



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
FILIERE GÉNIE LOGICIEL ET SYSTÈMES INFORMATIQUES

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

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 client web, Décembre 2014

TABLE DES FIGURES

FIGURE 1	Exemple d'entité du Datastore	3
FIGURE 2	Les classes métiers	5
FIGURE 3	Implémentation du lien entre les classes Livre et Auteur	5
FIGURE 4	Interface du service web	8
FIGURE 5	Classe Author	12
FIGURE 6	Classe Book	13
FIGURE 7	Vue principale	13
FIGURE 8	Bouton d'ajout	14
FIGURE 9	Ajout d'un auteur	14
FIGURE 10	Auteur modifié	14
FIGURE 11	Ajout d'un livre	15
FIGURE 12	Recherche d'un auteur pour un livre	15
FIGURE 13	Livre modifié	16
FIGURE 14	Recherche sans paramètre	16
FIGURE 15	Recherche avec paramètre	16

TABLE DES MATIÈRES

1	INTRODUCTION	1
2	STRUCTURE DE LA BASE	2
2.1	Le Datastore	2
2.2	Accès aux données	3
2.3	Modélisation et implémentation	4
3	SERVEUR ET WEB SERVICES	7
3.1	Interface du service web	7
3.2	Support des RPC	9
4	CLIENT DE L'APPLICATION	11
4.1	Choix du client	11
4.2	Structure du client	12
4.2.1	Design	12
4.2.2	Modèle	12
4.2.3	Fonctionnement	13
4.3	Communication avec le serveur	16
5	RÉSULTATS ET TESTS	18
6	CONCLUSION	19

INTRODUCTION

À l'ère du *Big Data*, la majorité des applications informatiques recueillent des quantités importantes de données au sein de leurs bases. La problématique de la valorisation de ces données entre alors en compte. Cette valorisation passe dans un premier en la création d'une application serveur pouvant communiquer avec la base, puis en l'exposition des méthodes de cette application pour que des applications client puissent interagir avec la base de données. L'utilisation de *web service* permet de rendre l'application disponible n'importe où et permet une pluralité des applications client pouvant communiquer avec le serveur.

Le travail demandé ici est de mettre en valeur une base de données par la création de web services et d'une application client pouvant communiquer avec ceux-ci.

La création de *web services* à partir de rien peut être une tâche complexe. Pour faciliter ce travail, l'utilisation d'API est conseillée. Ici, nous nous baserons sur les API du *Google App Engine* pour la création des web services. Enfin, nous pourrons déployer notre application dans le *Cloud* de Google.

Dans ce rapport, nous verrons dans un premier temps la structure de la base à valoriser. Dans un deuxième temps, nous verrons l'analyse et la réalisation de l'application serveur et les *web services* exposés. Ensuite, nous expliquerons le choix effectué pour l'application client et expliquerons la structure et le fonctionnement de celle-ci. Enfin, nous verrons les résultats et les tests effectués.

STRUCTURE DE LA BASE

Cette partie a pour rôle d'expliquer l'architecture de la base de données sur laquelle repose l'application. Ainsi dans un premier temps, le service de stockage utilisé pour conserver les informations générées par l'application sera exposé. Dans un second temps, il sera question d'étudier les différentes options possibles afin de stocker et d'accéder aux données via ce service. Enfin, une dernière partie exposera le modèle de données conçu pour les besoins du projet.

2.1 LE DATASTORE

Comme beaucoup d'applications, ce service de gestion de livres nécessite de stocker des données. Ce stockage aurait très bien pu se baser sur un SGBD classique tel que MySQL, ou ORACLE, cependant, c'est une technologie différente qui a été choisie.

En effet, pour ce projet, le stockage des données est effectué grâce à la technologie NoSQL de Google baptisé Datastore. NoSQL ne signifie pas No SQL mais Not Only SQL. Néanmoins, le Datastore n'a rien à voir avec une base de données relationnelle classique, et de ce fait, ne s'utilise pas du tout de la même manière.

Le principe du NoSQL est d'optimiser le stockage et l'accès aux données qu'il emmagasine tout en évitant la lourdeur de certains mécanismes propres aux SGDB classiques comme les jointures par exemple.

