Implémentation d'une API HTTP de style REST pour Apache OFBiz

Mathieu Lirzin

3 septembre 2018

Résumé

Ceci est un rapport de stage de fin d'études effectué au sein de la société Néréide. Ce stage a consisté à mettre en place une API HTTP basée sur le style architectural REST pour les échanges entre différents composants logiciels du framework Apache OFBiz afin de réorienter l'Architecture Orienté Service (SOA) de ce framework.

Table des matières

Introduction

Ceci est un rapport de stage de fin d'études de Master 2 Informatique de l'Université de Bordeaux effectué au sein de la société Néréide pendant une durée de 5 mois entre le 2 Avril et le 30 Août 2018.

Le sujet de ce stage a consisté à mettre en place une API HTTP basée sur le style architectural REST pour les échanges entre les différents composants logiciels du framework Apache OFBiz. Cela a consisté dans une première phase à réaliser un état de l'art de l'utilisation de ce style architectural, puis à étudier les principes d'architectures orientées services (SOA) utilisés dans le framework Apache OFBiz. Dans une deuxième phase l'API HTTP basée sur le style REST a été implementée au sein du framework Apache OFBiz en suivant les principes SOA. Ces améliorations ont été réalisées en interaction avec la communauté Apache OFBiz afin de permettre leurs intégrations dans les prochaines versions.

Nous allons dans un premier temps faire une présentation de la société d'accueil. Dans un deuxième temps nous présenterons le framework sur lequel s'est basé le travail effectué lors de ce stage qui sera présenté dans un troisième temps.

Chapitre 1

Présentation de l'entreprise

Néréide est une société de service en logiciel libre fondée en 2004 spécialisée dans l'intégration du progiciel de gestion intégré Apache OFBiz. Il s'agit d'une société coopérative et participative (SCOP) dont le siège social est situé à Tours. Les activités de la société se décomposent en différents axes :

Dévéloppement spécifique L'intégration du framework OFBiz implique généralement des développements spécifiques pour s'adapter au différents composants du système d'information existant.

Maintenance et support applicatif (TMA) Prestation de support et de maintenance corrective et/ou évolutive pour les développements spécifiques précédemment réalisés.

Administration système Mise en place d'un élément système, d'un réseau. Intégration et migration des composants systèmes.

Néréide s'appuie sur le réseau Libre Entreprise dont elle fait partie pour proposer des services tels que de l'hébergement et de la formation. L'équipe est actuellement composée de 11 personnes, ayant pour point commun une expertise fonctionnelle sur OFBiz.

Chapitre 2

Présentation de OFBiz

Apache *Open For Business* (OFBiz) est un framework Web ayant pour but de faciliter la construction de progiciels de gestion intégré. Cet objectif est réalisé au moyen d'un modèle de données générique et d'un ensemble de modules dédiés à des tâches communes à la plupart des entreprises, telles que la comptabilité, la facturation, ou la gestion des stocks.

OFBiz a été créé en 2001 par David E. Jones et Andy Zeneski dans l'idée de développer une solution commune aux différentes problématiques rencontrées par les entreprises en utilisant un modèle de développement en logiciel libre. En janvier 2006 OFBiz a été accepté dans l'incubateur de la fondation Apache avant d'être accepté en tant que projet à part entière en décembre 2006.

2.1 Architecture générale

Techniquement OFBiz est basé sur la plateforme Java et sur l'utilisation de *langages spécifiques au domaine* (DSL) basés sur des grammaires XML. Concenant le coeur du framework, les communications HTTP sont gérées par des servlets [2] et les communications avec les bases de données se font au moyen de l'API Java JDBC. Au dessus de cela, la déclaration des routes HTTP, des écrans et des services se fait dans des fichiers XML faisant référence à des implementations de services généralement en Java et Groovy. Les écrans sont quant à eux basés sur Freemarker qui est un langage de templating.

OFBiz est structuré en séparant le framework, les applications, et les plugins. Le framework fournit les mécanismes de communications réseau et d'interfacage entre les différentes applications. Les plugins sont des applications spécifiques qui dépendent des applications de base mais dont aucune application de base ne dépend. OFBiz est décomposé en plusieurs composants qui sont regroupés en plusieurs catégories ou meta-composants. Ainsi les composants fondamentaux définissant le socle de l'architecture d'OFBiz sont contenus dans l'ensemble de composants *Framework*. Les composants fonctionnels fondamentaux tels que la comptabilité, la facturation, ou la gestion des stock sont regroupés dans l'ensemble de composants *Applications*. Les composants spécifiques sont eux mis dans la catégorie *Plugins*. Un point important dans la relation entre ces différents composants est qu'il est important que les relations de dépendances entre les méta composants soient respectés comme cela est représenté sur la figure 2.1. Cela permet entre autre de développer les plugins dans un dépôt externe. Nous allons dans les sections suivantes détailler le contenu et l'architecture des deux méta-composants *Framework* et *Applications*.

2.2 Le framework

Le framework fournit les éléments fondamentaux définissant l'architecture d'OFBiz ainsi que des outils de maintenance permettant d'intéragir avec une instance d'OFBiz en cours d'exécution. Nous présentons ici les concepts et élements fondamentaux pour comprendre l'organisation de l'architecture du framework.

