



Découverte du framework Apache OFBiz.

Développement d'une API HTTP,
basé sur le style architectural REST et
intégration dans un contexte de projet
client.

Auteur :
Artemiy ROZOVYK

Tuteur de stage :
Mathieu LIRZIN
L'enseignant référent :
Florent FOUCAUD

2 juin 2019

Table des matières

Remerciements	1
1 Introduction	1
2 Contexte du stage	2
2.1 Entreprise	2
2.1.1 Présentation générale	2
2.1.2 Activité	2
2.1.3 Projets	2
2.1.3.1 Décathlon	2
2.1.3.2 Déjbox	3
2.2 Framework OFBiz	3
2.2.1 Vue d'ensemble	3
2.2.2 Architecture	3
2.2.3 DSL XML	4
2.2.4 Container	4
2.2.5 Composants	4
2.2.6 Web applications	4
2.2.7 Entity engine	5
2.2.8 Service engine	6
2.2.9 Screen engine	6
2.2.10 Fonctionnel métier	7
2.3 Sujet de stage	7
2.3.1 Découverte OFBiz	7
2.3.2 API REST au sein d'OFBiz	8
3 Travail réalisé	9
3.1 Aperçu général	9
3.2 Environnement	10
3.2.1 Installation de l'environnement	10
3.2.2 Formation générale	10
3.2.3 Jira	10
3.2.4 Approfondissement de Git	10

3.2.5	Découverte de communauté libre Apache	10
3.3	Prise en main d'OFBiz	11
3.3.1	Premier plugin	11
3.3.2	Projets existants et leur structure	11
3.3.2.1	Décathlon	11
3.3.2.2	Dejbox	11
3.3.3	Problématique vis-à-vis du développement	11
3.4	Etat de l'art	11
3.4.1	Histoire et problématique des applications web	11
3.4.2	Representational state transfer	13
3.4.2.1	Histoire	13
3.4.2.2	Principe	13
3.4.2.3	En pratique	14
3.4.2.4	Utilisation des méthodes HTTP	15
3.4.2.5	Exemples d'API du style REST	15
3.4.3	Implementations existantes	16
3.4.3.1	Camel	16
3.4.3.2	JAX-RS	16
3.5	Analyse de l'existant	17
3.5.1	Gestion des application web dans OFBiz	17
3.5.2	ControlServlet	17
3.5.3	RequestHandler	17
3.5.4	Controleur.xml	17
3.5.5	API en cours	18
3.5.6	Mécanisme de résolution des URI	18
3.5.7	<i>OverrideView()</i> et le conflit avec les URI segmentées	18
3.6	Analyse des besoins et attentes de la maîtrise d'ouvrage	19
3.6.1	Besoins d'évolution	19
3.7	Réalisations techniques	19
3.7.1	Choix vers URITemplate	19
3.7.2	Librairie CXF	20
3.7.3	Intégration en parallèle avec le système existant	20
3.7.4	Nouveau controller.xml	20
3.7.4.1	Les collisions d'URI	21
3.7.4.2	Testes Mockito	21
3.7.5	Modification de la partie "Administration : gestion des entités" (entitymaint)	21
3.7.5.1	Choix de la partie illustrative	22
3.7.5.2	Difficultés de modification	22
3.7.5.3	Clés composées	22
3.7.6	Stateless	22
3.7.6.1	Les réalisation par la communauté	22
3.7.7	RESTClient pour la communauté	23

3.7.7.1	Généralisation de code	23
3.7.7.2	Correction d'incohérences	23
4	Conclusion	24
4.1	Lien avec les connaissances obtenu lors de la formation universitaire . .	24
4.1.1	MVC	24
4.1.2	Servlet	24
4.1.3	raeeMarker -JSP	24
4.1.4	Notion d'entié - Symony	24
4.1.5	Routage	24
4.1.6	Gradle Maven	24
4.1.7	Testes Unitaires	24

Remerciements

Merci tout le monde !

Chapitre 1

Introduction

Le présent document expose le travail effectué lors du stage de fin de licence au sein de la société Néréide. Ce stage se décompose en deux parties : Premièrement, il a pour but de se familiariser avec la suite d'applications libres pour l'entreprise Apache OFBiz et son utilisation dans le contexte de la société d'accueil. Finalement, il consiste à intégrer un système permettant la définition des API HTTP du style REST, ainsi que la modification d'une API existante afin de fournir une preuve de concept consistante. Le travail a été effectué en étroite collaboration avec le principal concerné : la communauté Apache, ce qui a contribué à une meilleure cohérence entre le travail réalisé et les besoins des utilisateurs.

Dans un premier temps nous allons présenter l'entreprise d'accueil ainsi que faire une description de l'outil principal utilisé. Dans un deuxième temps nous exposerons la démarche qui a permis une compréhension suffisante du Framework OFBiz nécessaire à la partie finale du stage, REST, qui sera décrite dans un troisième temps.

Chapitre 2

Contexte du stage

Intro

2.1 Entreprise

2.1.1 Présentation générale

Néréide est une société de services en logiciels libres créée en 2004 qui se spécialise dans l'intégration du progiciel de gestion intégré Apache OFBiz. Il s'agit d'une société coopérative et participative (SCOP) qui se situe à Tours.

2.1.2 Activité

Transparence libre entreprise

Integration (utilisation des briques en tant que telles) et Développement spécifique (adaptation d'OFBIZ pour les besoins) plugins propres à la logique métier. Pas de forfait (contrat doit être rempli, l'entreprise s'engage à ce qu'il soit livré dans les temps), mais la régie (sont payé au temps de travail et pas à condition de remplir un contrat). Pareil avec Décath 6 pers font de la régie... Administration système

2.1.3 Projets

2.1.3.1 Décathlon

L'un des principaux clients de Néréide est le groupe Decathlon qui se spécialise en grande distribution de produits de sport et de loisirs. Le projet représente une plateforme de vente de puces RFID (bookmark), qui assure l'intégralité du processus d'achat.

pour les acheteurs et producteurs, dev front pour le store spécifique au lieu d'écrans OFBiz, communication avec des API...

2.1.3.2 Déjbox

Dejbox est une société de la foodTech qui propose aux salariés d'entreprise de leur livrer des repas directement sur leur lieu de travail. L'ensemble des vente est réalisé au travers d'un site e-commerce par lequel le salarié commande un repas.