Pour ce faire, le Datastore propose de stocker les données sous forme d'entités associées à des clefs. Une entité est définie par un type et contient un ensemble de propriétés auxquelles sont associées des valeurs. Il est possible de voir les entités comme des objets en POO. Les attributs sont aux objets ce que les propriétés sont aux entités du Datastore (figure 1).

Type : Livre

Clef : 50216512002

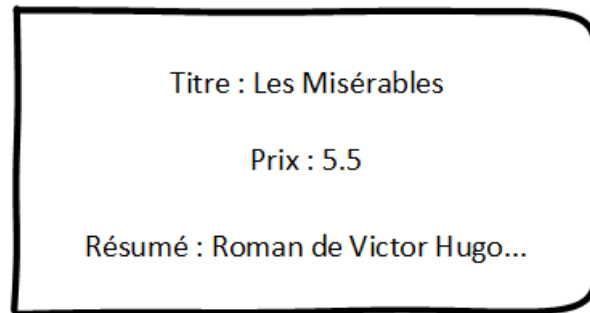


FIGURE 1: Exemple d'entité du Datastore

Une autre analogie qui est souvent faite en NoSQL est celle de la table de hachage. En effet, dans sa globalité, le Datastore peut être vu comme une table de hachage géante. A la manière de ces structures, il associe une clef aux entités qu'il stocke. Ces clefs permettent ensuite d'accéder aux entités sauvegardées avec une complexité de $O(1)$, exactement comme pour une table de hachage classique.

La partie suivante, plus technique, explique de manière concrète comment les données sont manipulées par un programme qui utilise le Datastore.

2.2 ACCÈS AUX DONNÉES

Plusieurs options sont possibles pour interagir avec le Datastore. La première, est d'utiliser l'API Java bas niveau fournit par Google. Cette API permet de bien comprendre comment le Datasore fonctionne, mais elle est assez complexe à utiliser. De plus, le code source qui en résulte est assez répétitif et peut conduire à des problèmes de duplications de code avec tout ce que cela comporte comme inconvénients.

Une autre option qui s'offre aux développeurs est d'utiliser l'implémentation de JDO spécifique au Datastore que fournit Google via le SDK du App Engine. Ce standard, bien connu des développeurs Java, permet d'accélérer grandement la vitesse de développement d'une application et la portabilité du code lorsqu'il s'agit d'interagir avec différents types de bases de données. Cependant, même si Google fournit

une implémentation de ce standard son côté trop abstrait et trop généraliste le rend parfois difficile à manipuler.

La dernière option qui s'offre aux développeurs est d'utiliser une API Java spécialement conçue pour la manipulation de données dans le Datastore appelée Objectify. Objectify offre une syntaxe nettement plus claire et plus transparente que JDO, tout en étant plus pratique à utiliser que l'API de bas niveau du Datastore. Un peu à la manière d'un Framework d'ORM, Objectify permet de créer des classes qui vont ensuite pouvoir être persistées dans le store de manière simplifiée, grâce à un jeu d'annotations.

Pour résumer, Objectify permet d'augmenter grandement la productivité des développeurs qui travaillent avec le Datastore. De plus, le code produit est très lisible, et le temps d'apprentissage pour manipuler l'API est très court. Le seul inconvénient, à l'utilisation d'Objectify, est la portabilité du code, puisque l'API est spécifique au Datastore.

Concernant le projet de gestion de livres, l'application n'a pas vocation à être portée sur d'autres plate-formes que celle de Google. C'est pourquoi il a été choisi d'utiliser Objectify pour manipuler les données stockées en base.

La partie suivante expose le modèle de données bâti sur cette API pour répondre aux besoins de l'application.

2.3 MODÉLISATION ET IMPLÉMENTATION

Comme pour beaucoup d'applications Java EE, il a fallu concevoir une couche métier afin d'implémenter les concepts associés à l'utilité finale du projet, c'est à dire, gérer le stock de livres d'une librairie.

Pour ce faire, deux classes métiers ont été conçues. Une classe Auteur et une classe Livre. La classe Auteur encapsule les informations associées à un auteur comme son nom, son prénom et son adresse. La classe Livre fait de même avec les informations associées à un livre, telle que son titre, son prix ou encore son résumé (figure 2).