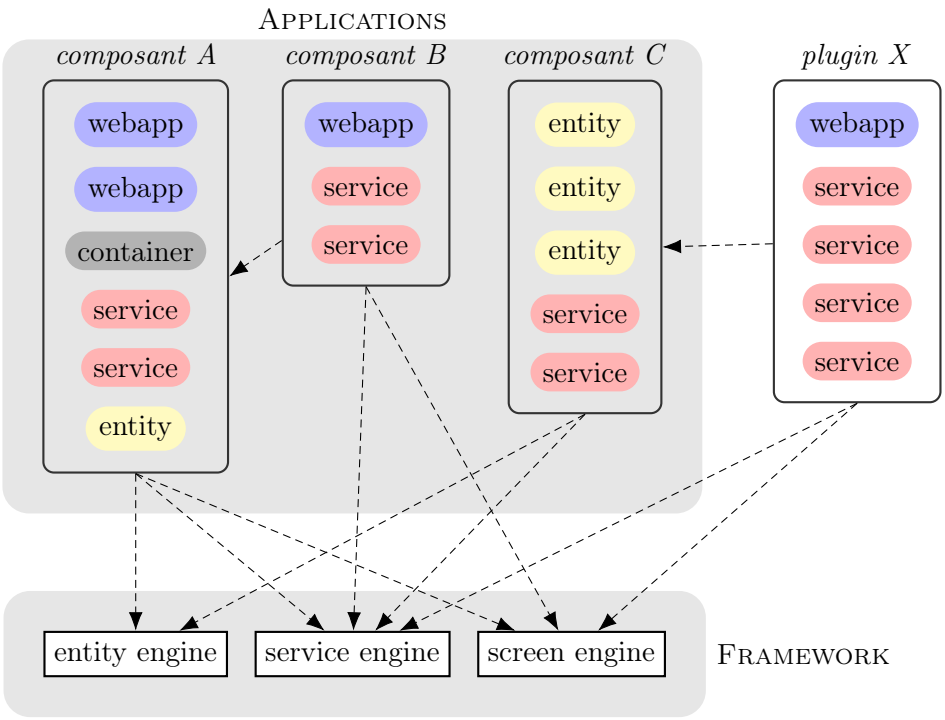

Figure 2.1 – Dépendances entre les composants et le framework

2.2.1 Container

Un container est l'élément de plus bas niveau que définit OFBiz. Un container est un processus ayant un cycle de vie, c'est à dire pouvant être initialisé, démarré et arreté. Concrètement cela est représenté par l'interface `Container` représentée sur la figure 2.2. OFBiz ne définit par défaut qu'un seul container dans un fichier `ofbiz-container.xml` et il est possible pour chacun des composants de définir leurs propres containers. L'intérêt de cette abstraction est de permettre de lier l'exécution un daemon externe au lancement de OFBiz. Ainsi le plugin OFBiz Camel définit un container pour le runtime de Apache Camel en charge de faire communiquer les différents points de sorties. Un container est paramétré par un ensemble de phases décrites dans l'attribut `loaders` du container. Cet attribut détermine à quelles phases ce container doit être démarré. Les trois phases suivantes sont définies :

main la phase d'exécution de ofbiz

load-data la phase de chargement des données dans la base de données

test la phase de lancement des tests d'intégrations

2.2.2 Composants

Un composant est une abstraction permettant de séparer les différentes fonctionnalités de OFBiz. Un composant est une agglomération de containers, entités, services, écrans, et applications Web. Un composant est définit au moyen d'une structure XML stockée dans le fichier `ofbiz-component.xml` qui est à la racine du répertoire contenant l'ensemble des définitions que le composant contient. La figure 2.3 montre l'exemple du composant *Webtools* qui fait référence à deux fichiers contenant des définitions d'entités, à un fichier de service et à une application Web.

L'élément `resource-loader` définit de quelle manière ce composant sera chargé par OFBiz. L'élément `classpath` définit le chemin vers des ressources telles que les traductions d'étiquettes contenues dans le dossier pointé par l'attribut `location`. Concernant les éléments `entity-resource` il est possible d'avoir des types "model" ou "data". La différence étant que "model" contient des

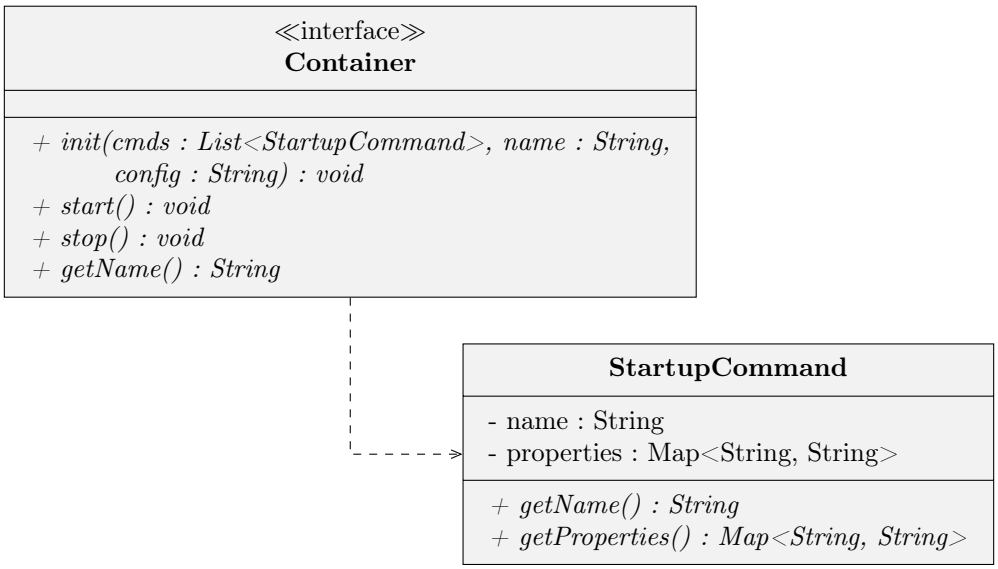

FIGURE 2.2 – Définition du type container

```
<ofbiz-component name="webtools"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://ofbiz.apache.org/dtds/ofbiz-component.xsd">
  <resource-loader name="main" type="component"/>
  <classpath type="dir" location="config"/>

  <entity-resource type="data" reader-name="seed" loader="main"
                   location="data/WebtoolsSecurityPermissionSeedData.xml"/>
  <entity-resource type="data" reader-name="demo" loader="main"
                   location="data/WebtoolsSecurityGroupDemoData.xml"/>
  <service-resource type="model" loader="main" location="servicedef/services.xml"/>
  <webapp name="webtools"
          title="WebTools"
          menu-name="secondary"
          server="default-server"
          location="webapp/webtools"
          base-permission="OFBTOOLS,WEBTOOLS"
          mount-point="/webtools"/>
</ofbiz-component>
```

FIGURE 2.3 – Définition du composant Webtools

```
<site-conf xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns="http://ofbiz.apache.org/Site-Conf"
      xsi:schemaLocation="http://ofbiz.apache.org/Site-Conf...">
  <include location="component://common/webcommon/WEB-INF/common-controller.xml"/>
  <include location="component://common/webcommon/WEB-INF/portal-controller.xml"/>
  <include location="component://common/webcommon/WEB-INF/security-controller.xml"/>
  <include location="component://common/webcommon/WEB-INF/tempexpr-controller.xml"/>
  <description>WebTools Site Configuration File</description>
  <handler name="ftl" type="view"
           class="org.apache.ofbiz.webapp.ftl.FreeMarkerViewHandler"/>
  ...
  <request-map uri="ping">
    <event type="service" invoke="ping"/>
    <response name="error" type="view" value="ping"/>
    <response name="success" type="view" value="ping"/>
  </request-map>
  ...
  <view-map name="ping" type="ftl" page="component://webtools/template/Ping.ftl"/>
  ...
</site-conf>
```

FIGURE 2.4 – Définition du contrôleur de Webtools

définitions d'entités et "data" contient des exemples de données qui servent à fournir des données de démonstration.

