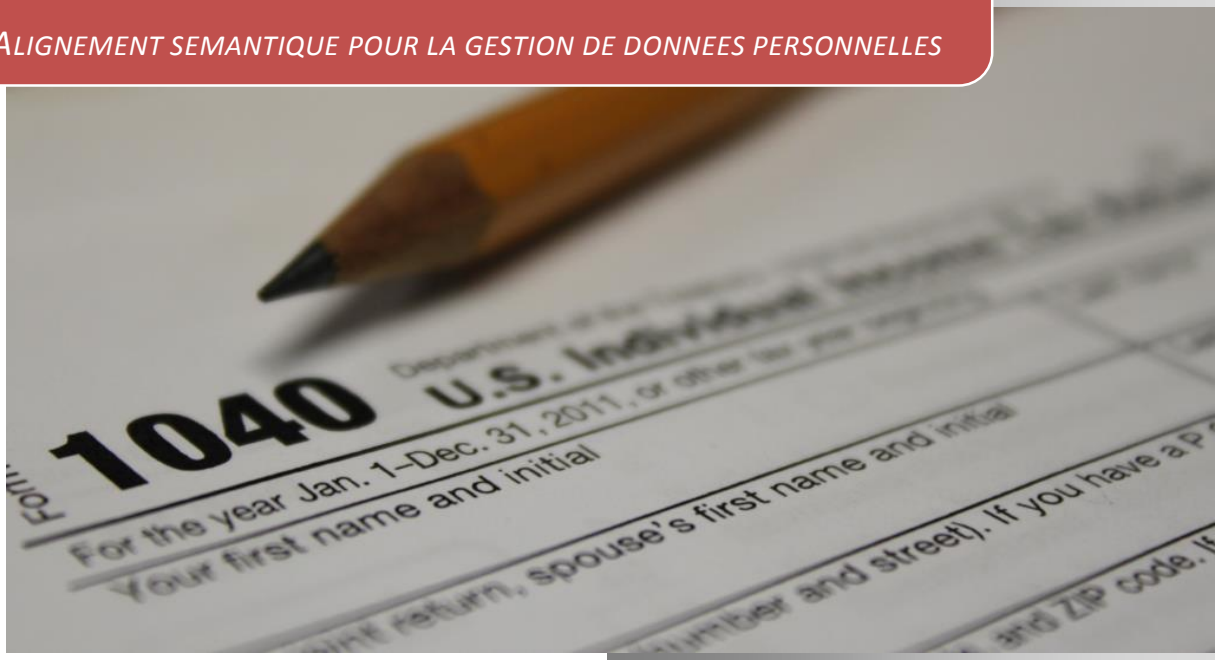
Lyon 1

2013

RAPPORT DE STAGE

ALIGNEMENT SEMANTIQUE POUR LA GESTION DE DONNEES PERSONNELLES



Emmanuel GAUDE

M2 Pro Technologies de l'Information
Université Claude Bernard Lyon I

Maitre de stage : Philippe LAMARRE
Tuteur : Emmanuel COQUERY

Laboratoire : L.I.R.I.S. – Équipe Bases de Données
Bâtiment Blaise Pascal
7 Avenue Jean Capelle
69621 Villeurbanne Cedex

REMERCIEMENTS

Je souhaite remercier toutes les personnes ayant contribué au bon déroulement de mon stage, et tout particulièrement :

M. Philippe LAMARRE, mon maître de stage, ainsi que **MM. Elöd EGYED-ZSIGMOND** et **Fabien DUCHATEAU**, pour l'encadrement et les conseils qu'ils m'ont donnés tout au long de ces cinq mois.

M. Olivier CAVADENTI et Mme. Souhir JOUILI, mes deux collègues de bureau effectuant aussi leur stage de Master, pour leur amabilité.

Je tiens également à remercier **laboratoire du L.I.R.I.S.** pour m'avoir accueilli durant ces cinq mois.

Enfin un grand « merci » **aux équipes pédagogique de l'Université Lyon I** pour la formation prodiguée tout au long de ces années d'études et leur bienveillance et qui m'ont apportées tant de connaissances techniques et méthodologiques indispensables.

SOMMAIRE

INTRODUCTION

Le stage de Master est l'occasion pour nous, futurs diplômés, de mettre en pratique nos connaissances acquises jusqu'alors pour commencer à nous insérer dans le monde du travail.

Pour ma part, indécis entre l'univers de l'entreprise et celui de la recherche, j'ai choisi de faire un stage de recherche à l'issue de mon master pro afin de m'ouvrir un maximum d'opportunités.

Et curieux du Web sémantique, décrit par Tim BERNERS-LEE comme composante du Web 3.0, je souhaitais avoir l'occasion de m'en approcher un peu plus dans le cadre de mon stage.

Le thème de mon stage porte en effet sur l'alignement sémantique de données personnelles, dans le cadre du projet Adnosco.

Ce fût pour moi l'occasion d'une part d'effectuer un travail technique sur l'application servant à la démonstration de la faisabilité de ce projet, et d'autre part de voir en quoi consiste un travail de recherche.

Ce rapport rend compte de mon activité durant mon stage de la période du 18 février au 19 juillet 2013.

Il sera tout d'abord introduit en première partie par une présentation du laboratoire et de mon environnement de travail, ainsi que de la mission et du sujet du stage. En seconde partie seront présentés les aspects théoriques et pratiques de mon travail. Enfin, les perspectives sur l'avenir du travail réalisé et le retour sur l'expérience acquise seront traités lors de la dernière partie.

I ENVIRONNEMENT ET MISSION

I.A ENVIRONNEMENT PROFESSIONNEL

I.A.1 PRESENTATION DU LABORATOIRE

Le **L.I.R.I.S.**, pour « **L**aboratoire d'**I**nfo**R**matique en **I**mage et **S**ystèmes d'information », est un laboratoire de recherche en informatique de la région lyonnaise affilié au C.N.R.S.

Le laboratoire regroupe environ trois cents personnes, dont près de cent-dix chercheurs et enseignants chercheurs.

Sous la tutelle de l'INSA de Lyon, l'Université Claude Bernard Lyon 1, l'École centrale de Lyon et l'Université Lumière Lyon 2, ses deux principaux domaines de recherche sont *Image* et *Données, Connaissances, Services*. Ces départements sont composés chacun de cinq équipes reparties entre plusieurs pôles de compétence.

La mission de mon stage s'est réalisée dans l'équipe *Bases de Données* du département *Données, Connaissances, Services*. Son responsable est **M. Jean-Marc PETIT** et son responsable adjoint est **M. Philippe LAMARRE**.

Mon stage a été encadré par **MM. Philippe LAMARRE**, **Elöd EGYED-ZSIGMOND** et **Fabien DUCHATEAU.** Celui-ci s'est déroulé dans les locaux du bâtiment Blaise PASCAL du campus de la Doua.

I.B ENVIRONNEMENT DE TRAVAIL ET DE DEVELOPPEMENT

I.B.1 DESCRIPTION GENERALE DE L'ENVIRONNEMENT INFORMATIQUE

À mon arrivée une unité centrale équipée d'un Intel Xeon et de 8 Go de mémoire vive, et un écran m'ont été généreusement prêtés pour la réalisation de mes tâches. J'ai pu y installer librement le système de mon choix ainsi que les outils m'étant nécessaires.

J'ai choisi d'installer sur ce poste la distribution **Arch Linux** dont je suis coutumier. Le principal intérêt de cette distribution est de disposer d'un dépôt conséquent d'applicatifs récents, et dans leurs versions originales contrairement à **Debian**. Dans l'éventualité de l'absence d'un programme ou d'une bibliothèque dans le dépôt officiel, il est alors possible d'utiliser le dépôt utilisateur *AUR* encore plus complet ou si nécessaire de créer et soumettre son propre paquet.