Une fois les classes métiers implémentées, vient la question de l'interaction avec la base de données. En effet, comment faire correspondre un modèle objet avec les

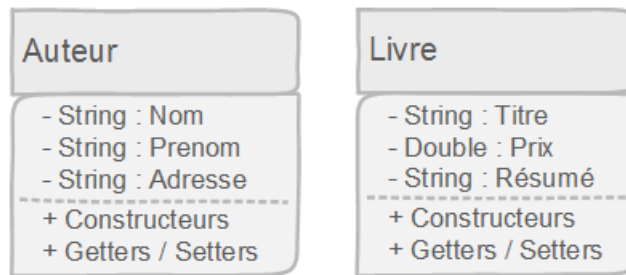


FIGURE 2: Les classes métiers

entités que stocke le Datastore? C'est là qu'Objectify intervient. Grâce à quelques annotations placées dans la déclaration des classes, la bibliothèque va permettre de rendre celles-ci persistantes, un peu comme le ferait une bibliothèque d'ORM (Object-Relational Mapping) avec une base de données classique.

Les futurs objets de type Livre ou Auteur, instanciés par le programme, pourront ensuite facilement être envoyés au Datastore pour être stockés. Cependant, une autre question subsiste. Comment faire pour implémenter le lien entre les classes Livre et Auteur? Avec le Datastore, c'est au développeur d'implémenter lui même un système de clef étrangère (figure 3).

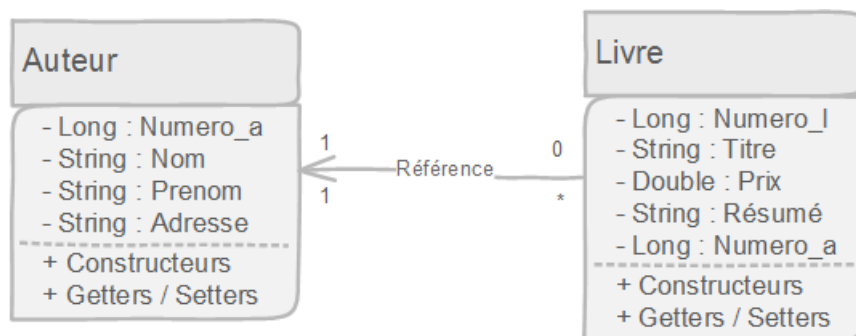


FIGURE 3: Implémentation du lien entre les classes Livre et Auteur

Dans le cas de cette application, il faut mémoriser le lien entre un livre et son auteur, sachant qu'un livre ne peut être écrit que par un seul auteur et qu'un auteur peut avoir écrit 0 à n livres. Avec un SGBD classique, il faudrait créer une clef étrangère dans la table des livres qui référencerait une clef primaire dans la table des auteurs. Avec le Datastore, le développeur ne peut qu'ajouter un identifiant dans la classe livre qu'il

devra faire lui même correspondre avec celui d'un auteur. Voilà pourquoi la classe livre possède un attribut Numero_a.

Enfin une dernière question reste à résoudre. C'est celle de la recherche d'informations dans le Datastore. Effectivement, ce dernier n'est pas prévu pour fonctionner comme un SGBD classique et les requêtes habituelles, écrites avec le langage SQL, ne sont pas présentes ici.

Avec le Datastore, il faut prévoir à l'avance ce que le service va devoir rechercher comme informations et le lui signaler. Pour ce faire, il faut indexer les propriétés des entités sur lesquelles les recherches vont s'effectuer. Dans le cas des objets de la classe Livre, les recherches s'effectueront principalement sur leurs titres, voilà pourquoi l'attribut titre possède l'annotation @Index dans le code de la classe Livre. De même que pour les auteurs, les recherches s'effectueront principalement grâce à leurs noms, voilà pourquoi l'attribut nom est lui aussi indexé.