2.2.3 Applications Web

Chaque composant peut définir des applications web qui sont une extension du modèle des servlets et sont montés dans un serveur d'applications partagé par tous les composants. Généralement un composant possède au plus une application mais il est possible d'en avoir plusieurs dans un même composant. Dans le cas du méta-composant *Framework* il n'y a qu'une seule application Web qui est associée au composant *Webtools*. Les composants contenus dans *Applications* ont presque tous au moins une application Web associés. Il est a noter que bien qu'OFBiz soit basé sur l'architecture des servlets il n'est pas possible de déployer les différentes applications Web dans un serveur d'application externe dans la mesure où le processus de compilation d'OFBiz embarque dans un même JAR l'ensemble des applications web ainsi que le serveur d'application. Classiquement la définition des routes pour les servlets se fait dans un fichier `web.xml`. Dans le cas d'OFBiz ce routage est délégué à un fichier `controller.xml` associant les différents traitements à des routes HTTP ainsi que les écrans qui seront retournés au terme de ces traitements. Seules les requêtes `POST` et `GET` sont traitées du fait que le client est uniquement prévu pour être un navigateur. Il n'y a pas de différences dans le traitement des requêtes. Peu importe la méthode choisie cela a pour effet d'appeler l'événement associé. La figure 2.4 présente une partie du fichier `controller.xml` du composant *Webtools*.

2.2.4 Moteur d'entités

Un point crucial d'un framework comme OFBiz est l'accès aux bases de données. Pour cela OFBiz possède un moteur d'entités pour communiquer avec les différents systèmes de base de données relationnelles (SGBD). Ce moteur d'entités permet de définir des schémas adaptés aux différent SGBD à partir d'un modèle entité-relation avec des types de données génériques associés à des types concrets adaptés à l'implémentation cible. Un deuxième aspect est la construction de requêtes SQL au sein des programmes Java et Groovy au moyen d'un DSL utilisant le pattern *builder* [8]. Lorsque ces requêtes sont envoyées, leur résultat retourne des valeurs « actives » c'est à dire gardant une référence sur la connexion à la base de données pour pouvoir sauvegarder en

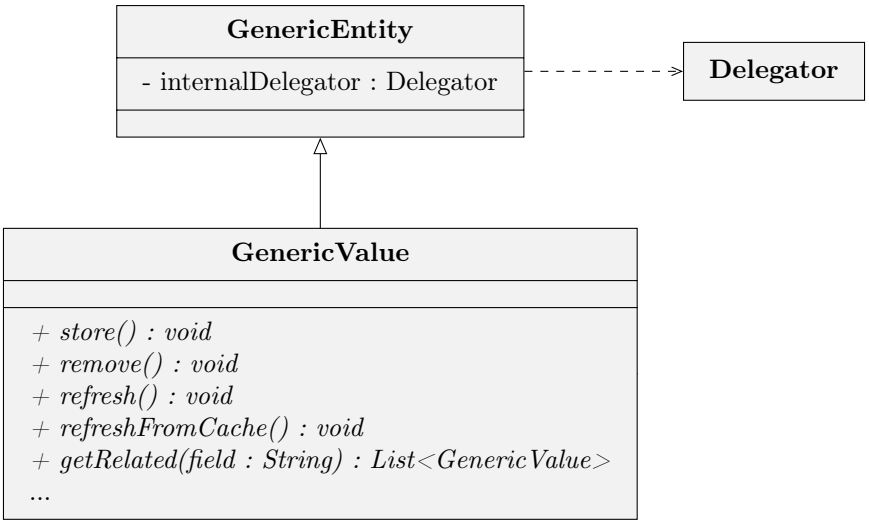

FIGURE 2.5 – Définition du type valeur générique

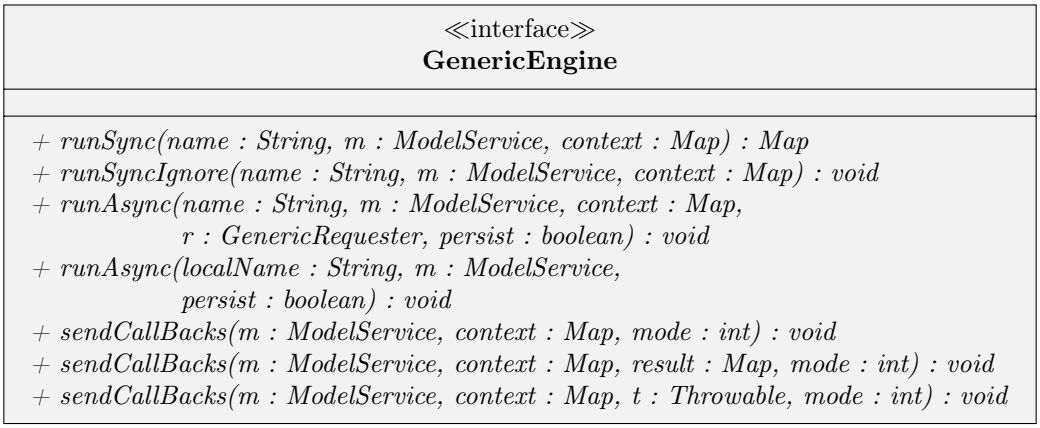

FIGURE 2.6 – Définition du type service

base les modifications faites localement sur ces objets. Cela correspond au pattern architectural *active record* [6]. La figure 2.5 montre les relations entre ces valeurs « actives » représentées par la classe `GenericValue` et la connexion à la base de données représentée par la classe `Delegator`.

2.2.5 Moteur de services

OFBiz est basé sur une architecture orientée service (SOA). La notion de moteur de service est définit par l'interface Java `GenericEngine` représentée sur la figure 2.6. Un service peut être appelé en lui passant un contexte en entrée, il renvoie alors une *Map* permettant de récupérer le type de retour conventionnellement nommé "success", "error", ou "failure" ainsi que les données renvoyées par le service. Il est possible d'exécuter un service soit de manière synchrone avec les méthodes `runSync` ou `runSyncIgnore` soit de manière asynchrone avec les méthodes `runAsync`. Les méthodes `sendCallBacks` permettent d'exécuter des traitements associés de manière *ad hoc* à un service. Ces méthodes sont considérées comme obsolètes et ont été remplacées par l'utilisation des *Event Condition Access* (ECA) décrite plutard. Dans la pratique les services sont exécutés la plupart du temps de manière synchrone. Un service a accès au moteur d'entités pour manipuler les données et au moteur de services pour appeler d'autres services.

