du serveur se retrouvent affichées dans le navigateur de l'utilisateur. Ce mécanisme repose en réalité sur l'échange de requêtes AJAX entre le client et le serveur, mais la complexité de ce type de fonctionnement est masqué par l'API de GWT.

3

CLIENT GWT

La force majeure de GWT réside dans l'implémentation de la partie cliente d'une application. En effet, celle-ci s'effectue en Java, ce qui permet aux développeurs à l'aise avec cet environnement de ne pas être perdus. Néanmoins, même si c'est en Java qu'une application GWT se code, c'est bien du Javascript qui va finalement s'exécuter chez le client. Le framework permet de traduire un code Java en son équivalent JavaScript. Il est possible de voir GWT comme un compilateur qui traduirait le code de haut niveau (Java), d'une application en code bas niveau, ici représentait par le JavaScript.

Dès lors, le client se code entièrement comme une application Java standard, notamment la partie graphique, dont la syntaxe se rapproche grandement de l'API Swing. De ce fait, certaines techniques utilisées pour le développement d'une application Java standard sont transposables pour une application GWT. Parmi ces techniques on retrouve la mise en place de design patterns permettant de séparer l'affichage du traitement des données, comme le modèle MVC, ou encore la mise en œuvre de tests unitaires.

Parmi les architectures possibles pour implémenter la partie cliente, il en est une qui prédomine avec GWT, c'est le modèle MVP (Model-View-Presenter). La partie suivante a pour but de mieux présenter ce modèle, ainsi que sa mise en œuvre dans la réalisation de ce TP.

3.1 L'architecture MVP

Avant de présenter le modèle, il faut expliquer les motivations qui ont conduit à son utilisation. Tout d'abord, comme le MVC, il permet de fortement découpler l'affichage des données, de leurs logiques de traitement. Ainsi, il facilite le travail en équipe des

développeurs, dans le sens où chacun peut se concentrer sur le développement d'une des parties du modèle sans influer sur les autres.

La seconde raison qui joue en la faveur du MVP est la facilité qu'il procure pour la mise en place de tests unitaires, ce qui pour une application web est bien souvent source de problèmes.

L'architecture MVP repose sur quatre composant qui vont être expliqués par la suite. Un modèle, une vue, un présentateur et un contrôleur.

3.1.1 Le Modèle

Le modèle est représenté par l'ensemble des classes métiers de l'application. Dans le cas de ce TP, il s'agit bien entendu des classes Auteur et Livre.

3.1.2 La vue

La vue, ou plutôt les vues vont regrouper toutes les classes qui définissent une page web affichable de l'application. Elles sont constituées d'un ensemble de composants graphiques GWT qui permettent de concevoir une UI. Il s'agit des boutons, des barres de textes, ou encore des tableaux affichables dans la fenêtre d'un navigateur web par exemple. Toutes la mise en place de ces composants s'effectue au sein des vues de l'application. Les vues n'ont aucune connaissance du modèle, elles ont pour seul rôle d'afficher des composants graphiques au sein de pages web.

3.1.3 Le présentateur

Chaque vue est accompagnée d'un autre composant appelé présentateur (presenter). Le présentateur va avoir pour rôle de lier la vue au modèle de l'application. Chacun des présentateurs, associé à une vue, contient la logique applicative qui se cache derrière celle-ci. Cette logique inclut la gestion des événements lorsqu'un utilisateur interagit avec la vue, mais aussi la gestion des RPC qui permettent de synchroniser le contenu de celle-ci avec les données du serveur.

3.1.4 Le contrôleur

Le dernier composant à présenter est le contrôleur. Il est en quelque sorte le chef d'orchestre de l'application. Son travail consiste à gérer tout ce que ne peut être fait dans le code des présentateurs, à savoir, la gestion de l'historique de navigation entre les différentes vues de l'application et la gestion des transitions entre celles-ci.

Chaque partie du modèle à donc un but bien précis. Cette décomposition permet de limiter le couplage entre les différents composants de l'application ce qui aboutit a un développement plus robuste et plus efficace en équipe.

La section suivante présente l'implémentation réelle de cette architecture pour l'application de gestion de livres du TP.

3.2 Mise en oeuvre du modèle MVP

Tout d'abord, cet outil de gestion de livres ne requière que trois vue. Une première permettant d'afficher une liste d'auteurs, une seconde permettant l'édition d'un auteur, et enfin une troisième dans le but d'éditer les informations d'un livre. Avec ces trois vues viennent également trois présentateur. La liaison vue / présentateur requière alors de définir une interface commune entre les deux composants. Cette interface est définie et imbriquée dans le code de chaque présentateur (figure 5).

Ces interfaces imbriquées, appelées `Display`, permettent de définir de manière abstraite ce que chaque présentateur s'attend à trouver, dans la vue qui lui est liée.

Par exemple, le présentateur `AuteurPresenter` s'attend à ce que la vue qu'il contrôle présente un bouton permettant d'ajouter un nouvel auteur. Il le fait donc savoir en exposant dans son interface `Display`, une méthode `getAddButton` dont le type de retour est `HasClickHandlers`. Ce type de retour particulier fait partie d'une collection d'interfaces définies par l'API de GWT. Les composants graphiques qui l'implémentent sont obligés de fournir un moyen pour enregistrer un `clickHandler` à leurs listes de handlers.