SERVEUR ET WEB SERVICES

Concevoir la couche métier et la couche d'accès aux données d'une application ne suffit pas à la rendre fonctionnelle. Il faut également trouver un moyen de rendre ces services utilisables. Dans le cadre de ce TP, l'application développée est dite centralisée. C'est à dire qu'elle est destinée à s'exécuter sur un serveur. Ainsi, dans son fonctionnement, elle doit fournir un moyen à d'autres applications, dites clientes, de se connecter et d'interagir avec le serveur. Le but final étant d'avoir accès au service de gestion des livres à distance.

Cette partie du rapport a donc pour rôle de présenter la mise en oeuvre de ce système. Aussi, dans un premier temps, la manière d'interagir avec le programme sera dévoilée, puis dans un second temps l'implémentation concrète de ces interactions sera dévoilée.

3.1 INTERFACE DU SERVICE WEB

Tout d'abord, il faut parler de la stratégie adopter pour communiquer avec le serveur. Dans le cas d'une architecture semblable à celle de ce TP, une technique très employée pour faire dialoguer des applications hétérogènes est celle des services web.

La mise en place d'un service web permet via un protocole standardisé, SOAP ou autre, de demander à un programme qui tourne sur un serveur d'exécuter du code pour une application cliente. Ce genre d'appels à des méthodes distantes se nomme RPC (Remote Procedure Call).

C'est cette technique qui a été mise en oeuvre pour l'application de ce TP. Le service web disponible côté serveur fournit une interface à des applications clientes pour demander l'exécution de procédures implémentées côtés serveur.

Ainsi le premier travail à faire avec ce type d'architecture est de choisir quelles méthodes vont être appelables par les applications clientes. Dans le cadre d'une application de gestion de livres, ces méthodes vont principalement tournées autour de la création, de la lecture, de la mise à jour et de l'effacement d'entités stockées dans la base de données.

Puis il faut passer à l'implémentation. En Java, un service web se définit avant tout grâce à une classe qui va comporter un ensemble de méthodes. Cet ensemble va permettre de définir l'interface du service, c'est à dire l'ensemble des méthodes appelables par les applications clientes (figure 4).



FIGURE 4: Interface du service web

Les méthodes de la classe **BiblioService** représente l'ensemble des procédures appelables par des applications tierces. Elles réalisent une interface de type CRUD (Create, Read, Update, Delete) pour les objets de type Livre et de type Auteur. C'est cette classe qui va permettre aux applications clientes de travailler avec le serveur dans le but de gérer les livres et les auteurs de la base.

Maintenant que l'interface du service web a été exposée, il faut aborder une partie plus technique et il peut être appelé.

3.2 SUPPORT DES RPC

Une fois l'interface du web service implémentée, il reste encore quelques étapes avant de le rendre utilisable par des applications clientes. En effet, pour l'instant, l'application ne fournit qu'une classe exposant les méthodes à appeler, mais elle ne dit pas comment faire pour appeler ces procédures.

C'est l'objectif du protocole SOAP. Ce dernier fournit un standard permettant d'encapsuler un appel de fonction dans une enveloppe XML. Le standard décrit par exemple la manière avec laquelle les paramètres doivent être ordonnés, ou comment retourner le résultat d'une procédure aux clients.

Heureusement pour les développeurs Java EE, il existe une API pour mapper les appels aux méthodes d'une classe avec des requêtes SOAP. Il s'agit de JAX-WS. Grâce à cette API, et à un utilitaire fourni dans le JDK, la traduction d'une requête SOAP en l'appel à une méthode du web service est très aisée.

Pour ce faire, il faut commencer par indiquer quelles méthodes vont être mappées à des requêtes SOAP. Ainsi, il faut placer des annotations devant les méthodes de la classe `BiblioService` pour déclarer concrètement quelles vont être ces méthodes.

Puis, une fois que ces marqueurs ont été placés devant toutes les méthodes à exposer, il faut se servir l'utilitaire `wsgen`. Ce dernier automatise la création de classes appelées artefacts. Elles décrivent en Java les prototypes des méthodes du service web et leurs types de retour. L'API JAX-WS permet ensuite en quelques lignes de transformer une requête SOAP en l'un de ces artefacts. Les artefacts sont ensuite directement manipulables, via du code Java, afin de transmettre les appels SOAP, avec leurs paramètres, aux méthodes du web service. Au passage, `wsgen` permet aussi de générer les fichiers WSDL et XSD du service web. Ces deux fichiers sont essentiels puisqu'ils permettent d'exposer les messages SOAP (correspondant à des appels de méthodes) que le serveur va pouvoir traiter. Lorsqu'un client se connecte au serveur, il consulte ces fichiers, et sait directement comment dialoguer avec le serveur.