Il existe plusieurs implémentations de cette interface parmi lesquelles on trouve les classes `EntityAutoEngine`, `StandardJavaEngine`, et `GroovyEngine`. Cela permet d'écrire des services dans différents langages de programmation ou même par des services distants ce qui est le cas du

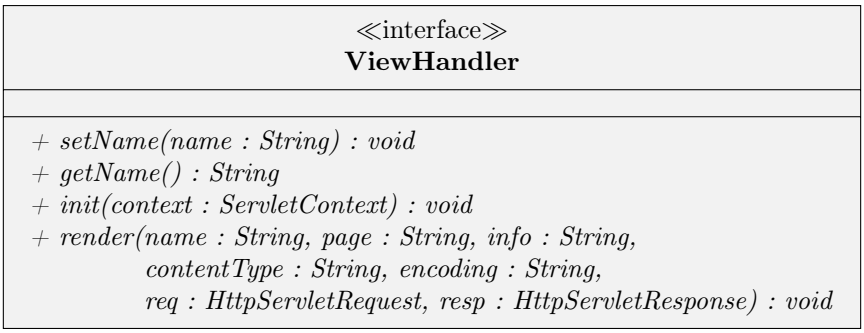

FIGURE 2.7 – Définition du type gestionnaire de vues

moteur de services `XMLRPCClientEngine`. Il est possible pour les plugins de fournir leur propre implémentation de services afin de faire communiquer OFBiz avec d'autres composants logiciels.

Les ECA permettent d'exécuter des traitements à différentes phases de l'exécution d'un service. Les différentes phases existantes sont "auth", "in-validate", "out-validate", "invoke", "commit", et "return".

2.2.6 Moteur d'écrans

Afin de fournir des écrans, OFBiz possède un DSL pour définir des écrans, formulaires, widgets auxquels est associé un mécanisme de rendu générique permettant de générer différents formats de sortie tels que HTML, XML, CSV, ou XLS. Ce mécanisme de rendu est défini par l'interface `ViewHandler` définie sur la figure 2.7. L'implémentation principale utilisée par la plupart des moteur d'écrans existant est la classe `MacroScreenViewHandler`. Elle délègue le rendu au mécanisme de templating Apache Freemarker. Dans le cas du moteur d'écran HTML l'inclusion des scripts Javascript permettant d'apporter du dynamisme et d'effectuer des appels AJAX [9] à des services OFBiz est fait directement dans la définition des macros Freemarker. Chaque vue (`view-map`) définie dans le fichier `controller.xml` doit faire référence explicitement à l'implémentation du moteur d'écran de son choix dans l'attribut `type` qui fait référence à un élément `handler`.

```
<handler name="screen" type="view"
         class="org.apache.ofbiz.widget.renderer.macro.MacroScreenViewHandler"/>
<view-map name="login" type="screen"
          page="component://common/widget/CommonScreens.xml#login"/>
```

Sauf exception l'ensemble de ces éléments sont définis dans le fichier `common-controller.xml`. Il peut définir à la fois des gestionnaires de rendu d'écran et des gestionnaires de requêtes HTTP couramment appelés *événements* dans le contexte d'OFBiz.

2.3 Les applications

Après avoir décrit les éléments définis dans le méta-composant *Framework*, nous présentons le méta-composant *Applications* contenant des composants métiers c'est à dire des composants associés à un type d'application fonctionnel. Ainsi nous avons les composants suivants :

accounting définit des services et écrans pour effectuer de la comptabilité.

commonext est une extension du composant *common* du *Framework* contenant des dépendances fonctionnelles

content définit des services et écrans pour créer des contenus divers tel que des articles, photos, vidéos, pour par exemple maintenir un blog.

datamodel contient la définition des entités applicatives et des relations entre ces entités.

humanres définit des services et écrans pour la gestion du personnel.

manufacturing définit des services et écrans pour la gestion des processus industriels et de l'entreprosage.

marketing définit des services et écrans pour la communication d'entreprise.

order définit des services et écrans pour le traitement des commandes.

party définit des services et écrans pour la gestion des partenaires et clients.

product définit des services et écrans pour la gestion du catalogue de produit et services.

securityext est une extension du composant *security* qui définit des services ayant des dépendances fonctionnelles.

workeffort définit des services et écrans pour la gestion de projet et le suivi de tâches (pointage).

Cet ensemble de composants est le coeur fonctionnel d'OFBiz, il sert de socle pour les composants spécifiques qui sont définis au niveau des plugins.

Chapitre 3

État de l'art

Nous présentons dans ce chapitre le contenu de l'étude faite lors de la première phase de ce stage. Cette étude à consisté d'une part à étudier de manière théorique le style architectural REST notamment au travers des travaux de Roy Fielding. D'autre part cela a consisté à étudier des solutions existantes permettant de faciliter le développement de service REST. Ces deux aspects ont eu pour objectif d'avoir une bonne compréhension des enjeux de ce style architectural, et d'avoir une bonne connaissance des solutions existantes pour soit en réutiliser une, soit trouver de l'inspiration pour l'implémentation développée dans le cadre d'OFBiz.

3.1 Representational State Tranfer

Representational State Transfer (REST) est un style architectural ayant émergé au début de l'année 1995. Il s'agit d'un modèle idéalisé décrivant comment le *Web* devrait fonctionner [5]. Il a été affiné pendant 5 ans jusqu'à être décrit en détails par Roy Fielding dans sa thèse de doctorat [4] en 2000. Ce travail a eu une forte résonance dans les milieux non-académiques du fait de l'essor du Web, de ses qualités en terme de passage à l'échelle, mais également du fait de sa simplicité.

Un style architectural est un ensemble de contraintes placées sur une architecture logicielle c'est à dire une abstraction décrivant les composants, les connecteurs, ainsi que les données présentes au sein d'un système [13]. REST est composé des 6 contraintes représentées par les noeuds grisés du graphe de dérivation de la figure 3.1. Chacune de ces contraintes est explicitée dans la description suivante :

Client-Serveur (CS) Cette contrainte permet la séparation des préoccupations entre le(s) serveur(s) et le(s) client(s). Typiquement il est important de ne pas lier une interface utilisateur aux services rendus par le serveur, pour permettre la réutilisabilité des services et la variété des interfaces utilisateurs. Dans un contexte distribué cela permet au client et au serveur d'évoluer indépendamment.

Serveur sans état (CSS) Chaque requête envoyée par le client doit pouvoir être traitée de manière indépendante. Cela impose de transmettre dans chacune des requêtes les données permettant d'authentifier le client. Cela rend l'architecture robuste dans la mesure où cela limite les contraintes d'ordre dans l'envoi de messages. Dans un contexte avec plusieurs serveurs cela permet de passer simplement à l'échelle. Cependant cette redondance d'informations entre les requêtes impose un sur-coût de transfert de données.