Le projet a pour objectif de mettre en œuvre un outil de type ERP afin de gérer la chaîne de réapprovisionnement en produit frais vendu en ligne. Il s'agit donc de créer un référentiel d'article et de fournisseur et de pouvoir saisir des commandes qui seront envoyées aux fournisseurs et réceptionnées suite à leur livraison. Enfin, il s'agit de mettre en place la sortie de stock en intégrant les consommations de produit provenant du site de vente en ligne.

.... Depuis 2013 les dev remontantes à la communauté , parce que besoin de support, divergence. moutons-acteur

2.2 Framework OFBiz

2.2.1 Vue d'ensemble

Open For Business (OFBiz) est une suite d'applications pour la gestion de l'entreprise qui se base sur une architecture très couramment utilisé (*MVC*) et qui implémente des composantes classiques de gestion des donnés, de logique métier, et de traitement spécialisé.

On peut notamment distinguer les modules génériques destinées à la gestion des tâches communes à la plupart des entreprises, telles que la gestion des stocks, la comptabilité, la facturation et bien d'autres. Quant à leur structure, toutes les composantes sont étroitement liées entres elles, ce qui facilite la compréhension, l'utilisation et la personnalisation de ces dernières.

En plus d'une architecture qui encourage la customisation, OFBiz est entièrement distribué en tant que *open source software*¹ ce qui le rend particulièrement intéressant car le logiciel développé à base de OFBiz n'est pas soumis à la condition d'être libre comme c'est le cas de la licence GPL² par exemple.

2.2.2 Architecture

D'un point de vue purement technique OFBiz se base sur la plateforme Java ainsi que sur l'utilisation des DSL³ basés sur des grammaires écrites en XML (mettre la bib). En ce qui concerne la partie principale du framework, les échanges HTTP sont implémentes par une extension de la classe `HttpServletRequest` [2] et la communication avec les bases de données se fait via l'API Java JDBC [3].

1. Logiciel libre sous licence ASL2 (Apache License Version 2.0) ce qui donne le droit de personnaliser, d'étendre, de restructurer et de vendre le système concerné.

2. GNU General Public Licence

3. Domaine specific language (*Langages spécifiques au domaine*)

Dans sa structure on distingue *le framework*, *les applications* et *les plugins*. Le *framework* comporte l'ensemble des outils et des mécanismes techniques de bas niveau utilisés par les applications. Il fournit notamment des fonctionnalités présentes dans la plupart des frameworks de développement (couche données, logique métier, gestion des transactions, etc...). Les principaux composants métier tels que la comptabilité, la gestion des stock, ou la facturation se trouvent dans la partie *applications*. Finalement la notion du plugin correspond à une application spécifique qui repose sur des composantes générales : par exemple le plugin *eCommerce* correspond à un boutique en ligne interagissant avec des nombreuses *applications* comme *la gestion du stock* ou *la facturation*.

2.2.3 DSL XML

L'une des particularités d'OFBiz ce sont des fichiers XML qui servent à déclarer entre autres des routes HTTP, des pages de rendu appelés *Écrans*, ainsi que des services. Le principe est de transformer des informations sous format XML facilement compréhensibles par le développeur, en objets Java correspondants.

2.2.4 Container

L'interface container représenté sur la figure 2.1 permet de définir des objets qui correspondent à des processus qui peuvent être initialisés, démarrés et arrêtés. L'intérêt est de pouvoir lancer un daemon spécifique en parallèle de l'exécution d'OFBiz comme c'est le cas de `EntityDataLoadContainer` qui est responsable du chargement des données et leur mise à jour en cas de modification du modèle. Quand à `TestRunContainer` il s'assure du lancement des testes unitaires grâce à un mécanisme propre au framework.

2.2.5 Composants

Les éléments constitutifs de OFBIZ sont des composants. Un composant est un regroupement des containers, des entités, des services, des vues (*Écrans*) et des applications Web.

L'exemple classique d'un composant est celui de `webtools` qui assure la gestion technique de l'ensemble du système par l'administrateur via une application web, ce qui implique le fait que ce composant regroupe la plupart des éléments majeurs du framework. Nous en tant que développeurs avons la possibilité de définir nos propres composants, notamment des *plugins*.

2.2.6 Web applications

Des composant OFBiz ne peuvent pas être accédés directement par les utilisateurs, ils servent simplement à organiser le framework en parties individuelles de chaque aspect de l'ERP afin de faciliter leur gestion. Les applications web (*webapps*) sont destinées à