Néanmoins, la simple mise en place du fichier WSDL et la création des artefacts ne suffisent pas pour qu'un client puisse dialoguer avec le serveur. Un processus de traite-

ment est nécessaire afin de transformer les requêtes SOAP, reçues depuis un client, en artefacts correspondant à l'appel d'une des méthodes du web service. A l'origine, le serveur Java EE utilisé pour l'application ne sait rien faire d'autre que transmettre les requêtes HTTP qu'il reçoit aux servlets qu'il contient. Il est donc nécessaire d'adapter ce dernier au traitement de requêtes SOAP.

Ainsi, ce mécanisme repose principalement sur deux classes. La première baptisée `SoapHandler` va être chargée de récupérer les requêtes SOAP bruts depuis la servlet et de les transformer en artefact correspondant à la méthode du web service à appeler.

Puis le `SoapHandler` va transmettre cet artefact à un autre composant appelé `SoapAdapter`. C'est ce dernier qui va directement appeler les méthodes de la classe `BiblioService` en leur passant les paramètres contenus dans l'artefact. Puis, une fois l'appel à la procédure terminé, le `SoapAdapter` va lui aussi générer un artefact correspondant à la réponse de la procédure. Il va bien entendu remplir ce dernier avec les résultats retournés par la méthode du service web, et les transmettre à son tour au `SoapHandler`.

La réponse va alors suivre le chemin inverse jusqu'à remonter au niveau de la servlet où elle va être transformée en réponse SOAP, puis renvoyée au client ayant appelé la méthode.

CLIENT DE L'APPLICATION

Une fois l'application serveur développée, il nous a fallu choisir le type d'application client à développer. Dans cette partie, nous expliciterons le choix effectué puis nous détaillerons la structure du client et la façon dont celle-ci communique avec le serveur.

4.1 CHOIX DU CLIENT

Pour l'application client, nous désirions que l'application soit légère, portable et compatible avec un maximum de plateformes. Le choix du client web s'est alors montré comme une évidence. En effet, celui-ci ne nécessite aucune installation pour le client, est compatible avec les ordinateurs classiques comme avec les mobiles. Le client web a l'avantage de pouvoir être maintenu plus facilement puisque ses fonctionnalités sont mises à jour pour tous les clients dès qu'il est mis à jour. Le déploiement est également facilité puisqu'il suffira de le déployer sur le cloud Google en même temps que l'application serveur.

Dans le cadre de ce projet, nous avons préféré utiliser un client riche basé sur du HTML5 et du JavaScript. Les interactions avec le serveur sont donc effectuées via des requêtes AJAX. Nous avons préféré cette technique car elle permet de décentraliser la logique de la page ce qui permet d'alléger la charge du serveur et peut donner un plus grand dynamisme à la page. De plus, l'utilisation de cette technique est favorisée par le développement des normes du web et l'abandon de certains navigateurs obsolètes auxquels on préfère désormais des navigateurs avec des mises à jours automatisées. Le développement de ses applications peut donc de plus en plus se baser sur des API JavaScript innovantes.

4.2 STRUCTURE DU CLIENT

Maintenant que le choix du client a été explicité, nous allons analyser plus précisément sa structure. Dans un premier temps, nous aborderons son design. Ensuite, nous verrons le modèle développé en JavaScript pour le stockage et la gestion des entités. Enfin, nous verrons le fonctionnement du client.

4.2.1 DESIGN

Le design et l'ergonomie occupe une grande part de tout projet. Pour celui-ci, nous nous sommes basés sur un le Framework CSS *Metro UI CSS*¹. Celui offre un *flat & responsive design* et de nombreux composants avec une syntaxe relativement légère. Il inclut également un script JavaScript permettant l'animation de ses composants se basant sur jQuery et son plugin jQueryUI.