Possible mise en cache ($) Pour contrecarrer ce sur-coût, REST impose de pouvoir spécifier dans les messages si les informations qu'ils contiennent peuvent être gardées en cache et ainsi limiter le nombre de messages transmis sur le réseau.

Interface uniforme (U) Cette contrainte est centrale à REST et consiste dans la manipulation des ressources uniquement par leur(s) représentation(s), l'utilisation de messages

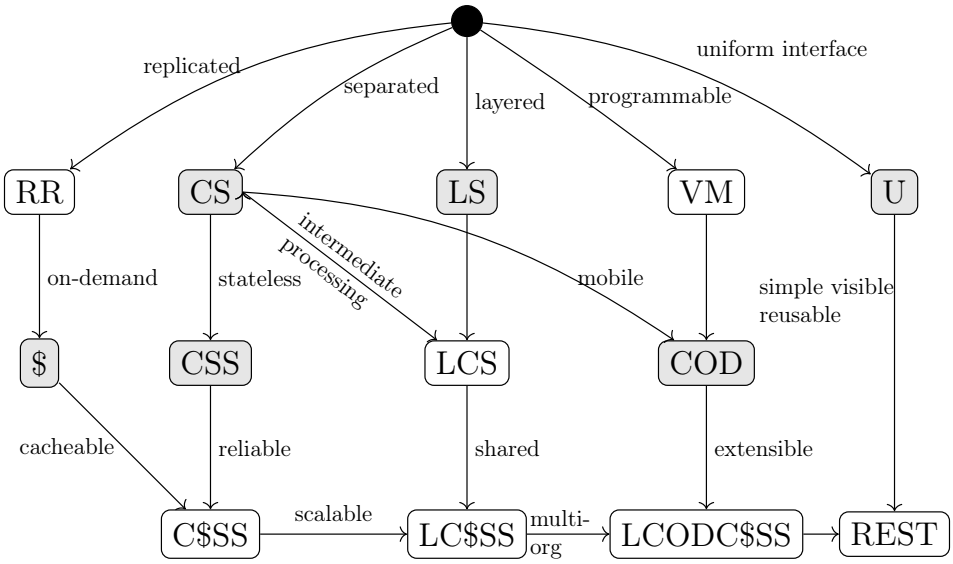

FIGURE 3.1 – Dérivation du style architectural REST

auto-descriptif, et l'utilisation de *l'hypermédia comme moteur de l'état de l'application*[1]

En couches (LS) Cette contrainte permet de limiter la complexité de l'architecture. En revanche elle a l'inconvénient d'avoir un sur-coût en terme d'exécution.

Code à la demande (COD) Ceci est une contrainte optionelle permettant au serveur d'étendre les fonctionnalités du client.

Cette combinaison de contraintes permet d'obtenir les propriétés architecturales suivantes :
— Performance
— Passage à l'échelle
— Simplicité
— Évolution
— Visibilité
— Portabilité
— Fiabilité

Ce style architectural s'oppose principalement au style *Remote Procedure Call* (RPC) qui souffre d'un problème de performance et passage à l'échelle. Ce style RPC se retrouve dans le cadre du Web dans l'utilisation du protocole SOAP [1] et correspond au type des appels de services OFBiz. Malgré son succès REST reste assez mal compris ainsi son application dans le cadre de l'architecture Web que nous détaillons dans la partie suivante est souvent mal comprise et on perd ainsi les propriétés d'évolution, portabilité, et fiabilité.

3.2 Architecture Web

L'architecture Web est une instance du style architectural REST. En fait REST a été conceptualisé conjointement au World Wide Web pour permettre de décrire de manière abstraite les enjeux d'un tel système. Nous présentons ici les différents composants de cet architecture que sont le protocole HTTP, les URI, et les formats de données tels que HTML, XML, et JSON.

3.2.1 Protocole HTTP

Le protocole *Hypertext Transfer Protocol* (HTTP) est un protocole texte utilisé sur le Web pour transmettre les ressources HTML, CSS, Javascript. Ce protocole appartient à la couche

1. plus connu sous sa forme anglaise *Hypermedia As The Engine Of Application State* (HATEOAS)

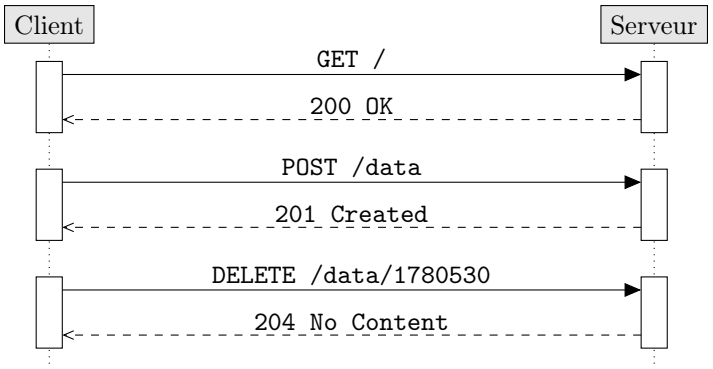

FIGURE 3.2 – Protocole HTTP

application du modèle OSI et se base sur le protocole TCP dans la pile TCP/IP. Il a un rôle particulier dans l'architecture *Web* puisqu'il sert à la fois à la communications entre les composants *Web* et est l'unique protocole de cette architecture intégrant la notion de représentations et de ressources. L'accès aux ressources se fait au au travers d'une URI et au moyen de 4 méthodes principales `GET`, `POST`, `HEAD`, `DELETE` mais également de 5 autres méthodes moins courantes `OPTIONS`, `CONNECT`, `TRACE`, `PUT`, `PATCH`. La figure 3.2 montre un exemple de communication entre un client et un serveur HTTP. Les requêtes envoyées au serveur prennent la forme d'une ligne de requête, un ensemble d'entêtes, et optionellement une charge utile. Ainsi cet exemple de communication pourrait inclure la requête de la figure 3.3 et la réponse de la figure 3.4.

```
POST /data HTTP/1.1
Host: www.example.com
Content-Length: 14

lorem ipsum
```

FIGURE 3.3 – Requête HTTP

```
HTTP/1.1 201 Created
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: application/json; charset=UTF-8
Content-Length: 42

{
  "msg" : "Resource successfully created",
  "href" : "/hello/data/1780530"
}
```

FIGURE 3.4 – Réponse HTTP

Le fait que les méthodes et les codes de retours soient contraints par le protocole permet aux agents de faire des suppositions sur la manière de traiter la requête et la réponse. Par exemple la méthode `GET` a comme propriété d'être *nullpotent* c'est à dire qu'elle ne doit pas modifier l'état des ressources, et la méthode `PUT` a la propriété d'être *idempotent* c'est à dire que la réception de plusieurs message ne doit produire qu'une seule action. La méthode `POST` n'a aucune de ces propriétés. La sémantique fixe attachée aux méthodes permet aux différents mécanismes de cache d'agir de manière efficace au sein de l'architecture Web et donc d'apporter les propriétés de performance et de passage à l'échelle [3].