Par la suite, pour pouvoir plus facilement travailler sans avoir à synchroniser systématiquement mon travail, j'ai choisi de travailler directement sur mon ordinateur portable sous OS X : un *Macbook Pro Late 2011* équipé d'un SSD et 16 Go de mémoire vive.

En raison d'incompatibilités de certaines commandes utilisées dans mes scripts, il m'a été nécessaire de récupérer via Mac Ports leurs versions GNU en complément des versions BSD fournies dans le système d'Apple.

I.B.2 ORGANISATION DU TRAVAIL ET COMMUNICATION

Comme établi par la convention de stage, la durée du travail est de trente cinq heures par semaines. La communication avec mes encadrants passait à la fois verbalement en raison de la proximité des bureaux de **MM. LAMARRE** et **EGYED-ZSIGMOND**, et par courrier électronique autrement.

Les semaines étaient ponctuées par une à deux réunions avec mes encadrants afin de connaître mon avancement, discuter des problèmes rencontrés et me confier les objectifs à accomplir pour les prochaines réunions.

Le travail a été divisé en deux parties : une partie théorique consistant à chercher une méthode pour l'alignement sémantique d'un formulaire à une ontologie, et une partie pratique concernant l'outil de démonstration.

I.B.3 ENVIRONNEMENT DE DEVELOPPEMENT

La majeure partie du projet s'articulant autour du langage **Java**, l'IDE **Eclipse** reste l'outil de choix privilégié pour développer.

Les données sont stockées dans une base de données **MySQL**, gérée avec **MySQLWorkbench**.

Différentes bibliothèques ont été utilisées au sein du projet et sont gérés par le biais de **Maven** :

- **Hibernate** - Mapping JPA entre classes Java et tables SQL et requêtes JPQL
- **MySQL-Connector/J** – Pilote de connexion de l'application à la base de données MySQL
- **Jersey** – Implémentation de JAX-RS pour la création d'un service Web REST
- **JAXB** – Mapping entre classes Java et représentations XML
- **Jackson-JAXRS** – Permet le support en lecture/écriture de sérialisation JSON sur un service JAX-RS. Utilisé à la place de MOXy qui était proposée par défaut.
- **Grizzly** – Permet de déployer l'application sous forme d'un serveur Web autonome.
- **JUnit** – Pour réaliser les tests unitaires attestant du bon fonctionnement des API en cours de développement.
- **SecondString** pour les mesures terminologiques entre chaînes de caractères.

La partie client du projet se base sur le travail fait par l'extension **Lazarus** pour *Firefox*, *Chrome* et *Safari*. **SQLite Database Browser** a été utilisée pour étudier le contenu de la base de données associée à cette extension.

L'extension *Firefox* **RESTClient** a permis de vérifier le bon fonctionnement du service Web avant d'en implémenter les appels dans le client. Ce dernier a en revanche été développé en effectuant les tests sur le navigateur **Chrome** seulement, point sur lequel je reviendrai.

Pour le client, j'ai préféré faire appel à simple éditeur de texte comme **Sublime Text**.

Pour la partie mesure/évaluation, j'ai réalisé des scripts à partir de **GNU Bash 4** et **GNU awk,** ainsi que généré des graphiques avec **Gnuplot**.

Enfin, j'ai utilisé **LibreOffice Writer** pour le travail de rédaction des différents comptes rendus, ainsi que **Dia** pour mes figures.

I.C LE PROJET ADNOSCO

I.C.1 PRESENTATION DU PROJET

Le stage s'inscrit dans le cadre du projet **Adnosco** qui vise à la conception d'un outil de gestion des données personnelles des utilisateurs dans le cadre des formulaires Web.

Actuellement les internautes ne peuvent pas accéder nécessairement aux informations soumises à des tiers, et lorsqu'elles le sont ne sont pas directement exploitables.

Adnosco a pour objectif de permettre aux utilisateurs, que ce soit dans un usage personnel ou professionnel, de gérer les données qu'ils soumettent à différents services sur le Web.

Les outils de *VRM* (*Vendor Relationship Management*), par opposition aux outils de *CRM* (*Customers Relationship Management*) permettent la gestion des relations du côté des clients. C'est donc sous la forme d'un outil de *VRM* qu'Adnosco doit permettre à l'utilisateur de savoir ce qu'il a soumis et à quels services, le tout dans un format qu'il maîtrise.

Ainsi, l'utilisateur peut être en mesure de réaliser des recherches sur les informations transmises telles que leur nature, leurs destinataires, leur date d'envoi, leur raison, etc. Ce qui peut permettre de mettre en exergue ce qui est sensible et ce qui ne l'est pas.

La connaissance des informations déjà soumises permet aussi à l'utilisateur de s'aider dans le cadre du renseignement de nouveaux formulaires en puisant dans les saisies déjà connues de ce même formulaire, d'autres formulaires ou d'une activité de remplissages multiples en cours *(e.g. réservation d'hôtel, puis d'un moyen de transport)*.

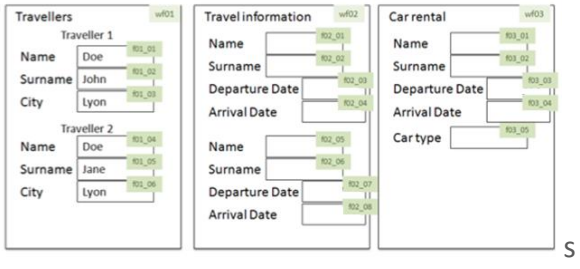


FIGURE 1 - UNE ACTIVITE DE RENSEIGNEMENT DE FORMULAIRES

I.C.2 AUTO-COMPLETION ET PREREMPLISSAGE

Une des finalités du projet **Adnosco** dans la gestion par l'utilisateur de ses données est de lui faciliter la soumission de formulaires, que ce soit par l'auto-complétion d'un champ ou par le préremplissage complet.

L'**auto-complétion d'un champ de formulaire** consiste à proposer automatiquement ou semi-automatiquement une liste de valeurs pouvant remplir ce champ en fonction même du contenu en train d'y être saisi par l'utilisateur.

Le **préremplissage d'un formulaire** consiste à remplir l'intégralité des champs d'un formulaire automatiquement ou semi-automatiquement. Le préremplissage se faisant à partir d'anciennes instances du formulaire soumises par l'utilisateur.

Dans le cadre d'Adnosco, ces deux actions se réalisent selon trois contextes par ordre croissant de priorité : **syntaxique**, **sémantique** et **activité**.

I.C.3 RECHERCHE SYNTAXIQUE, SEMANTIQUE ET ACTIVITES

Lorsque l'utilisateur commence une saisie et qu'il souhaite la compléter, une liste de suggestions lui est présentée dans l'ordre suivant :

1. Ceux se référant à une **activité** de renseignement de formulaires en cours ;
2. Ceux obéissant à une certaine **sémantique** relative au formulaire courant et les données déjà saisies, c'est-à-dire ayant une signification proche en se basant sur des *ontologies* connues ;
3. Ceux établissant une correspondance **syntaxique** avec la saisie courante, c'est-à-dire.

Dans le contexte syntaxique, lui est proposé dans l'ordre :

1. Un préremplissage du formulaire si au moins une instance de ce même formulaire a été complétée ;
2. Une complétion de sa saisie pour le champ actuel à partir d'un autre champ de même type de tout formulaire.