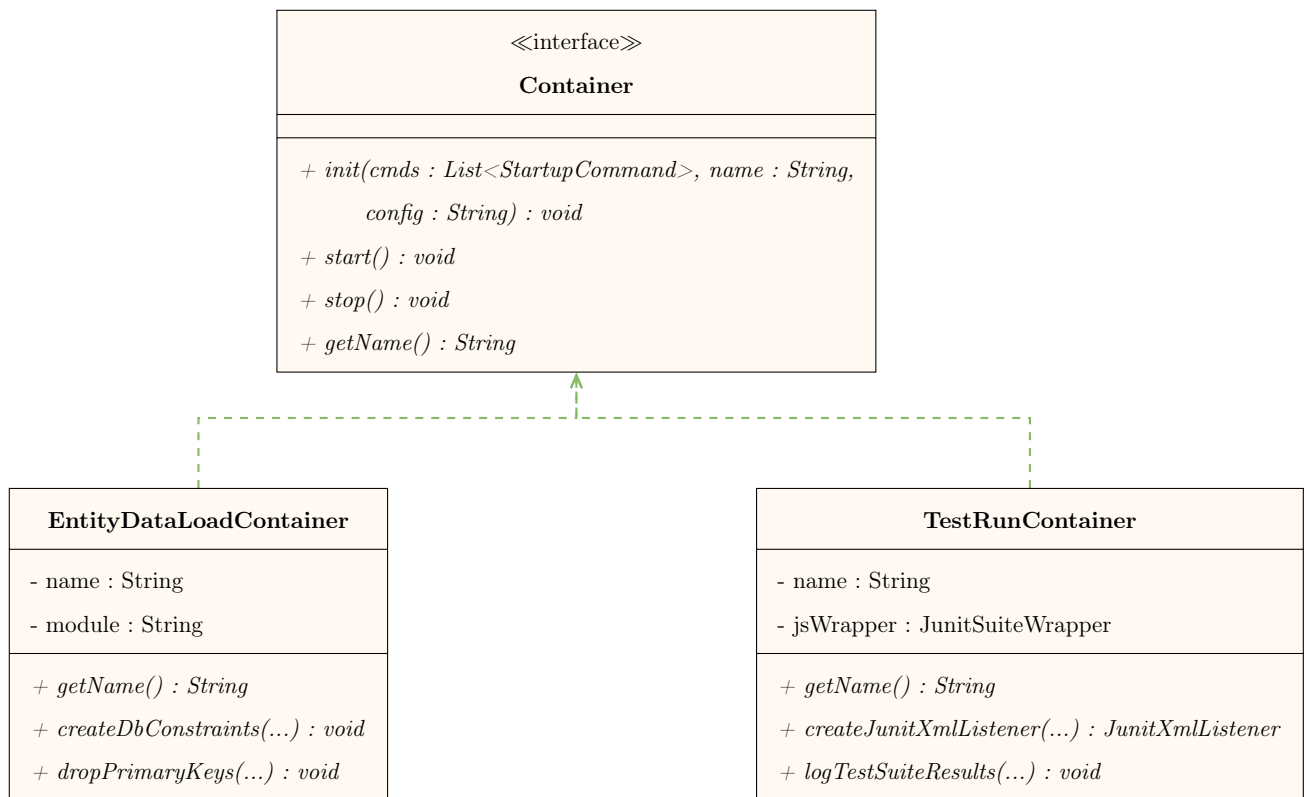


FIGURE 2.1 – Définition du type container

fournir un front-end afin que les utilisateurs puissent interagir avec OFBiz. En ce qui concerne les routes HTTP, définis classiquement dans le fichier `web.xml`, dans le cas de OFBiz leur gestion est délégué à un fichier `controller.xml` qui à son tour associe des traitements spécifiques à chaque point d'entrée HTTP ainsi que la valeur de retour qui peut être une vue (*Écran*), du type **JSON** ou bien une redirection. Cela se fait au moyen d'une `request-map` comme on peut voir sur l'extrait de code suivant 2.2

2.2.7 Entity engine

Comme dans beaucoup d'autres frameworks, l'interaction avec les bases de données à une place principale dans le OFBiz. Le moteur d'entités (*Entity engine*) se charge de

```

<request-map uri="stock">
  <event type="service" invoke="getStock"/>
  <response name="success" type="view" value="stockScreen"/>
</request-map>
  
```

FIGURE 2.2 – Association d'un point d'entrée et d'une réponse

```

<service name="getStock" engine="entity-auto"
           default-entity-name="Stock">
  <auto-attributes include="pk" mode="IN"
                  optional="false"/>
  <attribute name="authKey" type="String" mode="IN"
            optional="true"/>
  <attribute name="stockList" type="String" mode="OUT"
            optional="true"/>
</service>

```

FIGURE 2.3 – Définition d'un service

la communication avec les bases de données à travers les déclarations uniformes, c'est à dire qui changent pas peu importe le choix de l'outil externe de gestion.

2.2.8 Service engine

Les services web assurent les échanges d'information entre les applications, communément via le protocole HTTP. Les services OFBiz fonctionnent dans une architecture orientée service (SOA). Non seulement ces services ont une capacité d'évoquer les autres intérieurement, mais peuvent aussi être appelés par une application extérieure en utilisant des protocoles d'échange d'information tels que SOAP.

Les services OFBiz sont appelés en passant un contexte⁴ et retournent une réponse parmi celles conventionnellement nommés : *"success"*, comme on peut voir dans 2.2 , *"error"* ou *"failure"* ainsi que l'ensemble des données retournées par le service.

On peut voir l'exemple de la définition d'un service sur 2.3 , qui montre notamment la saisie des attributs attendu par le service qui sont définis de deux manières : En utilisant le mécanisme de **auto-attributes** qui génère des attributs⁵ à partir de l'ensemble des clés primaires de l'entité **Stock**. L'autre manière de faire est de rajouter des attributs manuellement comme on peut le voir dans la suite de l'exemple.

2.2.9 Screen engine

La partie Vue du MVC est représentée par des *Écrans* ou les **Screen** qui font partie du **Widget toolkit**⁶ de OFBiz. Le principe du fonctionnement d'un écran se base toujours sur un DSL qui interagit avec un mécanisme de rendu générique capable de gérer différents formats de sortie comme HTML, XML, CSV, ou XLS. Le cas d'utilisation le plus commun est celui d'une réponse HTML qui est éventuellement générée en déléguant le rendu à un autre mécanisme : **Apache Freemarker**.

4. Définis souvent dans les paramètres de la requête HTTP

5. Qui sont en l'occurrence en entrée (de paramètre IN)

6. Boîte d'outil de composant d'interface graphique

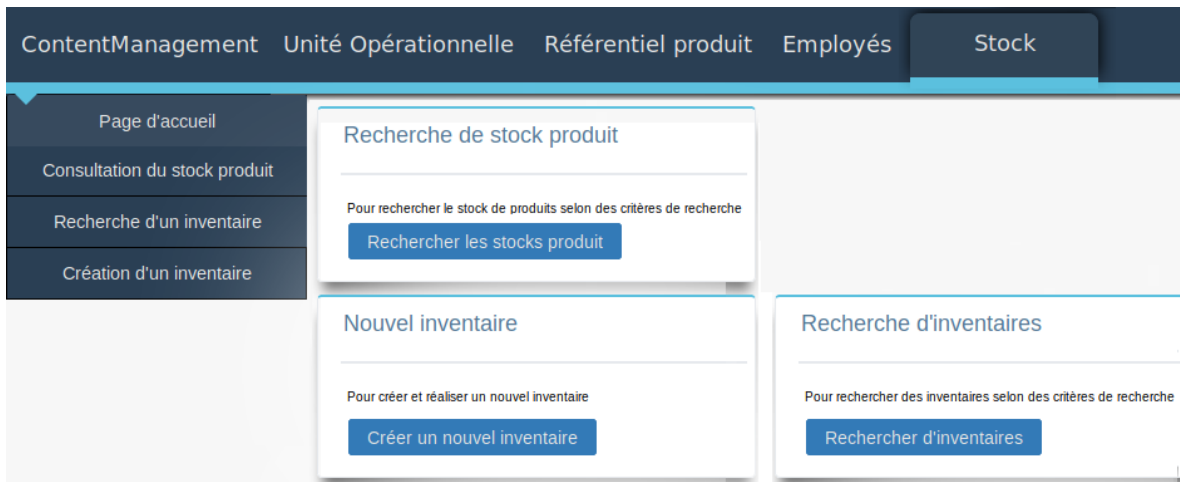


FIGURE 2.4 – L'exemple d'un Ecran de rendu OFBiz.