3.2.2 Uniform Resource Identifier

Les *Uniform Resource Identifiers* (URI) servent à nommer toute sorte de ressources (document, images, service, ...) [11]. Les URI ne sont pas des emplacements et ne sont donc pas liées à un serveur spécifique ce qui permet d'utiliser des proxies. Les URI permettent uniquement d'accéder à des représentations et non directement à des ressources. Cette distinction permet une forme de polymorphisme en associant plusieurs représentations à une même ressource. En pratique l'accès à ces représentations se fait au moyen de l'entête `Accept` présent dans la requête émise vers une URI, qui est dispatchée alors vers la représentation la plus adaptée. Les URI peuvent à la fois référencer des ressources statiques ou des ressources dynamiques. Une ressource R est une fonction $M_r(t)$ associant à un temps t une entité ou valeurs. L'ensemble des entités ou valeurs appartenant au co-domaine sont équivalentes en terme de leur sémantique. Même si ces ressources pointent sur la même valeur il s'agit bien de 2 ressources distinctes. Ce qui identifie une ressource c'est la sémantique associé à son URI.

3.2.3 Formats de données

Lors d'une requête, il est possible pour le client de spécifier le type contenu souhaité au moyen du header `Accept` cependant il faut que le serveur recevant cette requête soit en mesure de fournir le format de donnée souhaité. Dans la pratique les formats XML et JSON sont les plus courants pour la transmission de données structurées. Un point important est que le format représente les données transmises sous forme de structures arborescentes pouvant être mis en correspondance avec les structures de données présentes dans la plupart des langages de programmation. Afin de permettre l'implémentation de l'hypermédia qui est décrit dans la section suivante, il est également important que le format choisi puisse être associé à une sémantique. Cela est nativement le cas par exemple du format HTML mais pas de JSON.

3.2.4 Hypermédia

La notion d'*hypermédia* est l'extension de l'*hypertexte* à d'autres médias tels que les images, le son, et les vidéos. Il désigne un réseau d'information accessible de manière non-linéaire et interactive au moyen de liens. Cette notion est cruciale dans la réalisation d'une API HTTP RESTful bien qu'elle soit trop souvent négligée. En suivant les principes de l'architecture REST, les URI ne devraient pas constitué l'API. L'API devrait être idéalement un point d'entrée unique avec l'ensemble des ressources accessibles en tant que liens *hypermédia* fournits dans la représentation contenu dans la charge utile de la réponse. Cela permet de répondre en parti à la problématique du versioning en limitant la surface d'attaque de l'API, et en rendant l'accès aux ressources essentiellement dynamique. L'*hypermédia* apporte les propriétés de découvrabilité et de faible couplage.

3.3 Implémentations existantes

En parallèle de cette analyse théorique, nous avons également étudié les différentes solutions adoptées pour implémenter des API RESTful. Nous nous intéressons ici à plusieurs frameworks et bibliothèques ayant alimentés la réflexion autout des choix d'implémentations qui ont été faits pour l'intégration de REST dans OFBiz. Nous présentons donc successivement la spécifications JAX-RS ainsi que les framework Vert.x et Camel.

3.3.1 JAX-RS

JAX-RS est une API java permettant de créer des services Web « RESTful » [12]. Cette spécification est intégrée dans *JavaEE* et possède plusieurs implémentations dont Jersey qui est l'implémentation de référence, RESTeasy, Restlet, et Apache CXF. L'utilisation de Jersey est

```
rest("/customers/")
  .get("/{id}").to("direct:customerDetail")
  .get("/{id}/orders").to("direct:customerOrders")
  .post("/neworder").to("direct:customerNewOrder");
```

FIGURE 3.5 – REST DSL Java

```
<rest path="/customers/">
  <get uri="/{id}">
    <to uri="direct:customerDetail"/>
  </get>
  <get uri="/{id}/orders">
    <to uri="direct:customerOrders"/>
  </get>
  <post uri="/neworder">
    <to uri="direct:customerNewOrder"/>
  </post>
</rest>
```

FIGURE 3.5 – REST DSL XML

impossible du fait de l'incompatibilité de sa license avec celle d'OFBiz. Du fait de son intégration dans la communauté Apache, CXF semble être une implémentation adapté à une utilisation dans OFBiz. L'avantage de cette solution est une intégration et adoption forte dans l'univers JavaEE. Du fait qu'OFBiz est déjà couplé au JDK ce choix ne présente pas un grand risque. Cette spécification utilise un système d'annotation comprenant les annotations `@Path`, `@GET`, `@POST`, `@Produces`, `@Consumes` permettant d'associer les méthodes HTTP à des méthodes Java pour une route donnée.

3.3.2 Vert.x

Vert.x est un framework permettant de créer des API Web Asynchrone. Il s'inspire grandement de ce qui a été fait dans *Node.js* avec l'utilisation intensive de callback dans des requêtes HTTP et les appels vers la base de données. Un point fort de Vert.x est le fait que les traitements asynchrones soientt bien intégrés à l'API. Cela permet d'écrire des traitements de requêtes de manière relativement déclarative grâce à des callbacks tout en ayant de bonnes performances. À la différence de JAX-RS, il n'y a pas de standard pouvant donner une bonne confiance dans la pérénnité de Vert.x d'autant que ce projet créé en 2011 reste relativement jeune.

3.3.3 Apache Camel

Apache camel est un framework implémentant un ensemble de patterns de communications par envoi de messages décrits dans l'ouvrage « Enterprise Integration Patterns » [10]. Une solution intéressante proposé par ce framework et montrée dans la figure 3.5 est la définition d'un langage spécifique REST transposable à la fois en Java et en XML permettant de décrire les routes et les handlers de traitement des types de requêtes associés à ces routes.

Chapitre 4

Réalisations

4.1 Analyse des besoins

Afin de répondre au mieux aux besoins de ce stage, nous avons effectué une analyse des besoins décomposée d'une part dans des besoins fonctionnels pour décrire les fonctionnalités que devra avoir l'API REST, et les besoins non-fonctionnels d'autre part pour décrire des besoins qualitatifs ou des contraintes d'environnement ou de développement. Étant donné le mode de développement agile imposé par le caractère communautaire du projet, cette analyse des besoins n'est pas à voir comme un cahier des charges complet, mais comme une description précise minimale de ce qui est attendu en terme de développement et de manière de travailler.