4.2.2 MODÈLE

Le client est capable de communiquer avec le web service et donc d'effectuer toutes les opérations du CRUD pour des entités de types Auteur et Livre. Nous avons donc choisi de créer deux classes Author (figure 5) et Book (figure 6) permettant d'effectuer ces opérations. De plus, ces classes sont capables de donner une visualisation des informations des entités dans l'interface. Les classes créées sont dépendantes de jQuery.

Author
<div><div>-id: Number</div><div>-name: String</div><div>-firstName: String</div><div>-address: String</div></div>
<div><div>+create(): Author</div><div>+read(query: String): List<Author></div><div>+read(id: Number): Author</div><div>+update(name: String, firstName: String, address: String)</div><div>+delete()</div></div>

FIGURE 5: Classe Author

En effet, leur création utilise les fonctionnalités de celui-ci pour pouvoir obtenir une syntaxe claire.

¹ metroui.org.ua

Book
-id: Number -title: String -description: String -price: Number -authorId: Number
+create(): Book +read(query: String): List<Book> +read(id: Number): Book +update(title: String, description: String, price: Number, authorId: Number) +delete()

FIGURE 6: Classe Book

4.2.3 FONCTIONNEMENT

Dans cette section, nous verrons les fonctionnalités des différentes vues de l'application. La vue principale contient la barre de navigation (figure 7). Celle-ci contient un élément avec le nom de l'application qui permet de revenir sur la page d'accueil, une barre de recherche qui permet de rechercher une chaîne de caractère parmi les auteurs et les livres. Enfin, un bouton d'ajout permet d'ajouter un nouveau livre ou un nouvel auteur (figure 8).



FIGURE 7: Vue principale

Lorsque l'on clique sur le bouton d'ajout d'auteur, un nouvel auteur est créé (figure 9). Les propriétés de l'auteur peuvent être éditées en cliquant sur les éléments. Ceux-ci sont ensuite transformés en input. La perte de focus de l'élément remet l'élément à son état d'origine (h1, h2 ou h3) et appelle la méthode mise à jour auprès du web service. La figure 10 montre une vue avec un auteur modifié. L'auteur peut également être supprimé en cliquant sur l'icône de corbeille en haut à droite.

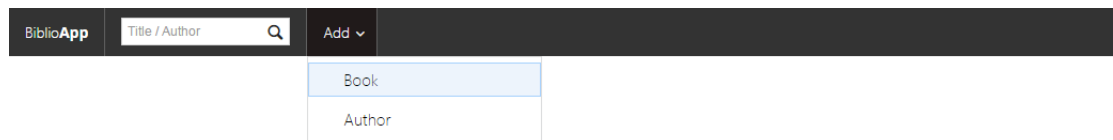


FIGURE 8: Bouton d'ajout



FIGURE 9: Ajout d'un auteur

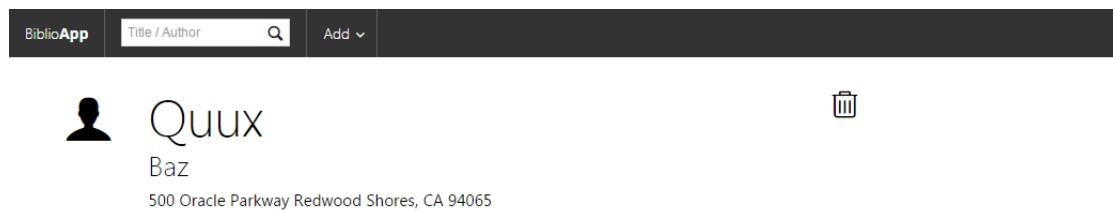
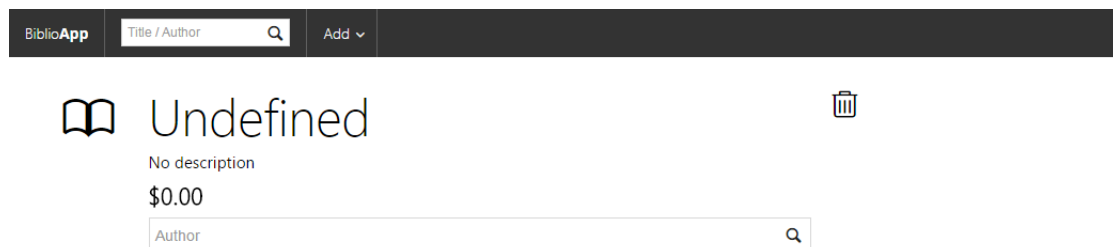