2.2.10 Fonctionnel métier

Le méta-composant *Applications* comporte des composants métier prédéfinis, qui ont pour vocation de fournir des solutions fonctionnelles "prêtes à l'emploi".

Ainsi on distingue parmi d'autres :

accounting fournit l'ensemble de services et d'écrans de rendu pour la comptabilité.

datamodel s'occupe de la définition des entités et des relations entre elles.

humanres correspond à la gestion du personnel.

product permet la gestion du catalogue des produits et des services fournis par l'organisme.

2.3 Sujet de stage

Comme mentionné avant, le sujet de stage est divisé en deux parties. La première partie consiste à prendre en main l'outil de développement concerné, le framework OFBiz. La deuxième partie du stage est la modification de certains éléments du framework, notamment des mécanismes de gestion de services afin d'assurer la conformité au style architectural REST.

2.3.1 Découverte OFBiz

Cette étape sert à se familiariser avec l'environnement du framework à travers des tutoriels et l'analyse des projets existants, afin de comprendre le fonctionnement général. Cela permet aussi de repérer les points essentielles auxquelles il faut tenir compte lors de la deuxième phase de travail.

2.3.2 API REST au sein d'OFBiz

Finalement, l'intérêt principal du stage est la mise en place d'un système de gestion des services REST. L'idée a été évoquée pour la première fois dans une discussion communautaire Apache OFBiz, car le mécanisme en cours nécessitait de l'évolution. Cette discussion a suscité des nombreuses remarques en terme de faisabilité et a permis de retrouver des pistes à poursuivre lors de l'implémentation.

Dans un premier temps on considère la possibilité d'intégration d'une solution externe notamment à travers des librairies JAX-RS de CXF, ainsi que Apache Camel. Malgré les premières prototypes fonctionnelles l'idée d'une solution externe a été abandonné pour des raisons expliqués plus loin dans le document. A sa place une implémentation de bas niveau a été adoptée. Une fois le système mis en route, l'étape suivante consistait à faire adopter la modification dans la branche principe du projet OFBiz.

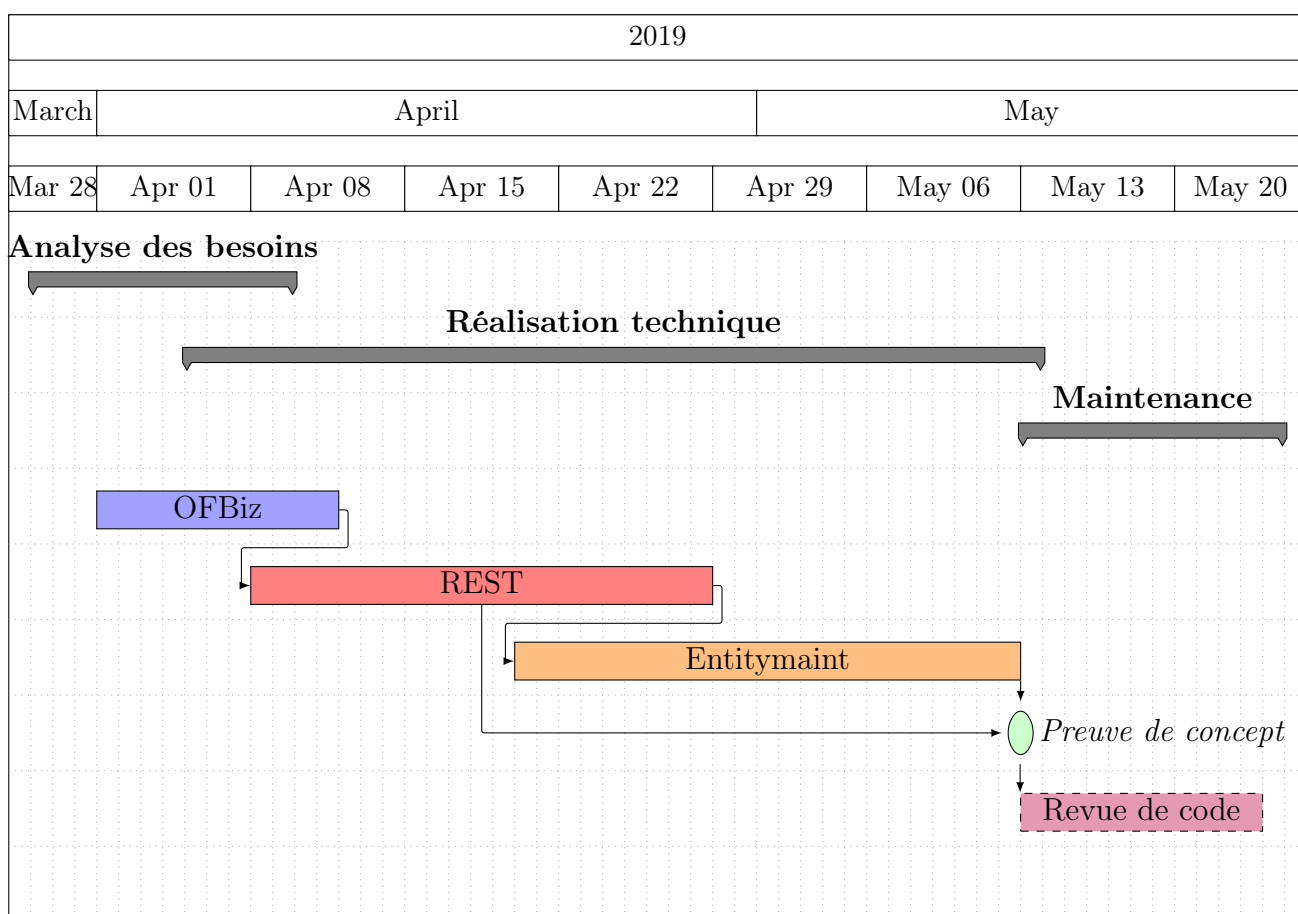
Afin de prouver la conformité du nouveau système et démontrer les nouveaux fonctionnalités, la décision a été prise de modifier une partie front-end existante, notamment l'interface de gestion des entités *entitymaint*. Finalement, l'ensemble de modification sous la forme des patchs a été soumis à la communauté afin d'être revus et potentiellement adoptés dans le framework.