Ainsi, la vue `AuteurView`, qui implémente l'interface `Display` du présentateur `AuteurPresenter`, sait qu'elle doit fournir un accesseur retournant une référence sur un composant graphique qui propose une méthode pour enregistrer un `ClickHandler`. Le présentateur

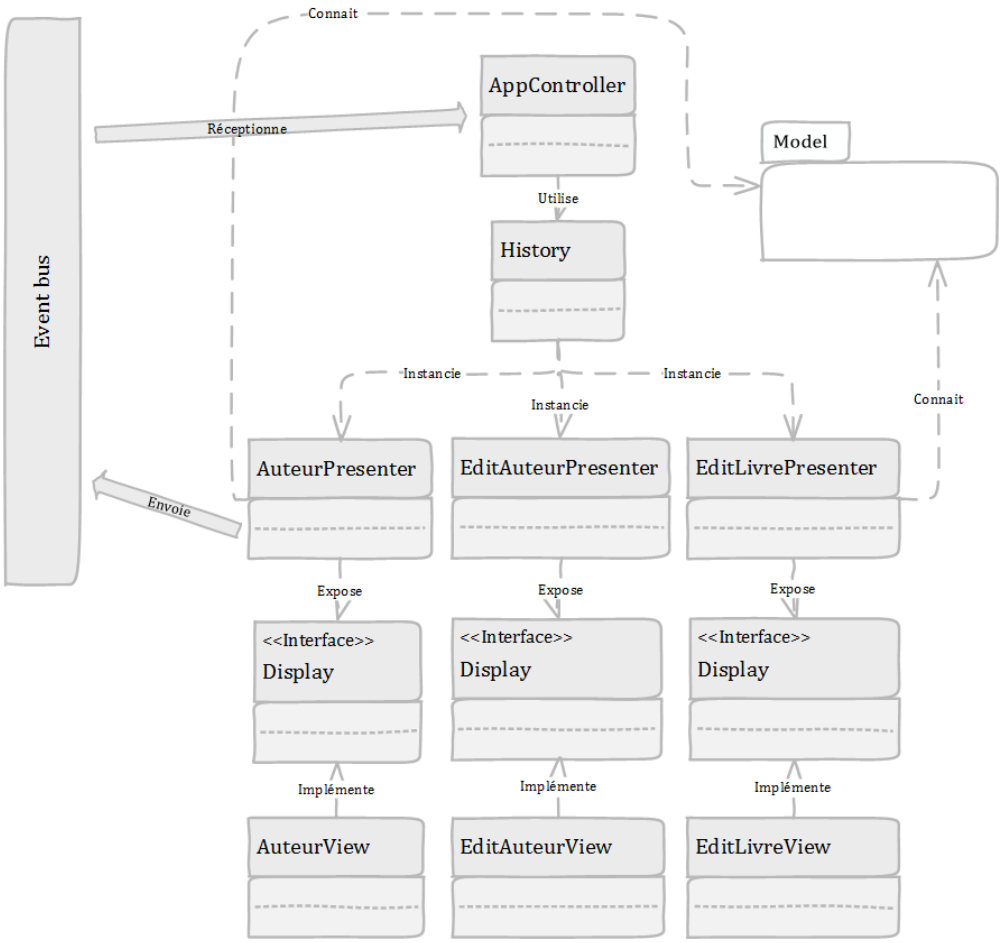

FIGURE 5: Architecture MVP

pourra alors enregistrer son propre handler sur ce composant et faire le traitement attendu lorsque l'utilisateur cliquera dessus.

Ce système de contrats passés entre les présentateurs et les vues permet de faciliter l'interchangeabilité de ces dernières. En effet, il alors facile de proposer une vue adaptée au terminal sur lequel navigue l'utilisateur, pour peu que celle-ci implémente l'interface.

Un autre point important du modèle MVP concerne la gestion des évènements qui ne peuvent être traités par les présentateurs. Bien souvent, il s'agit d'un évènement qui implique de changer de vue. Par exemple, lorsque l'utilisateur souhaite ajouter un nouvel auteur, il va cliquer sur le bouton `Ajouter`, et la page d'ajout d'un auteur va être chargée.

Pour ce genre d'évènements, le modèle prévoit un composant utilisé par tous les présentateur : le bus d'évènement. Pour reprendre l'exemple précédant, lorsque le présentateur `AuteurPresenter` détecte un appuie sur le bouton ajouter, il va émettre un évènement sur le bus qui va être transmis au contrôleur de l'application. Ces évènements sont matérialisés par des classes qui héritent de `GwtEvent`. Au préalable, le contrôleur à enregistrer l'ensemble des évènements qu'il est capable de traiter, et lorsqu'il en reçoit, il exécute l'opération adéquate.

Concrètement, ce qui se passe au niveau du contrôleur implique un dernier composant nommé `History`. Celui-ci se comporte comme une pile et va stocker des jetons qui représentent l'historique de navigation de l'utilisateur dans l'application. A chaque fois qu'un nouveau jeton va être ajouté à l'historique, la méthode `onValueChange` du contrôleur va être appelée. C'est au niveau de cette dernière que l'instanciation du nouveau présentateur demandé et la nouvelle vue à charger vont avoir lieu.

Ainsi, pour résumer, le contrôleur va en permanence écouter le bus d'évènements. Dès lors qu'un évènement enregistré se produit, le contrôleur va ajouter un nouveau jeton à l'historique pour indiquer et mémoriser un changement de vue. Par la suite, la méthode `onValueChange` va être appelée, et cette dernière va réaliser l'instanciation de la nouvelle vue et du présentateur associé.

4

RÉSULTATS ET TESTS

5

CONCLUSION