FIGURE 10: Auteur modifié

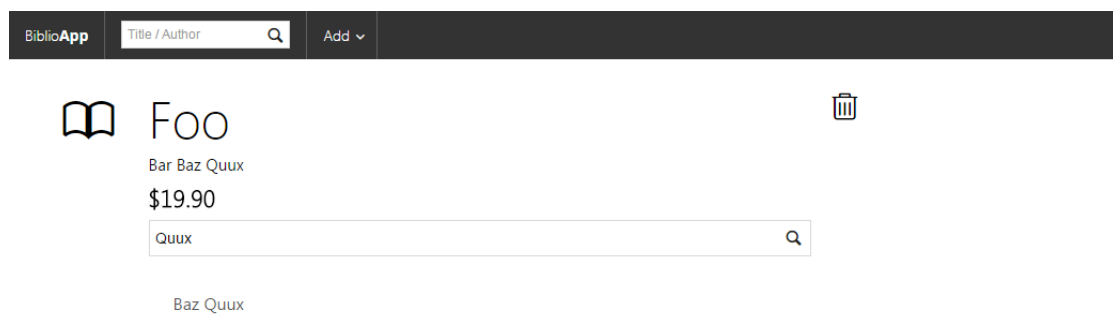
L'ajout d'un livre est également possible en cliquant sur le bouton d'ajout de livre (figure 11). Les propriétés du livre peuvent également être modifiées en cliquant sur

les éléments. Ceux-ci sont transformés en input ou textarea. Comme pour l’auteur, les éléments reviennent à leur état d’origine (h1, p et h3) avec la perte de focus et sont ensuite mis à jour auprès du serveur. Pour changer l’auteur du livre, on peut effectuer une recherche dans la barre de recherche du formulaire. La liste des auteurs correspondant sera ensuite affichée (figure 12). Un clic sur l’un d’entre eux change l’auteur du livre. Pour changer à nouveau l’auteur, il faut cliquer sur l’icône de crayon pour réafficher la barre de recherche. La figure 13 montre une vue avec un livre modifié. On peut également



The screenshot shows the top navigation bar of the BiblioApp with a search input labeled 'Title / Author' and an 'Add' button. Below the bar, a book entry is displayed with a book icon, the title 'Undefined', a trash icon, and the text 'No description'. The price is '\$0.00'. There is an 'Author' input field with a search icon. Below the input field, the text 'Baz Quux' is visible, indicating a search result.

FIGURE 11: Ajout d’un livre



The screenshot shows the same BiblioApp interface as Figure 11, but with the title 'Foo' and the price '\$19.90'. The 'Author' input field now contains the text 'Quux'. Below the input field, the text 'Baz Quux' is visible, indicating a search result.

FIGURE 12: Recherche d’un auteur pour un livre

Enfin, la barre de recherche de la barre de navigation permet d’effectuer la recherche d’auteurs et de livres existants. Si aucun paramètre n’est passé, la recherche renvoie tous les auteurs et livres (figure 14). Sinon, le paramètre passé est recherché parmi les auteurs et les livres et les éléments correspondant sont affichés (figure 15).