Chapitre 3

Travail réalisé

3.1 Aperçu général

Voici la chronologie du travail réalisé en entreprise.



3.2 Environnement

3.2.1 Installation de l'environnement

Avant tout, mon intégration a commencé par l'installation du poste de travail suivi par une discussion sur le choix de distribution Linux , la configuration des outils utilisés par l'entreprise ainsi que par la mise en place des accès aux ressources internes. Le choix d'IDE à été fait au faveur de IntelliJ car il possède des nombreux moyens de navigation qui sont incontournables dans la structure de OFBiz riche en dualité XML/Java.

3.2.2 Formation générale

Lors de la formation générale, traditionnellement prévue pour tout les nouveau venus de Néréide, j'ai pu découvrir le fonctionnement basique de OFBiz à travers les démonstration des projets existants. Les points soulevés comportait la gestion des dépendances à travers **Gradle** et **Ant**¹, une introduction au langage **Groovy** et les raisons pour lesquelles il a été préféré au DSL propre à OFBiz²

3.2.3 Jira

Un autre point intéressant était le système de gestion de tickets **Jira** utilisé par l'entreprise qui permet de suivre et gérer les bugs tout en interagissant avec les clients. L'outil est utilisé également par la plupart des projet communautaires Apache, dont OFBiz.

3.2.4 Approfondissement de Git

Même si la gestion de versions de OFBiz était historiquement géré par l'outil Apache SVN, dans la gestion de ses propres projets, l'entreprise a fait le choix pour un système plus moderne - Git.

Alors que j'avais déjà une certaine maîtrise basique de Git, je n'ai jamais eu l'occasion de travailler dans un projet qui comporte des dizaines de branches qui évoluent quotidiennement. Donc, pour monter en compétences sur ce point-là j'ai utilisé le site d'apprentissage conseillé par mon maître de stage : [Learn Git Branching](#)

3.2.5 Découverte de communauté libre Apache

OFbiz est un projet libre, maintenu par les intéressés qu'on peut catégoriser comme : **Contributeurs** ceux qui suggèrent des modifications utiles au projets mais ne modifient pas la branche principale.

1. Ant étant déprécié depuis la version 16.11 de OFBiz mais certains projets client l'utilisent toujours car ils se basent sur une version antérieure.

2. Pour remplacer le DSL en XML sous le nom Mini lang, en train d'être entièrement déprécié.

Commiteurs sont des contributeurs responsables de la validation des modifications du framework ainsi que de leurs adoption dans le code source.

Membres de PMC sont responsables des décision sur la structure générale du projet et sur la cohérence des modification vis-à-vis de cette dernière³

3.3 Prise en main d'OFBiz

3.3.1 Premier plugin

3.3.2 Projets existants et leur structure

3.3.2.1 Décathlon

RFID et tout ça

3.3.2.2 Dejbox

Pierre et Antoine ont tout géré

3.3.3 Problématique vis-à-vis du développement

What is "fonctionnel", La prob

3. PMC acronyme de Project Management Committee (Comité de gestion du projet)

3.4 Etat de l'art

3.4.1 Histoire et problématique des applications web

Avec l'évolution des technologies du réseau, on a obtenu à la fin des années 60 - début des années 70, la possibilité d'échanger des informations numériques entre les machines. Cela a permis l'émergence des systèmes d'échange d'information de plus en plus efficaces. Au début, il s'agissait des architectures très simples, avec une seule couche où une machine unique (*le serveur MainFrame*) effectuait tous les traitements relatifs et qui était accédé par un terminal passif⁴. Cela présentait l'avantage d'un système centralisé et homogène facile à implémenter, mais de nombreux inconvénients comme la complexité de maintenance du code monolithique et panne générale en cas d'indisponibilité du MainFrame.

L'arrivée des ordinateurs personnels a permis la séparation de la couche présentation et parfois aussi de la couche application qui étaient désormais placés sur la machine des clients. Grâce à cela on a pu proposer des affichages plus sophistiqués (des clients lourds et légers avancés). C'est notamment avec l'arrivée des architectures à 2-tiers qu'a fait l'apparition la notion de service - une fonctionnalité fournie par le serveur.

La manière d'évoquer ces services était définie par une interface standardisée, d'où la notion de API (Application Programming Interface), ou bien interface de programmation applicative qui sert à proposer des points d'entrée pour un nombre de clients de nature différente (utilisateur humain ou une autre machine).

Cependant, l'architecture 2-tiers avait présenté aussi des inconvénients majeurs : Si le client interagissait avec plusieurs serveurs il devait comprendre l'API de chacun d'entre eux. De plus, c'était dans la responsabilité du client de gérer l'ordre des appels, la cohérence et la combinaison des données reçues. S'ajoute à cela les serveurs qui pouvaient pas communiquer l'un avec l'autre, résultat - la complexité grandissante des clients.

L'arrivée d'une architecture à 3 tiers (voire N-tiers avec les couches sécurité, couche routage, etc...) et ainsi que de la notion de Middleware⁵ ont apporté un certain nombre d'avantages en terme de réduction de complexité côté client et d'interopérabilité entre ces différentes couches. De plus, de nombreux standards appelés WS-* ont été adoptés afin de combler les failles apparues suite à l'utilisation répandue de Middleware. Suite à cela des développeurs et des sociétés ont commencé à mettre en œuvre tout le stack WS-* même pour des tâches où l'utilisation du HTTP suffirait. Concrètement, le HTTP était réduit au protocole de transport avec une énorme charge XML transmise qui définissait l'échange.

Cette approche fonctionne relativement bien dans le contexte d'un seul ou plusieurs organismes qui partagent le même système d'information. Mais dès qu'il s'agit de proposer des services à l'extérieur (à l'échelle mondiale via Internet), cela s'avérait extrêmement compliqué.

4. Le terme français pour l'équivalent anglais moins gentil, *Dumb terminal* - une machine sans capacités de calcul qui sert uniquement à afficher l'information reçue

5. Des intergiciels destinés à lier des systèmes informatiques de nature différente

3.4.2 Representational state transfer

Dans ce chapitre on va présenter le style architectural REST et comment il répond à des problématiques évoquées dans la section précédente.