Évidemment, un dernier tri est réalisé dans ces différents contextes selon une méthode de scores interne à Adnosco sur laquelle nous reviendrons plus tard.

I.C.4 ONTOLOGIES : DEFINITION

Dans le domaine informatique, une ontologie est un moyen de représenter des connaissances de façon structurée d ans un graphe mettant en relation différents concepts.

En s'inspirant de la définition d'une ontologie de Jérôme EUZENAT[1], mais en détachant la notion d'une Propriété de celle d'une Relation, celles-ci peuvent être conceptualisées par le diagramme de classe suivant :

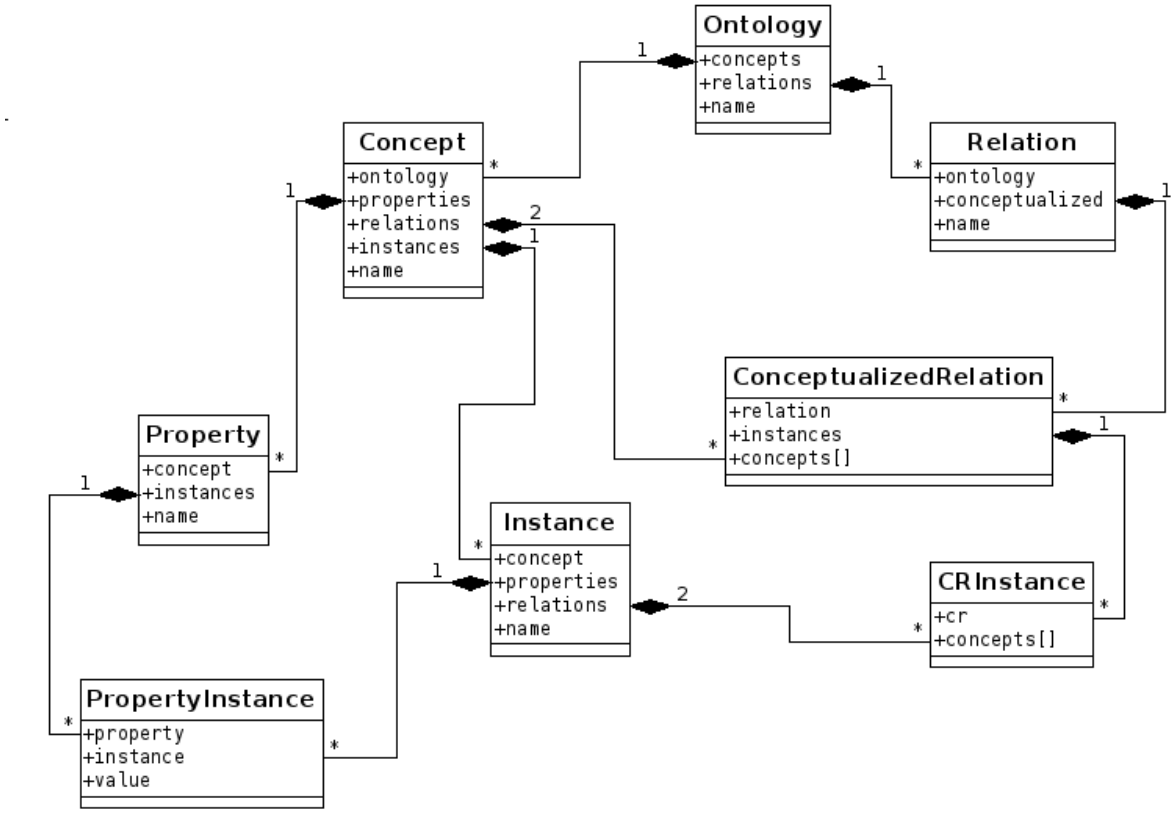


FIGURE 2 - DIAGRAMME DE CLASSES D'UNE ONTOLOGIE

- **Ontologie**
  Une ontologie est un ensemble de concepts et de relations (abstraites). Exemple d'ontologie : « Entreprise ».
- **Concept**
  Un concept appartient à une ontologie donnée. Celui-ci possède diverses propriétés, peut-être mis en relation avec d'autres concepts et peut-être instancié. Exemple de concept : « Employé ».
- **Propriété**
  Une propriété appartient à concept donné. Celle-ci peut-être instanciée au sein d'une instance de même concept. Exemple de propriété : « Nom d'un Employé »

- **Relation (abstraite)**
  Une relation appartient à une ontologie donnée. Celle-ci est abstraite et peut correspondre à plusieurs conceptualisations. Exemple de relation : « Assiste ».
- **Relation Conceptualisé**
  Une relation conceptualisée est l'application d'une relation entre deux concepts. Plusieurs instances de cette relation conceptualisation peuvent être réalisée. Exemple de relation conceptualisée : « Employé 1 Assiste Employé 2 » ou « Employé Assiste Service »

- **Instance (de Concept)**
  Une instance de concept est la réalisation de ce concept. Elle possède elle même plusieurs instances de propriétés correspondant aux propriétés du concept. Exemple d'instance : « Employé Jean MARTIN»
- **Instance de Propriété**
  Une instance de propriété est la réalisation d'une propriété d'un concept dans le cadre d'une instance de ce concept. Exemple d'instance de la propriété « Nom de l'Employé Jean MARTIN » : « MARTIN »
- **Instance de Relation Conceptualisée**
  Une instance de relation conceptualisée est la réalisation d'une relation conceptuelle entre deux instances de concepts correspondants. Exemple de relation conceptualisée : « Employé Jean MARTIN Assiste Employé Paul DURAND ».

Dans le cadre actuel du projet **Adnosco**, les différentes notions de Relations et les différentes notions d'Instances sont pour le moment inutilisées.

I.C.5 QUALIFICATION SEMANTIQUE

Le principe d'une ontologie désormais défini, nous pouvons nous intéresser au principe d'un **qualification sémantique** telle que définie dans le projet **Adnosco**[6].

Jusqu'à présent, les outils sémantiques travaillant sur les formulaires Web considèrent ces derniers comme étant des ontologies. C'est-à-dire, que chaque formulaire est une ontologie, ses champs les propriétés et les soumissions de données par un utilisateur des instances.

Adnosco propose une approche différente et originale, où les ontologies et leurs concepts sont distincts de la notion de formulaires Web. Le lien étant fait entre les deux par la notion de **concepts matérialisés**.

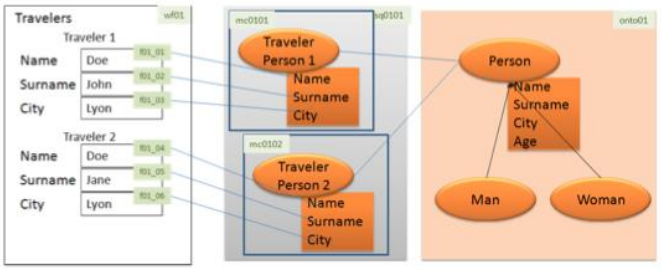


FIGURE 3 - QUALIFICATION SEMANTIQUE D'UNE INSTANCE DE FORMULAIRE

Une **qualification sémantique** est un ensemble de **concepts matérialisés** pour un **formulaire** donné. Celle-ci appartient ou non à une **activité** établissant des équivalences entre concepts matérialisés de **formulaires** différents.

Les **concepts matérialisés** sont associés à un **concept** d'ontologie et une **qualification sémantique** donnés. Ils sont constitués de **correspondances** entre un à plusieurs **champs de formulaires** et une à plusieurs propriétés du **concept** associé.