4.1.1 Besoins fonctionnels

L'objectif de ce stage est de créer une API basée sur le protocole HTTP et suivant le style architectural REST. Actuellement les traitements mis à disposition par OFBiz sont fournis au travers du moteur de services accessible au moyen d'une interface HTTP de type *Remote Procedure Call* (RPC) et du moteur d'entités qui lui n'est pas directement accessible par le réseau. Le besoin premier sous jacent à la création d'une API REST est de fournir une interface orienté ressources aux moteurs de services et d'entités. L'accès aux ressources se fait au moyen d'une *Uniform Resource Identifier* (URI) à laquelle le client HTTP envoit une requête HTTP avec une méthode (GET/POST/PUT...) adapté à la sémantique du traitement que le client HTTP veut effectué. Les requêtes utilisant les méthodes POST/PUT incluent une charge utile prenant la forme d'une représentation de ressource, permettant de créer ou modifier des données sur le serveur. Lors du traitement d'une requête par le serveur, cela a pour effet d'appeler des services OFBiz et/ou de faire des accès en base de données au moyen du moteur d'entités pour manipuler des entités. À partir de ces traitements le serveur envoit ensuite une représentation de la ressource requêtée avec un code d'erreur adapté à la sémantique du traitement effectué. Ceci est représenté de manière simplifié par le diagramme des cas d'utilisation de la figure 4.1.

4.1.2 Besoins non-fonctionnels

La solution proposée devra s'intégrer dans le framework OFBiz. Ainsi il est important de peser tout ajout de dépendance. En effet le fait de dépendre d'un autre framework a l'avantage de ne pas avoir à réimplémenter les services fournit par ce framework mais impose d'avoir une forte confiance quant à la pérénnité de cette dépendance autant en terme de qualité du code, et de la maintenance. Ainsi certains frameworks bien que pouvant correspondre aux critères techniques seront négligés du fait de leur immaturité ou de l'instabilité de leur gouvernance. La solution proposée devra être validée étape après étape par la communauté au moyen de discussions sur

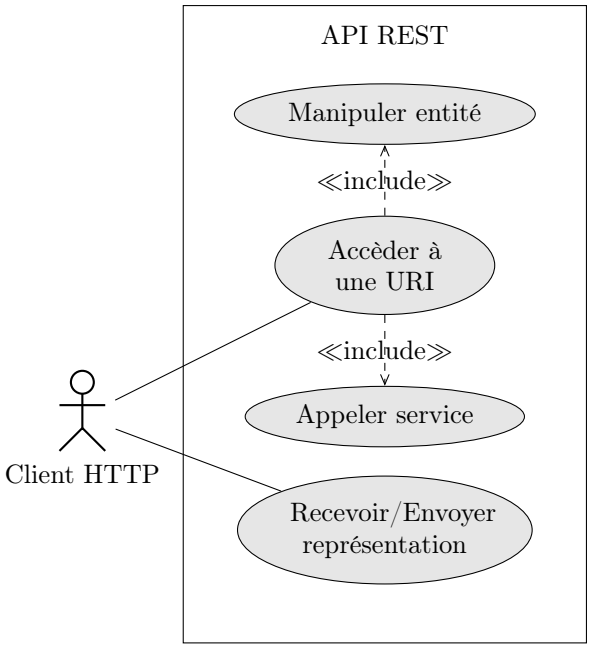

FIGURE 4.1 – Représentation des cas d'utilisation

la liste de diffusion [1] et de revue de codes sur le gestionnaire de ticket JIRA [2]. C'est en cela que le mode de développement de ce stage est agile dans la mesure où les remarques émanant de la communauté peuvent arrivées à tout moment et donc amenées un changement de direction ou de besoins. Le code écrit devra être soumis à la licence Apache 2.0 avec une signature d'un accord de license de contributeur [3] (ICLA) permettant d'apporter des garanties en terme de « propriété intellectuelle ».

4.2 Propositions passées

Cela fait plusieurs années que l'intégration d'un moyen d'accéder aux services et entitées au moyen d'une API HTTP de style REST est en discussion au sein de la communauté des développeurs d'OFBiz. En effet c'est en 2011 qu'Adrian Crum proposa un premier prototype [4]. La première idée était de permettre de faire du *CRUD* sur les entités aux travers des méthodes HTTP. Il a été ensuite question de permettre d'appeler des services et de faire correspondre les *Map* d'entrées et de sorties du service avec les représentations JSON et XML. Cependant la conversion des codes d'erreurs d'appels de services vers des codes d'erreur HTTP semble difficilement automatisable. En effet les types d'errreurs de services sont *Success*, *Failure*, et *Error*. Lorsque l'appel d'un service échoue du fait de la non-authentification de l'appelant le code de retour est *Fail* ce qui implique qu'il ne peut être distinguer d'une erreur d'exécution et que donc il n'y a pas de moyen de faire une correspondre l'échec d'authentification vers le code d'erreur 401.

4.3 Proposition

Actuellement OFBiz gère les requêtes HTTP grâce à une servlet Java [2] définit dans la classe `controlServlet`. Le traitement des requêtes HTTP considère les méthode `GET` et `POST` sans disc-

1. `dev@ofbiz.apache.org`
2. `https://issues.apache.org/jira/projects/OFBIZ`
3. `https://www.apache.org/licenses/icla.pdf`
4. `https://issues.apache.org/jira/browse/OFBIZ-4274`

```
<request-map uri="example" method="get">
   <security https="true" auth="true"/>
   <event type="java" path="ExamplesHandlers" invoke="getExamples"/>
   <response name="success" type="view" value="..."/>
   <response name="error" type="view" value="..."/>
</request-map>

<request-map uri="example2" method="post">
   <security https="true" auth="true"/>
   <event type="java" path="ExamplesHandlers" invoke="postExamples"/>
   <response name="success" type="view" value="..."/>
   <response name="error" type="view" value="..."/>
</request-map>
```

FIGURE 4.2 – Modification du contrôleur

tinction. La première voie explorée a été de définir une servlet alternative à `controlServlet`. Cela avait été fait dans l'idée de définir le support des méthodes HTTP et des templates d'URI au moyen de la spécification JAX-RS qui s'intègre bien avec le mécanisme de servlet. Il était également question dans ce contexte de séparer les routes de l'API et les routes pointant vers les écrans. OFBiz fournit déjà un routeur HTTP avec l'ensemble des routes définis dans `controller.xml` sans distinction de routes HTTP servant à appeler un service et celle correspondant à un écran. De ce fait il a semblé pertinent d'adapter la grammaire XML du contrôleur plutôt que de proposer une implémentation parallèle utilisant un paradigme de définition de routes différents des habitudes des développeurs OFBiz. Étant donné qu'il est hors de propos de déprécier le mécanisme de routage actuel, un telle solution étrangère aurait amenée de la complexité accidentelle du fait des différentes options quand à l'endroit où définir une route HTTP. Ainsi la solution qui a été approfondie a été celle d'adapter la grammaire XML définit par le schéma `site-conf.xsd`. Le choix a été fait de proposer une solution itérative plutôt que d'adopter une approche holistique. En s'inspirant du modèle de maturité REST de Richardson [7], nous avons considéré l'introduction de la notion de ressource et de support des méthodes HTTP sans nous préoccuper de la problématique de l'hypermédia.