3.4.2.1 Histoire

En 2000, alors que le nombre de site web dans le monde a atteint 17 millions, Roy Fielding, l'un des contributeurs du protocole HTTP et URI, définit REST dans sa thèse de doctorat intitulée *Architectural Styles and the Design of Network-Based Software Architectures* [6]

3.4.2.2 Principe

Du point de vue abstrait ce style architectural se base sur les technologies fondamentales du World Wide Web : HTTP, Uniform Resource Identifier (URI), les langages de balisage HTML et XML, ainsi que sur des formats adaptés web comme JSON. REST est un style architectural pour les applications qui interagissent via réseau. Il est défini par six contraintes suivantes :

Uniformité d'interface Cette contrainte fondamentale à REST, oblige à manipuler et identifier des ressources uniquement via leurs représentations (par exemple sous formats HTML, XML ou JSON). Ainsi, chaque représentation fournit suffisamment d'information au client, afin qu'il puisse la modifier ou supprimer.

Client-Serveur Assure l'absence d'interdépendance entre les clients et les serveurs. Ainsi le serveur propose des services d'une manière générique sans dépendre des spécifications du client (langage de programmation utilisé, plateforme, etc...). Cela permet au client et au serveur d'évoluer indépendamment.

Sans état Les échanges entre le client et le serveur doivent s'effectuer sans conserver l'état de la session sur le serveur entre deux requêtes successives. Le but est d'améliorer les performances du serveur ainsi que l'extensibilité du système.

En couches Cette contrainte permet de séparer l'architecture en plusieurs niveaux, cela facilite la mise en échelle du système.

Code à la demande Ceci est une contrainte facultative qui permet aux serveurs d'étendre ou de modifier le fonctionnement du client grâce à l'envoi du code exécutable.

Mise en cache Mise en cache de certaines données accédées fréquemment afin d'augmenter les performances.

En respectant ces contraintes on a pour le but de répondre aux mêmes problématiques couverts par les standards WS-* qui sont :

- **Séparation des préoccupations** issu de *l'anglais separation of concerns (SoC)* est le fait de séparer un programme informatique en parties, afin d'isoler des composantes qui répondent à un problème spécifique de la problématique générale.

- **Visibilité** Comment apprend-on l'existence d'un tel ou tel service ?
- **Passage à l'échelle(*scalability*)** Le système sera-il capable d'évoluer avec le temps ?
- **Fiabilité** Les opérations, ont-elle des effets de bord ?

3.4.2.3 En pratique

Concrètement pour développer une API qui suit les principes REST il faut tout d'abord respecter des règles de nommage des URI.

Pour manipuler des ressources, la règle générale est d'utiliser des noms au lieu des verbes.

Ainsi, pour récupérer l'ensemble des produits on écrit :

```
| example.com/products
```

au lieu de

```
| example.com/getAllProducts
```

Pour plus de simplicité, on peut diviser les ressources en 4 catégories : *document*, *collection*, *stockage* et *contrôleur*.

Les documents représentent des objets singulières qui correspondent, par exemple, à une entrée en base de données. Il peut être vu comme une ressource dans une ressource de type collection. Pour nommer un document on utilise des noms en singulier :

```
| example.com/api/car-management/managed-cars/{car-id}
| example.com/api/client-management/clients/{id}
| example.com/api/user-management/users/admin
```

Les collections sont des dossiers contenant d'autres ressources. Un client peut proposer l'ajout d'une nouvelle ressource pour être rajouté à la collection. C'est à la collection de décider si elle veut accepter la nouvelle ressource ou pas. Ainsi, pour désigner une collection on utilise des noms en pluriel :

```
| example.com/api/car-management/managed-cars
| example.com/api/client-management/clients
| example.com/api/user-management/users
```

Quant aux **stockages**, ce sont des ressources gérés coté client et qui permet de stocker d'autres ressources d'une manière temporaire sans affecter le serveur. Le panier d'achat et la liste des favoris sont des exemples classiques des ressources de stockage :

```
example.com/api/cart-management/users/{id}/cart  
example.com/api/song-management/users/{id}/playlist
```

Finalement, les **contrôleurs** servent à exécuter des fonctions, avec les paramètres d'entrée et les valeurs de retour :

```
example.com/api/cart-management/users/{id}/cart/checkout  
example.com/api/song-management/users/{id}/playlist/play-random
```

3.4.2.4 Utilisation des méthodes HTTP

Les URI ne doivent pas indiquer qu'une opération CRUD est en train d'être effectuée. Les URI doivent seulement identifier la ressource. Au lieu de cela on utilise des méthodes HTTP comme suit :

```
//retourner tous les utilisateurs  
GET /device-management/user-management/users  
//créer un nouveau utilisateur  
POST example.com/user-management/users  
  
//retourne un utilisateur avec cet id  
GET /car-management/users/{id}  
//modifier un utilisateur connu  
PUT /car-management/users/{id}  
//supprimer un utilisateur connu  
DELETE /car-management/users/{id}
```

3.4.2.5 Exemples d'API du style REST

Afin de voir ce que cela donne en réalité, j'ai décidé d'analyser des services existantes qui suivent l'architecture REST. L'un des premiers exemples d'API REST traités était celle de twitter.com. Ainsi j'ai découvert que twitter permet de concevoir des applications qui interagissent avec le site de manière autonome.

Par exemple on peut récupérer des statuts des gens en temps réel contenant un certain mot-clé :

```
GET https://stream.twitter.com/1.1/statuses/filter.json?track=potatoe
```

Ou bien retourner l'ensemble de messages récents d'un utilisateur donnée :

```
GET https://api.twitter.com/1.1/statuses/user_timeline.json?user_id=43123
```

Un autre exemple intéressant est la navigation du site Amnesty International UK ⁶ :
Ainsi la collection des blogs

www.amnesty.org.uk/blogs/

regroupe des catégories (elles mêmes des collections) :

Fin de la peine capitale :

www.amnesty.org.uk/blogs/anti-death-penalty-project

Action étudiante :

www.amnesty.org.uk/blogs/student-action-network

Réflexion au sujet du traité sur le commerce des armes comme la ressource de la collection "action étudiante" :

www.amnesty.org.uk/blogs/student-action-network/arms-trade-treaty-last-my-reflections
et ainsi de suite...