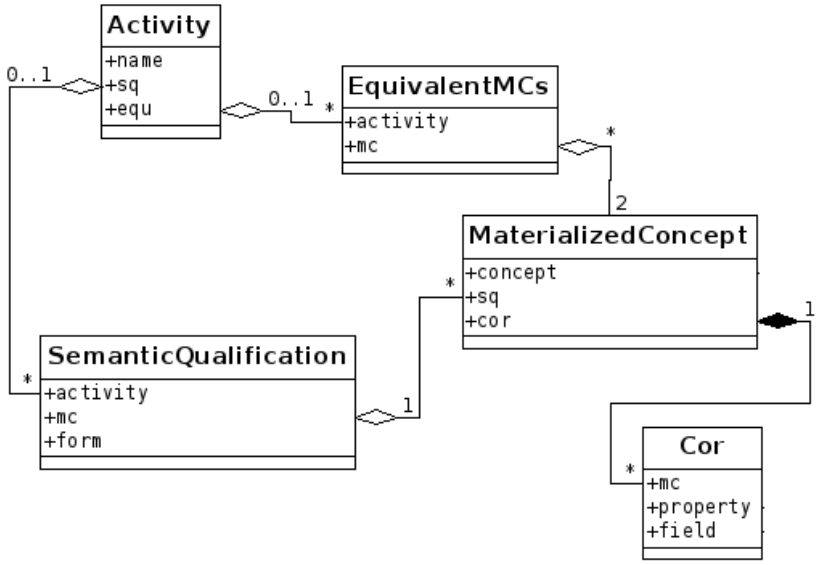


FIGURE 4 – DIAGRAMME DE CLASSES DU CŒUR D'ADNOSCO SUR LES QUALIFICATIONS SEMANTIQUES

I.D OBJECTIFS DE LA MISSION DU STAGE

L'objectif premier de ce stage est d'étudier les modèles et méthodes d'alignements de champs d'un formulaire selon les concepts d'une ontologie et de ses attributs.

Pour y parvenir, la démarche se fait en deux temps : tout d'abord la recherche d'une méthode d'alignement manuelle des champs par rapport à une ontologie afin de comprendre les notions liées aux ontologies et à l'alignement sémantique, et par la suite une approche semi-automatique tirant partie des informations déjà saisies et déjà qualifiées sémantiquement.

II TRAVAIL REALISE

Le travail réalisé durant ce stage comporte deux aspects qui ont été travaillés simultanément. Tout d'abord l'aspect recherche qui s'intéresse à la qualification sémantique de nouveaux formulaires, et à l'aspect technique qui se veut proposer un modèle expérimental destiné à l'utilisateur.

II.A PARTIE RECHERCHE

II.A.1 PROBLEMATIQUE

« Comment qualifier sémantiquement de nouveaux formulaires jusqu'alors inconnus ? »

II.A.2 DEMARCHE

La problématique scientifique de l'alignement posée, la démarche est la suivante :

1. Recherche d'une méthode d'alignement entre un formulaire et une ontologie
2. La préparation d'un jeu d'essai pour évaluer cette méthode
3. Évaluation de cette méthode à partir de mesures dîtes « expertes »

II.A.3 SCENARIO D'ALIGNEMENT ENTRE UN FORMULAIRE ET UNE ONTOLOGIE

Au moment de la découverte d'un nouveau formulaire, comment le relier automatiquement à une ontologie connue afin de lui apporter la précomplétion et le préremplissage ?

II.A.3.a Étape 1 – Recherche de formulaires similaires

Avant de chercher à comparer le nouveau formulaire à une ontologie, on recherche une sélection des formulaires les plus proches déjà connus du système.

Pour effectuer cette sélection de formulaires les plus proches il est nécessaire d'établir une mesure de proximité entre deux formulaires alignés.

Par exemple : $\Delta = \frac{2 \times m}{n_1 + n_2}$

Avec n_i *nombre de champs du formulaire i, et m nombre de champs alignés entre les deux formulaires.*

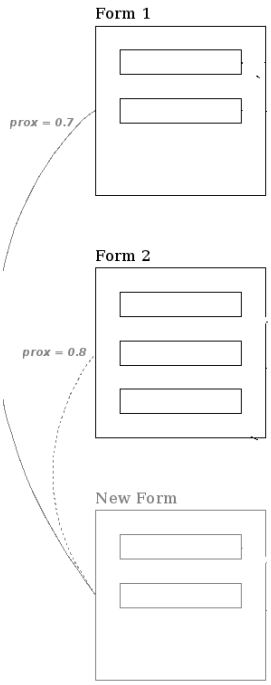


FIGURE 5 - CALCUL D'UN SCORE DE PROXIMITE ENTRE FORMULAIRES

Mais sur quels critères aligner deux formulaires ensembles ?

Un formulaire Web apporte beaucoup d'informations comme :

- Le nombre de champs qu'il comporte,
- Les noms de ses champs, pour l'utilisateur avec le *label* et en interne avec l'attribut *name*,
- Les types de ses champs : *texte court, texte long, liste déroulante, email, date, etc.*,
- Les groupes de champs : via *fieldset* ou même de simples blocs *div*,
- Des valeurs par défaut et/ou des sauvegardes d'anciennes valeurs soumises,
- L'*url* de sa page, celle de l'*action* et la *méthode* (*GET* ou *POST*)...

Et plus éventuellement tous ceux propres aux éléments HTML :

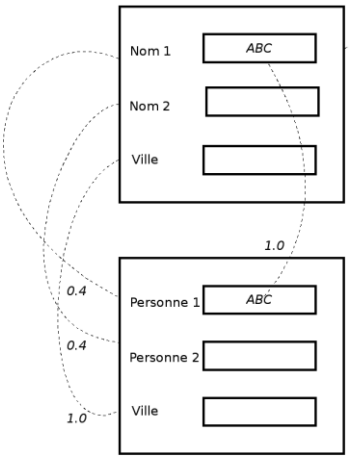
- Des identifiants et classes *(attributs id et class)*,
- Une langue *(attribut lang)*,
- Etc.

À propos des types de champs, on relève la liste suivante en HTML5 :

TEXT,	IMAGE,	DATE,	COLOR,
PASSWORD,	HIDDEN,	TIME,	METER,
FILE,	TEXTAREA,	DATETIME,	PROGRESS*,
RADIO,	SELECT,	DATETIME LOCAL,	OUTPUT*,
CHECKBOX,	TEL,	MONTH,	KEYGEN*
SUBMIT,	URL,	WEEK,	
RESET,	EMAIL,	NUMBER,	
BUTTON,	SEARCH,	RANGE,	

Ce qui amène, pour les nouveaux types de champs introduits, de nouvelles informations sémantiques pour les formulaires modernes. Certains, comme le type *HIDDEN* peuvent être ignorés dans l'évaluation.

L'***Annexe B*** présente un diagramme de classes détaillant davantage l'analyse des informations portées par un formulaire.



Le premier jet de la méthode d'alignement entre formulaire s'intéresse aux *libellés* des champs. Seules les paires d'alignement de champs ayant un score au-delà d'un seuil prédéterminé pour la mesure de similarité de leur libellé sont retenues.

* Sachant que les trois derniers ne sont pas exploitables en saisie.