4.3.1 Modification du controlleur

La modification du contrôleur a consisté d'une part a ajouté un attribut `method` à l'élément `request-map` acceptant un nom de méthode HTTP. Cet attribut est optionnel et sa valeur par défault est la chaine vide. La sémantique associé à cet attribut est une contrainte sur la résolution des requêtes. Ainsi dans l'exemple de la figure 4.2 la route `/example` n'est accessible qu'au travers de la méthode `GET` et la route `/example2` n'est accessible qu'au travers de la méthode `POST`. Le comportement par défault est de gérer l'ensemble des méthodes HTTP. Cela reste donc compatible avec l'implémentation précédente sans impliquer de changement majeur dans le schéma XSD.

Le deuxième travail réalisé à été de supporter les templates d'URI. En effet lorsque l'on modélise des ressources cela implique généralement de modéliser des collections. On a donc envie de pouvoir identifier de manière générique un élement de cette collection. Les templates d'URI permettent de faire cela. Afin d'avoir une implémentation robuste, nous avons fait le choix de réutiliser la classe `URITemplate` développé par Apache CXF. Cela nous permet d'entamer la transition d'un modèle RPC vers un modèle ressources comme le montre l'exemple de la figure 4.3 qui introduit la resources *geos* en lieu et place de l'ensemble de noms correspondant à des appels de services.

```
<request-map uri="LookupGeo">...</request-map>
<request-map uri="createGeo">...</request-map>
<request-map uri="updateGeo">...</request-map>
<request-map uri="deleteGeo">...</request-map>

<request-map uri="geos" method="get">...</request-map>
<request-map uri="geos" method="post">...</request-map>
<request-map uri="geos/{id}" method="update">...</request-map>
<request-map uri="geos/{id}" method="delete">...</request-map>
```

FIGURE 4.3 – Conversion des identitifiants géographiques

4.3.2 Rendu JSON

En complément de ce travail sur le routage, un travail sur le rendu des ressources au moyen de représentations JSON a été amorçé. Cela a été fait en utilisant l'abstraction du moteur d'écrans. Concrétement nous avons implémenté un visiteur [8] permettant de parcourir le modèle des écrans et widgets pour constuire un objet JSON.

4.4 Difficultés

D'un point de vue extérieur l'ensemble des fonctionnalités développées au cours des 3/4 mois consacrés au développement peut semblé restreint. Cependant quelques explications concernant le contexte permettent de mieux mesurer les enjeux de la tâche. En effet le framework OFBiz est un framework ayant accumulé une dette technique importante et impliquant beaucoup de complexité accidentelle. Par exemple le moteur d'écrans a accumulé des extensions de son modèle sans architecture adapté entrainant une grande difficulté de raisonnement autour de ce modèle.

En ce qui concerne mon travail au niveau du contrôleur, un grand frein dans le développement a été le fait que cela impliquait une modification d'une méthode de plus 800 lignes rempli de mutations, de comparaison à `null`, et avec des chemins d'exécution complexe. Cela a donc impliqué un travail important de ré-usinage faisant usage des nouvelles possibilités offertes par Java 8 que sont l'API des streams et l'ajout des lambda.

Un autre point important est le rapport à la communauté OFBiz. En effet le travail communautaire impose une temporalité particulière, et les soumissions de patches peuvent être plus compliquées que dans d'autres contextes. En effet lorsqu'il faut convaincre la personne en mesure de valider son travail de l'intérêt du code proposé cela peut prendre du temps. Cette lenteur et prudence est justifié par le fait que tout modification de code impact tous les utilisateurs, mais aussi les responsables du projet qui devront inévitablement maintenir le code ajouté.

Chapitre 5

Conclusion

Ce stage a rempli mon objectif de découvrir le développement de logiciel libre dans un cadre professionnel. Étant donné les difficultés rencontrées qui ont été explicitées dans la section précédente, il n'a pas été possible pour moi d'aller jusqu'au bout de l'objectif de développer une API REST pour OFBiz. Cependant la réflexion développée et partagée avec la communauté OFBiz autour de la manière d'intégrer une telle API et les premiers développements réalisés sont en mesure de fournir une ligne directrice pour une reprise du projet. En cela je considère ce stage comme un succès. À un niveau personnel j'ai apprécié le fait que mes compétences et connaissances techniques aient pu être mises en perspective de problématiques fonctionnelles ; le fonctionnel étant la raison d'être d'un framework comme OFBiz.

Bibliographie

[1] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer. Simple object access protocol (soap) 1.1, 2000.

[2] Shing Wai Chan and Ed Burns. Java servlet specification. *Oracle Corporation, July 2017*, pages 1–246, 2017.

[3] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol–http/1.1. Technical report, 1999.

[4] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*, volume 7. University of California, Irvine Doctoral dissertation, 2000.

[5] Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2) :115–150, 2002.

[6] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] Martin Fowler. Richardson maturity model : steps toward the glory of rest. *Online at http ://martinfowler. com/articles/richardsonMaturityModel. html*, pages 24–65, 2010.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns : Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*, pages 406–431. Springer, 1993.

[9] Jesse James Garrett et al. Ajax : A new approach to web applications. 2005.

[10] Gregor Hohpe, Bobby Woolf, et al. Enterprise integration patterns. 2003.

[11] Larry Masinter, Tim Berners-Lee, and Roy T Fielding. Uniform resource identifier (uri) : Generic syntax. 2005.

[12] Santiago Pericas-Geertsen and Marek Potociar. Jax-rs : Java™ api for restful web services. *Oracle Corporation, May 2013*, pages 1–84, 2013.

[13] Dewayne E Perry and Alexander L Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4) :40–52, 1992.