3.4.3 Implementations existantes

Finalement, j'ai décidé d'analyser des frameworks REST qui permettent de développer des API suivant ce style architectural. Cela avait pour le but soit d'intégrer ce système dans le contexte d'OFBiz, soit

3.4.3.1 Camel

Apache Camel est un framework libre destiné à faciliter l'intégration des composants dans le contexte d'entreprise. Il fournit un certain nombre de DSL dont Rest DSL qui est destiné à aider les développeurs à définir les API REST. L'un des points particulièrement intéressant de Camel Rest DSL est sa similitude aux techniques utilisées dans l'OFBiz, notamment les DSL sont utilisées avec Java et XML à la fois. Malgré ces avantages, la tâche d'interrogation de ce framework s'avère trop coûteuse, car elle nécessiterai la réécriture d'une vaste partie relative aux `HTTPServlet` dans l'OFBiz.

3.4.3.2 JAX-RS

Un autre framework permettant de concevoir des API REST est JAX-RS de Apache CXF. La particularité de cet outil est la gestion des URI via les annotations :

```
@Path("/hello/world")
public class Hello {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayPlainTextHello() {
        return "Hello_World";
    }
}
```

6. Qui traite des sujets sur les droits de l'homme

Après avoir réussi à intégrer la serviette de JAX-RS dans le framework OFBiz, je me suis rendu compte que la technologie risque de ne pas être acceptée par la communauté car elle aussi implique un travail important de réécriture de l'existant. De plus, les annotations ne sont pas dans l'esprit du projet, donc cette idée a été abandonnée.

3.5 Analyse de l'existant

Lors de la découverte de OFBiz j'ai prêté particulièrement attention à la gestion des échanges web afin de soulever les points importants relatives à l'implémentation des services REST.

3.5.1 Gestion des application web dans OFBiz

Le groupe des classes d'OFBiz, responsables de la gestion des applications web se basent sur patrons de conception de "J2EE Presentation Tier". Notamment la classe `ContextFilter` qui assure la sécurité des requêtes, implémente le pattern `Decorating Filter` les vues générées par les moteurs de widgets et les FTL correspondent quand à eux au pattern `Composite View`.

3.5.2 ControlServlet

La classe qui est au cœur du traitement des requêtes, `ControlServlet` correspond au pattern `FrontController`. Une fois la requête a franchi les filtres de sécurité, la `ControlServlet` initialise l'environnement d'OFBiz, cela inclue les `Entity Delegator`, responsable du modèle de données, ainsi que `Service Dispatcher` qui exécute les services. Finalement, la session est initialisé et tous les informations utiles sont stocké dans cette dernière. Une fois l'environnement en place la classe délègue le traitement au `RequestHandler`

3.5.3 RequestHandler

Cette classe récupère l'ensemble des association des requêtes(`request-map`) définis sous format XML dans le fichier `controller.xml`. Il s'agit des association entre les URI et les vues optionnelles. A part les vues, les URI peuvent être associées avec un certain `Event`. Les `events` traitent la logique spécifique, en interagissant avec le moteur des entités ou en appelant un service à travers le moteur correspondant.

3.5.4 Controleur.xml

La structure des `request-map` est définie comme suit :

```
<request-map uri="checkLogin" edit="false">
  <description>Verify a user is logged in.</description>
  <security https="false" auth="false"/>
  <event type="java" path="LoginEvents" invoke="checkLogin"/>
  <response name="success" type="view" value="name" />
  <response name="error" type="view" value="login" />
</request-map>
```

3.5.5 API en cours

Après avoir analysé plusieurs applications web dans l'OFBiz je me suis rendu compte que le style actuel correspond à celui de *Remote Procedure Call* (RPC), connu pour ses problèmes de performance et du passage à l'échelle.

3.5.6 Mécanisme de résolution des URI

Les URI de `request-map` ne contiennent pas plus d'un seul mot (d'où RPC) comme par exemple :

```
| uri="createEmptyShoppingList "
```

ou bien

```
| uri="removeFromShoppingList "
```

Ainsi, la résolution d'URI consiste tout simplement à vérifier la présence de la clé dans l'objet `Map` contenant l'ensemble des définitions du `controller.xml`.

3.5.7 *OverrideView()* et le conflit avec les URI segmentées

Une des spécificités est la gestion des URI segmentés comme dans

```
| /toto/titi/tata
```

sont traités par la méthode `String getOverrideViewUri(String path)` qui permet d'appeler une vue sans effectuer de traitement. Le résultat de cela est que toutes les URI

segmentées, incontournables à la définition des API du style REST, déclenchent cette méthode qui, à son tour, retourne une erreur.

3.6 Analyse des besoins et attentes de la maîtrise d'ouvrage

3.6.1 Besoins d'évolution

Avec la popularité du style architectural REST, plusieurs intégrateurs de OFBiz ont montrés leur l'intérêt à ce concept. Donc afin de répondre aux besoins des utilisateurs et afin d'assurer une meilleure évolutivité des projets basés sur OFBiz à l'avenir, la communauté a lancé une discussion Jira où des nombreuses suggestions de concept ont été proposées.

Après avoir analysé les tentatives et les problématique d'implémentation de services REST dans OFBiz je suis arrivé à la conclusion qu'une solution de bas niveau répondrais au mieux au besoins des intéressés. De plus il était évident que la solution doit être développée en parallèle du système actuel, c'est à dire sans provoquer le dysfonctionnement des mécanismes en cours.

3.7 Réalisations techniques

3.7.1 Choix vers URITemplate

Même si après l'analyse de la librairie JAX-RS de CXF, j'ai pris la décision de ne pas poursuivre ce chemin, certaines classes m'ont permis de trouver une inspiration en terme de concept qui peut être utilisé dans OFBiz. Ainsi, la classe URITemplate permet de créer des modèles de URI avec les variables.

Exemple 1 : Supposons le modèle de URI suivant :

```
| /foo/bar/{baz}/{qux}
```

Est mis en correspondance à travers la méthode `.match()` avec l'URI suivant :

```
| /foo/bar/toto/titi
```

Par conséquent, notre modèle contient désormais deux entrées de variables avec les clés `[baz, qux]` qui correspondent aux valeurs `[toto, titi]` respectivement.

Un autre point à noter est la possibilité d'avoir des variables adaptables au moyen d'une expression régulière. Ainsi le modèle

| /foo/{v1:\\d}/{v2}

Peut être mis en correspondance uniquement avec les URI qui ont un chiffre⁷ en premier argument v1.

3.7.2 Librairie CXF

Donc, vu la flexibilité de `URITemplate`, j'ai décidé de l'utiliser au sein d'OFBiz. Naturellement, l'ajout des dépendances dans un vaste projet comme OFBiz doit être fait avec précaution. La raison : des libraires deviennent souvent dépréciées ou changent à tel point que cela provoque la nécessité de réécrire des parties importantes du projet.