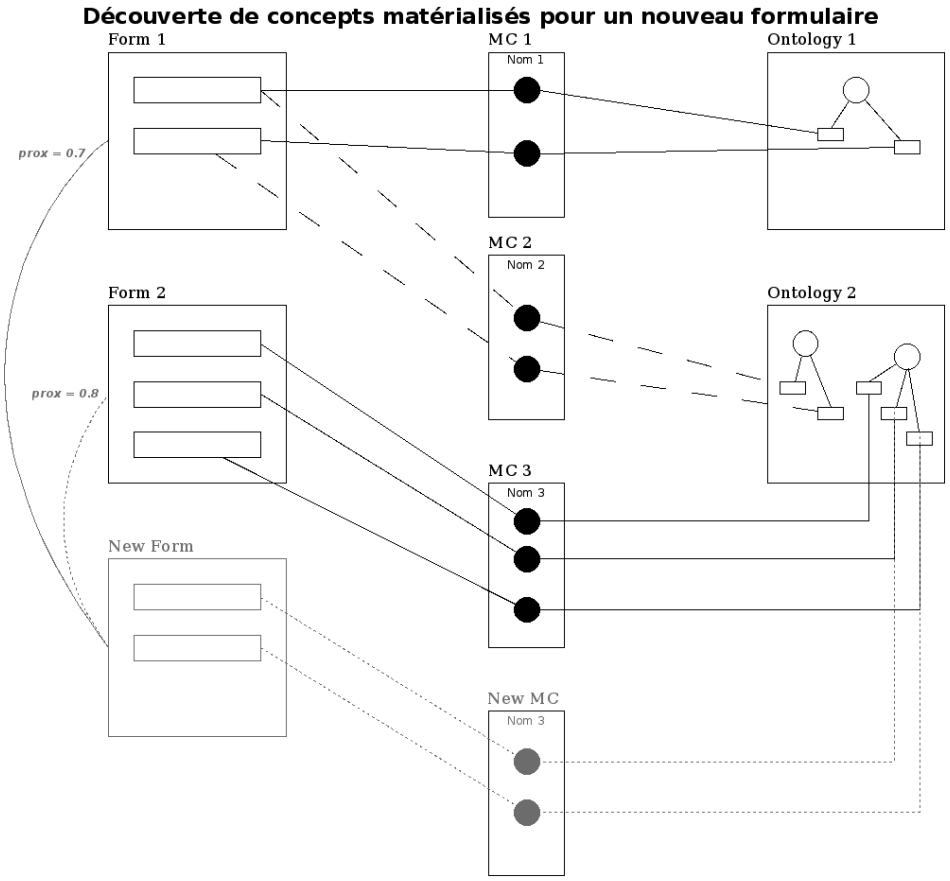
À noter que les libellés sont généralement rédigés dans la langue de la personne utilisant le formulaire, tandis que *name* peut :

- Etre en anglais par habitude ou par complexité *(E.g. sur page dans un alphabet non latin)* ;
- Dans la langue de l'utilisateur ;
- Etre codifiée en raison de mesures techniques *(E.g. une cellule de tableau, ou deux mots regroupés par le « _ »)*.

II.A.3.b Etape 2 – Réalisation du lien avec l'ontologie

Une fois un formulaire connu suffisamment proche du nouveau formulaire, on peut par transitivité des *concepts matérialisés* déjà associés remonter à l'ontologie la plus adaptée.

Ainsi de nouveaux concepts matérialisés sont créés entre le nouveau formulaire et l'ontologie, en prenant pour référence les champs alignés du formulaire connu.



Évidemment cette approche nécessite qu'un certain nombre de formulaires, de qualifications sémantiques et d'ontologies soient connus. Les qualifications manquantes ou erronées entre le nouveau formulaire et l'ontologie doivent aussi pouvoir être éditées manuellement.

II.A.4 MESURES ET METHODES DE COMPARAISON ENTRE ONTOLOGIES

Tout en présentant le projet Adnosco, nous avons défini en première partie ce qu'est une ontologie mais nous n'avons pas défini comment les comparer. C'est ce que nous allons faire ici à partir de l'état de l'art établit par Jérôme EUZENAT[1].

II.A.4.a Mesures de similarité et de dissimilarité

Pour comparer deux ontologies entre elles, il est nécessaire d'établir les mesures à employer. Nous pouvons les distinguer en deux types : les mesures de **similarité**, et les mesures de **dissimilarité**.

Soit O l'ensemble des ontologies, σ une mesure de similarité et δ une mesure de dissimilarité toutes deux prenant leurs valeurs dans $\mathbb{R}_+$.

Ces mesures doivent obéir aux propriétés suivantes :

Similarité $\sigma : O \times O \rightarrow \mathbb{R}_+$	
Positivité	$\forall x, y \in O, \sigma(x,y) \geq 0$
Maximalité	$\forall x, y, z \in O, \sigma(x,x) \geq \sigma(y,z)$
Symétrie	$\forall x, y \in O, \sigma(x,y) = \sigma(y,x)$

Dissimilarité $\delta : O \times O \rightarrow \mathbb{R}_+$	
Positivité	$\forall x, y \in O, \delta(x,y) \geq 0$
Minimalité	$\forall x \in O, \delta(x,x) = 0$
Symétrie	$\forall x, y \in O, \delta(x,y) = \delta(y,x)$

À noter que dans certaines littératures il est fait mention de mesures de « similarité non symétriques ».

Très souvent les mesures sont normalisées, et spécialement pour comparer entre différent types d'entités. Les versions normalisées ($\bar{\sigma}$ et $\bar{\delta}$) prennent leurs valeurs dans l'intervalle [0, 1].

Ainsi on obtient la propriété : $\bar{\delta} = 1 - \bar{\sigma}$

Il existe aussi des dissimilarités plus restrictives telles que la notion de distance et de distance ultramétrique. Toutes deux ajoutent des propriétés :

Distance $\delta : O \times O \rightarrow \mathbb{R}_+$
Définitude
$\forall x, y \in O,\ d(x,y) = 0\ iff\ x = y$
Inégalité triangulaire
$\forall x, y, z \in O, \delta(x,y) + \delta(y,z) \geq \delta(x,z)$

Distance ultramétrique $d : O \times O \rightarrow \mathbb{R}_+$
Inégalité ultramétrique
$\forall x, y, z \in O,\ d(x,y) \leq \max(d(x,z), d(y,z))$

II.A.4.b Catégories de méthodes de comparaison

Il y a plusieurs catégories de méthodes connues pour comparer deux ontologies. Celles-ci se regroupent principalement entre méthodes de comparaison dîtes :

- **locales** s'attardant uniquement sur les entités composant les ontologies ;
- **globales** s'intéressant aux ontologies dans leur ensemble.

Plus en détail, ces catégories regroupent des méthodes :

Locales

- **Terminologiques**
  - Basées sur les chaînes
    E.g. *Egalité, Distance d'édition, Distance de Jaro...*
  - Basées sur le langage
    E.g. *Morphologiques, syntaxiques, morphosyntaxiques, multilingues*
- **Structurelles**
  - Internes
    Reviennent à comparer les propriétés (types, instances, domaine)
  - Externes
    Voisinage dans la hiérarchie de l'ontologie
- **Extensionnelles**
  - Instances partagées entre classes
  - Instances non partagées

- **Sémantiques**
  E.g. Techniques de satisfaisabilité (SAT) propositionnelle, de satisafaisabilité modale et basées sur logique de description

Globales

- **Similarité composée (de similarités locales)**
- **Calcul global**
- **Apprentissage**

À noter que les méthodes purement sémantiques marchent rarement bien seules, et nécessitent une phrase de prétraitement fournissant des « ancres ». Sémantiquement exactes, ces méthodes donnent une similarité de 1 pour les objets équivalents.

On y ajoute les méthodes dîtes de :

- **saisie utilisateur**, impliquant l'utilisateur dans la comparaison ;
- **composition de méthodes**, mêlant entre elles plusieurs des autres méthodes ;
- **extraction**, filtrant les résultats de comparaison selon un seuil ou par optimisation de critères globaux.