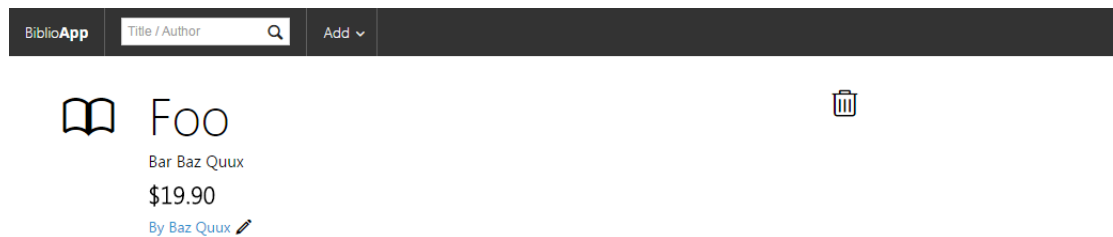


FIGURE 13: Livre modifié

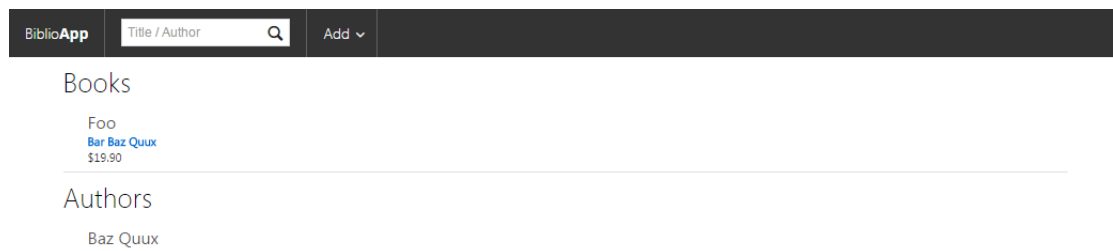


FIGURE 14: Recherche sans paramètre



FIGURE 15: Recherche avec paramètre

4.3 COMMUNICATION AVEC LE SERVEUR

Dans cette section, nous verrons plus en détails la façon dont le client commu-
nique avec le serveur. Nous effectuons des XHR (XML Http Requests) auprès du web

services. Les classes `Book` et `Author` possèdent des méthodes permettant d'effectuer ces requêtes. Les web services utilisant un format soap, nous avons utilisé le plugin `jQuery soap` afin de construire les requêtes à partir d'un JSON. Les requêtes reçues sont ensuite retransformées en JSON à l'aide de la bibliothèque `xml2json`.

Les objets JavaScript sont ensuite construits grâce aux JSON renvoyés par les web services répondant aux requêtes `read()` et `create()`. Les requêtes `delete()` et `update()` n'utilisent pas la réponse à la requête.

RÉSULTATS ET TESTS

Différents aspects de l'application ont pu être testé. Toutes les méthodes du CRUD sont fonctionnelles sur les objets livres et auteurs. On peut ainsi ajouter, modifier, récupérer et supprimer ceux-ci.

Ces fonctionnalités sont fonctionnelles mais une amélioration pourrait être apportée à la récupération des données. En effet, les modifications apportées sont traitées de façon asynchrones par le datastore. Si l'objet est lu juste après sa modification, celle-ci ne sera pas forcément directement visible. Différentes solutions seraient possibles pour implémenter cette amélioration. Soit côté serveur, en forçant l'actualisation lors d'une requête de lecture ou en récupérant les objets modifiés depuis un cache commun. Cette solution implique que l'application sur un seul serveur car les opérations de synchronisation entre serveur sont coûteuses en temps et en performance. L'autre solution serait d'implémenter un mécanisme de cache dans l'application cliente en gardant en mémoire les objets déjà récupérés. Cette solution permet uniquement à l'utilisateur de voir ses modifications directement effective, elle ne tiendra pas compte des modifications des autres utilisateurs.

L'application développée aurait également pu utiliser des frameworks tels que AngularJS offrant des niveaux d'abstraction plus importants pour faciliter le développement et la maintenance de celle-ci.

CONCLUSION

Pour conclure, ce projet a permis de développer une application fonctionnelle permettant de mettre en valeur une base de données en utilisant le *Google App Engine*.

Ce projet a été source d'enseignements pour les technologies incluses dans le *Google App Engine* et il a permis de mieux comprendre comment celles-ci fonctionnaient. Il a également permis de mieux appréhender certaines contraintes liées au développement, au déploiement et à l'utilisation d'application web.

Enfin, il nous a permis de tester un des types de clients possibles pour une application déployé dans le cloud et la phase de réflexion sur le choix de celui-ci nous a permis de prendre en compte les avantages et inconvénients de chaque client possible.