Après l'analyse des dépendances de OFBiz, j'ai remarqué la présence de l'outil Apache Tika qui est utilisé dans le OFBiz pour produire et analyser des fichiers du type XLS ou PDF. Ainsi, Tika contient la librairie CXF ce qui m'a donné la liberté d'utiliser cette dernière.

3.7.3 Intégration en parallèle avec le système existant

Finalement, j'ai intégré ce système de résolution des URI basé sur la classe `URITemplate` dans le `RequestHandler` de OFBiz. Cela était fait en modifiant légèrement le mécanisme existant. Concrètement, au début de la résolution d'une URI, l'ensemble des `request-map` est délégué au nouveau mécanisme qui, en cas de succès⁸ redirige soit vers la vue, soit vers le `Event` correspondant, indiqués dans la définition de `request-map`. En cas d'absence de "match" via `URITemplate`, la suite de la méthode exécute des traitement de manière traditionnelle.

3.7.4 Nouveau controller.xml

Grâce à ces modifications, nous avons la possibilité de définir des points d'entrée d'une manière plus flexible :

```
<request-map uri="products/pets/{product-id:\\d}" method="PUT">
...
  <response name="success" type="view" value="viewProduct"/>
...
</request-map>
```

Qui permet de modifier d'un produit animalier identifié un son *product-id* numérique, en appelant la méthode HTTP PUT, par exemple.

7. La lettre "d" signifie le mot l'anglais *digit* - "chiffre" en français

8. Ayant validé l'URI de la requête avec via la méthode `.match()` avec l'une des modèles construits à partir de l'ensemble de "request-map"

3.7.4.1 Les collisions d'URI

Cependant, suite à l'introduction du nouveau système, on a rencontré une difficulté mineure quant à la définition des modèles avec les variables.

Par exemple lors de la définition de plusieurs `request-map` successives dans un même `controller.xml` comme suit :

```
//afficher un stock identifié par la valeur de sa clé primaire
```

```
|<request-map uri="stock/{primary-key-id}"
```

```
//retourner le formulaire de modification d'un stock
```

```
|<request-map uri="stock/form"
```

```
//afficher les relations entre les stock
```

```
|<request-map uri="stock/relations" >
```

Cette situation suppose que dans la première `request-map` il est impossible d'avoir des clés primaires avec la valeur `"form"` ou `"relations"`. La solution pour ce genre d'exception⁹ est l'exclusion des mots spécifiques via des expressions régulières :

```
| ... uri="stock/{primary-key-id: (?!(form)|(relations)).*}"
```

3.7.4.2 Testes Mockito

Finalement, un point intéressant était l'écriture des testes unitaires en utilisant la librairie Mockito. Des principes et des bonnes pratiques de ces testes m'ont été montrés par mon tuteur de stage.

3.7.5 Modification de la partie "Administration : gestion des entités" (entitymaint)

Afin de montrer l'utilité du nouveau système, j'ai décidé de modifier une partie front existante ainsi que les points d'entrée avec lesquelles elle interagissait.

9. Qui doivent a priori être relativement rares

3.7.5.1 Choix de la partie illustrative

Pour pouvoir tester tout la puissance des méthodes classiques du protocole HTTP j'ai du choisir une partie qui permettait d'effectuer l'ensemble des opérations CRUD. Par conséquent, le choix était porté sur la partie administrative qui correspond à la gestion des entités : `entitymaint`.

3.7.5.2 Difficultés de modification

Vu que l'OFBiz a été construit sans tenir compte du REST, ce genre de modification impliquent des changement profondes de la structure du code, allant jusqu'à la réécriture complète. Donc, disposant un temps limité, j'ai du avoir recours aux certains compromis comme utilisation d'un même formulaire pour la création et la modification d'une entité.

3.7.5.3 Clés composées

Un autre compromis mineur qui devait être fait était la présence d'un nombre variable des clés primaires qui identifiait l'entité-ressource. Par conséquent, la variable qui correspond à l'ensemble des valeurs des clés primaires doit pouvoir accepter un nombre quelconque des clés. Pour cela, on utilise une expression régulière :

```
<request-map uri="entity/{entityName}/{pkValues:_.*}" method="get">
...
<request-map uri="entity/{entityName}/{pkValues:_.*}" method="delete">
...
```

3.7.6 Stateless

Un des concepts fondamentaux du REST est l'absence de stockage de l'état coté serveur.

3.7.6.1 Les réalisation par la communauté

OFBiz, n'étant pas Stateless¹⁰ utilise la session comme un moyen d'authentifier les utilisateurs. Ainsi, les tentatives de supprimer la notion d'état ont été faites par la communauté. Malheureusement, des nombreux concepts se basent sur la session et la suppression serait trop coûteuse. Au lieu de la supprimer, la communauté à implémenté l'authentification en utilisant les JSON Web Tokens, mais derrière, la session est toujours utilisée.

10. Sans état - sans conserver des informations

3.7.7 RESTClient pour la communauté

3.7.7.1 Généralisation de code

3.7.7.2 Correction d'incohérences

Chapitre 4

Conclusion

4.1 Lien avec les connaissances obtenu lors de la formation universitaire

METTRE DANS LA CONCLUSION

4.1.1 MVC

4.1.2 Servlet

4.1.3 raeeMarker -JSP

4.1.4 Notion d'entité - Symony

4.1.5 Routage

4.1.6 Gradle Maven

4.1.7 Testes Unitaires

L'utilité des testes : surtout démontrer l'utilisation

Bibliographie

- [1] Auteur Ailleurs. Titre3. <<http://www.url2.org/>>, 2014. [Online; accessed 16-January-2014].
- [2] authoe. Jdbc. <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>, date.
- [3] authoe. Jdbc. <https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/>, date.
- [4] Auteur Autre. Titre2. <<http://www.url1.org/>>, 2014. [Online; accessed 16-January-2014].
- [5] Auteur Elle. Titre5. <<http://www.url4.org/>>, 2014. [Online; accessed 16-January-2014].
- [6] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures.
- [7] groo. groovy.
- [8] Auteur Livre1. *Titre Livre1*. Editeur1, 2014.
- [9] Auteur Lui. Titre4. <<http://www.url3.org/>>, 2014. [Online; accessed 16-January-2014].
- [10] Auteur Untel. Titre1. <<http://www.url0.org/>>, 2014. [Online; accessed 16-January-2014].