II.A.4.c Deux mesures terminologiques : distances d'édition et de Jaro

La **distance d'édition**, appelée aussi **distance de Levenshtein**, est une mesure connue indiquant le nombre minimal de caractères à supprimer, insérer ou remplacer pour passer d'une chaîne *s1* à une chaîne *s2*.

La **distance de Jaro**, notée d_j, se calcule selon la formule suivante :

$$d_j = \frac{1}{3}\left(\frac{m}{|s_1|} + \frac{m}{|s_2|} + \frac{m-t}{m}\right)$$

Où :

- ***m*** est le **nombre de caractères correspondants**, c'est-à-dire tels que :

$$éloignement^2(\, s_1[i],\, s_2[j]) \leq \left\lfloor \frac{max(|s_1|,|s_2|)}{2} \right\rfloor - 1$$

avec $éloignement(\, s_1[i],\, s_2[j]) = |i-j|$
et $s_1[i] = s_2[j]$

- **t** est le **nombre de transpositions**, i.e. :

$$N = \max(|s_1|,|s_2|)\,,\; t = \frac{\sum_{i=0}^{N} diverge(s1, s2, i)}{2}$$

avec $diverge(s_1, s_2, i) = \begin{cases} 1\ si\ s_1[i] \neq s_2[i] \\ 0\ sinon \end{cases}$

- $|s_i|$ est la **longueur de la chaîne i**

Il existe plusieurs mesures dérivées de cette mesure de distance, notamment Jaro-Winkler, très adaptée à des chaînes courtes.

Dans le cadre d'une recherche de similarité syntaxique entre deux formulaires, nous avons choisi dans nos évaluations au niveau local la **distance de Jaro** avec un seuil de 0,6 qui semble donner des résultats intéressants sur lesquels nous reviendrons plus tard. Celle-ci est combinée par le biais d'une moyenne des scores des champs alignés.

La première ébauche de mesure de similarité globale entre deux ontologies est portée sur les noms des champs, et compose la mesure globale par le biais d'une moyenne.

[2] L'éloignement de deux caractères est la différence entre leurs positions dans leurs chaînes respectives

II.A.4.d Mesure globale et alignement 1-1 optimisé

Nous savons désormais comment mesurer la (dis)similarité entre deux ontologies d'un point de vue local.

Pour la mesure globale, il convient de déterminer quel type d'alignement est souhaité entre les deux ontologies :

- Un **alignement N-N,** où à chaque concept d'une ontologie peut correspondre plusieurs concepts de l'autre ;
- Un **alignement N-1**, où l'alignement est une surjection d'une ontologie à l'autre ;
- Un **alignement 1-1**, où l'alignement est une bijection des deux ontologies.

Calculer la mesure de similarité globale normalisée (donc sur [0,1]) pour les deux premiers types d'alignement sera plus compliqué que pour l'alignement 1-1.

Néanmoins, réaliser cet alignement 1-1 de façon optimisée n'est pas évident à première vue.

Une **première approche** serait de considérer un **alignement N-1** dont on ne **conserverait** que les **meilleures paires** des membres de gauche déjà référencés. Cette approche a néanmoins l'inconvénient d'éliminer purement et simplement certains concepts, privant potentiellement d'une nouvelle paire associant un concept droit à un concept gauche non déjà référencé.

Une **seconde approche** serait de considérer le problème d'un point de vue **global** en **maximisant la moyenne** des paires retenues :

- **L'algorithme naïf** calculerait l'ensemble des paires et mesures de similarité associées pour ne conserver que le meilleur résultat global. La quantité de concepts en jeu pouvant être importante, et la complexité étant de l'ordre $O(p^n)$, l'exécution de cet algorithme est bien trop lente.
- **L'algorithme Hongrois**, connu aussi sous le nom d'algorithme de **Kuhn-Munkres**[8], est un algorithme d'optimisation combinatoire en temps polynomial (*$O(n^4)$, voir $O(n^3)$, contre $O(p^n)$*).
  En l'adaptant pour chercher des scores maximaux et non minimaux, celui-ci répond bien au problème de l'alignement 1-1 :
  - On affecte les scores opposés aux alignements obtenus
  - Pour les alignements non retenus, on choisit un score maximal (inatteignable)

  Le détail de l'algorithme est présenté en *Annexe A*.

II.A.5 MISE EN PLACE DU JEU D'ESSAI

Ayant établi une démarche pour intégrer une qualification sémantique aux nouveaux formulaires, il est nécessaire de l'évaluer, et d'adapter la méthode d'alignement si nécessaire.

Pour cela un jeu d'essai est nécessaire. Nous sommes partis alors du jeu d'essai d'Avigdor GAL réalisé en 2008 pour son projet OntoBuilder[4]. Celui-ci est constitué de près de deux cents pages d'url différentes, composées d'un à plusieurs formulaires, le tout dans vingt domaines d'activités différents (hôtels, pari en ligne, réservation d'avions…)

II.A.5.a Conversion du jeu de données

La principale différence dans la méthode d'évaluation d'Avigdor est qu'il considère les formulaires comme des ontologies, les types de champs comme des concepts et les champs eux même comme des instances de concepts. Ces informations sont stockées et réparties dans différents fichiers

Il a donc été nécessaire de convertir ce jeu de données dans notre propre format et de le sauvegarder en base pour la future évaluation.

La conversion a été basée sur la bibliothèque Java fournie avec le projet OntoBuilder. Toutefois celle-ci s'est fait avec quelques pertes négligeables en raison d'une part de la documentation ne collant plus toujours à l'API actuelle, et d'autre part dans la version actuelle de certains fichiers devenus illisibles.

Les appels au programme de conversion se font par le biais de scripts bash[3], qui en fonction de l'emplacement du fichier d'origine, passent en paramètre l'url du fichier et le domaine d'activité concerné.

La conversion achevée, un fichier CSV retraçant les ontologies converties est généré. Pour chaque ontologie il précise : le domaine d'activité, l'url, le nombre de formulaires sur la page et le nombre de champs.

Parmi les problèmes rencontrés, très couramment aucun libellé n'avait été détecté à la création du jeu d'essai, celui-ci se trouve alors remplacé par le nom interne du champ. De plus, peu d'instances sont présentes, empêchant une évaluation à partir de données saisies sur ce jeu d'essai.

Fort heureusement il est assez dense et nous donne quelques points de référence sur des formulaires correspondant bien de par la classification effectuée par AVIGDOR.

[3] Les scripts bash ont été écrits ici selon la syntaxe de GNU Bash 4 et sont donc incompatibles avec la version POSIX telle que rencontrée sur les systèmes BSD. Il est toutefois possible d'installer la version GNU de Bash via un outil comme Ports.

II.A.5.b Application de la méthode de mesure de proximité globale

Le jeu d'essai converti, enregistré en base et tracé, il convient de mesurer la proximité deux à deux les formulaires connus et d'analyser les résultats obtenus.

Pour cela, un autre programme dédié cette fois-ci à l'évaluation de la proximité. Celui-ci prend en entrée la trace CSV générée précédemment et interroge la base afin de charger et aligner les formulaires deux à deux selon la méthode de notre choix.

Les résultats de ces mesures sont ensuite stockés dans de nouveaux fichiers CSV, précisant dans leur nom la mesure de similarité locale employée, ce sur quoi elle porte et le seuil à partir duquel un alignement entre deux champs est retenu. *E.g. « expert_1-1_delta_name_jaro_0.65 ».*

Pour chaque alignement, ce fichier comprend les **domaines** d'activité, **nombres de champs** et **URL** des deux formulaires, le **score** de la mesure globale, une indication s'ils portent sur le **même domaine** et le **temps d'exécution**.

Ces fichiers générés, ceux-ci sont ensuite analysés à partir de scripts **awk**, puis on génère des graphiques via **Gnuplot**.

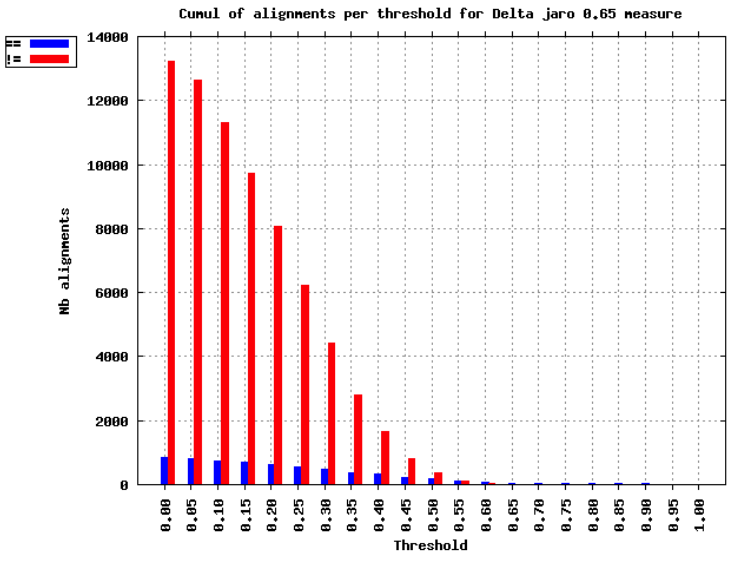
L'analyse des courbes tracées aide à la détermination des mesures locales a priori adaptées, et du **seuil global** nécessaire pour retenir suffisamment de valeurs tout en excluant le maximum de faux positifs.

Le graphique ci-dessous correspond à une mesure locale entre attributs *name* via Jaro et un seuil de 0.65.

Les **barres rouges** indiquent des formulaires de domaines d'activités différentes comme étant acceptés par l'alignement. Étant donné le jeu d'essai sur lequel nous travaillons, ces résultats-là doivent être éliminés.

Les **barres bleues** indiquent en revanche les formulaires de domaines identique, ces résultats sont alors vraisemblables.

On note alors que le seuil global pour cette mesure de proximité semble être aux alentours de 0.55. Seuls les paires de formulaires obtenant un score supérieur à ce seuil pour cette mesure sont retenus pour la seconde partie de l'évaluation de la mesure. Ils forment la **mesure experte** qui va servir de base d'évaluation : soit 382 paires de domaines différents et 873 de même domaine.



II.A.5.c Mesures d'évaluation

Trois mesures d'évaluation[2] permettent de déterminer si un alignement entre deux ontologies est correct ou ne l'est pas selon un certain seuil d'acceptabilité. Ces mesures sont les suivantes :

Précision

Permet de connaitre le nombre d'alignements corrects découverts par rapport au nombre d'alignements découverts total.

$$prec(al) = \frac{Nb\ alignements\ corrects}{Nb\ total\ alignements}$$

Rappel

Permet de savoir combien d'alignements ont été trouvés par rapport au nombre d'alignements experts.

$$rappel(al) = \frac{Nb\ corrects\ trouvés}{Nb\ corrects\ attendus}$$

F-Mesure

Moyenne des deux mesures précédentes. Elle vaut généralement :

$$fmes(al) = 2 \times \frac{prec(al) \times rappel(al)}{prec(al) + rappel(al)}$$

Elle peut-être pondérée par un terme α, de valeur 0,5 ci-dessus.

Ces mesures impliquent que l'on connaisse préalablement les alignements sur lesquels nous devons tomber. C'est fort heureusement ce qu'a fait Avigdor GAL pour certains domaines d'activités. Il nous suffit donc d'adapter ce qui a été fait sur ces domaines de formulaires et d'évaluer nos résultats à ceux du projet OntoBuilder via ces différentes mesures.

Malheureusement j'ai manqué de temps pour implémenter cette dernière partie de la méthode d'évaluation.

II.B PARTIE TECHNIQUE

La partie technique de mon stage concerne la réalisation du prototype démontrant de la faisabilité du projet. Cette partie décrit les problématiques rencontrées durant le développement.

II.B.1 APPLICATION PREEXISTANTE

Avant même que le projet prenne le nom d'Adnosco, un premier jet du modèle expérimental sous le nom de Portefeuille avait été réalisé par des étudiants de l'INSA. Mais comme le projet a continué d'évoluer, l'application ne correspond plus exactement à la vision actuelle.

Le projet était construit lui aussi autour d'une base de données MySQL avec un cœur applicatif en Java. La communication entre les deux éléments se fait par le biais de requêtes SQL en JDBC.

L'interception des données de formulaires se fait par le biais d'un proxy basé sur l'application OWASP Zed Attack Proxy[4]. Ce dernier avait été modifié pour pouvoir communiquer avec le cœur via RMI[5]. Les pages interceptées par le proxy sont aussi modifiées afin d'inclure le code JavaScript du client, permettant l'autocomplétion et le préremplissage.

À noter que l'application permet de créer son propre certificat SSL à inclure dans le navigateur afin que le proxy puisse intercepter les requêtes *https* sans avertissements impromptus.

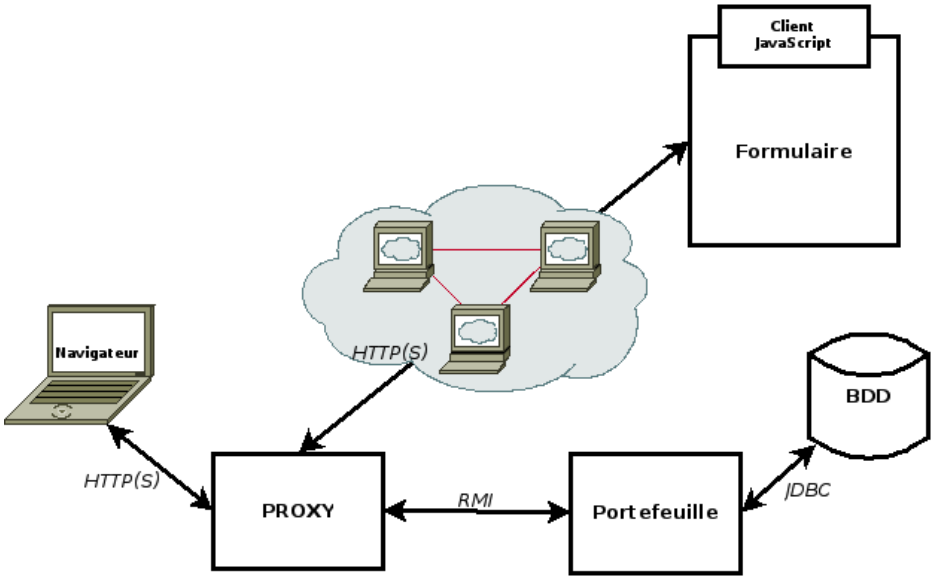


FIGURE 6 ARCHITECTURE DE L'ANCIENNE APPLICATION PORTEFEUILLE

[4] OWASP Zed Attack Proxy : http://code.google.com/p/zaproxy/
[5] RMI = Remote Method Invocation, API Java d'appels à distance

II.B.2 NOUVELLE ARCHITECTURE

J'ai toutefois choisi de partir sur de nouvelles fondations pour les raisons suivantes :

- Les requêtes SQL sont vraiment longues… J'ai préféré partir sur un modèle objet beaucoup plus modulable et lisible ;
- La modification des paramètres de proxy, et la génération/insertion d'un certificat racine peuvent dérouter des utilisateurs novices ;
- L'utilisation d'un service web REST[6] en lieu et place de RMI permet d'étendre plus facilement l'API à d'autres langages que Java ;
- L'insertion à la volée de code JavaScript en en-tête des pages interceptées par le proxy peut éventuellement poser des problèmes de collision.

L'emploi d'une extension de navigateur en guise de client m'a semblé être plus intéressant, car l'intégration aux pages visitées s'en trouve améliorée, l'installation facilitée par rapport à un proxy et certaines API propres au navigateur deviennent accessibles.

Mais bien que codées en JavaScript, les extensions peuvent poser aussi des problèmes :

- Le besoin de développer un **socle commun** pour s'adapter selon le navigateur.
  Nous avons basé notre extension sur le code de Lazarus qui fournit déjà un socle commun pour Firefox, Safari, Chrome (et par extension le nouveau Opéra).
- Le **cycle d'évolution rapide**[7] des navigateurs peut briser le fonctionnement de l'extension du jour au lendemain.
  En suivant les recommandations de développement de ces extensions, le problème ne devrait pas se poser.

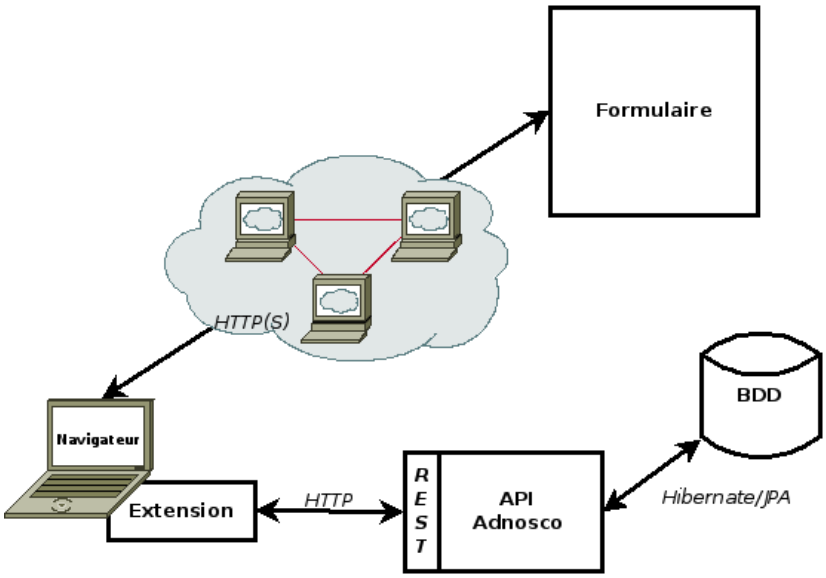


FIGURE 7 ARCHITECTURE DE LA NOUVELLE APPLICATION ADNOSCO

[6] REST = Representational State Transfer, style d'architecture pour les systèmes distribués

[7] Il se trouve que le passage à la version 22 de Firefox au mois de juin a cassé la compatibilité. Le socle devra être corrigé pour ce navigateu r.

À l'heure actuelle, ce prototype gère uniquement les requêtes sémantiques, mais l'ensemble des briques de base pour passer rapidement à la sémantique sont présentes.

II.B.3 CREATION DE LA BASE DE DONNEES

II.B.3.a Structuration de la base

La création de la base de données en elle-même est assez triviale en s'inspirant du diagramme de classe en *Annexe C*. Celle-ci a été normalisée en rajoutant un identifiant numérique comme clef primaire de chaque table.

Sachant que dans le contexte d'un formulaire le nom d'un champ est unique, je comptais originellement pour les tables *Field* et *FieldInstance* utiliser des clefs primaires composées de *name* avec respectivement *formId* et *formInstanceId*.

Deux problèmes se posaient dans cette configuration :

- Le nom du champ est dupliqué entre *Field* et *FieldInstance*, ce qui va à l'encontre de la normalisation ;
- Dans une balise HTML le champ *name* peut faire jusqu'à 65536 caractères. C'est un cas très inhabituel, mais cela fait qu'un tel champ utilisé en tant que clef primaire impose de ne référencer qu'un nombre limité de caractères et limite les performances.

II.B.3.b Mapping JPA[8]

La mise en place du mapping **JPA** est aussi assez simple aux premiers abords, ne nécessitant que la mise en place du fichier de configuration idoine, et l'annotation des méthodes des entités concernées.

Pour faciliter la manipulation des objets Java correspondants, j'ai opté pour des liens bidirectionnels entre les classes, ce qui amène évidemment à l'utilisation des annotations *OneToMany* et *ManyToOne*. Les liens sont configurés en *LAZY* pour n'interroger la base sur des données d'un autre objet que lorsque cela est nécessaire.

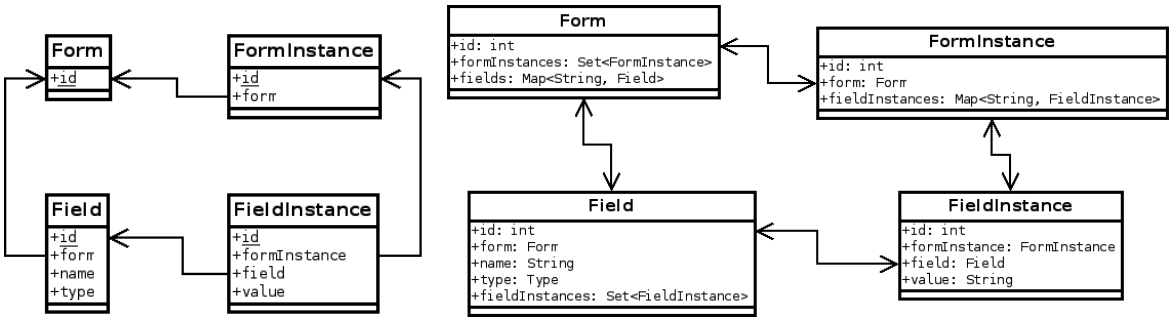


FIGURE 8 MODELE LOGIQUE ET IMPLEMENTATION JAVA

[8] JPA = Java Persistence API, interface de programmation pour organiser sous forme d'objets Java des données issues de bases relationnelles.

Le choix de tableaux associatifs a été fait pour faciliter la manipulation des *Fields* et des *FieldsInstances*.

La **principale difficulté** se trouve dans la classe *FormInstance* : il n'est pas possible d'annoter directement une Map prenant pour clefs les noms de champs. En effet, ceux-ci ne se trouvent ni dans *FormInstance*, ni dans *FieldInstance*.

La solution trouvée a été alors d'utiliser les instances Java de Field comme clefs pour le tableau associatif, et de gérer nous-même lors des insertions le référencement des instances de Fields en fonction de leur nom.

On a donc une *Map<Field, FieldInstance>* annotée et une *Map<String, Field>* que l'on gère nous-même. Les méthodes correspondantes permettent l'accès aux instances soit par instances de Field, soit par le nom correspondant.

Enfin plusieurs tests **JUnit** ont évidemment été réalisés durant la conception afin de s'assurer que l'insertion mais aussi l'extraction de données de la base se passe sans heurts.

II.B.4 SOUMISSION D'UNE REQUETE UTILISATEUR

Lorsque l'utilisateur cherche à compléter sa saisie de formulaire par du préremplissage ou de l'auto-complétion, celui-ci soumet une requête au cœur d'Adnosco comprenant les informations suivantes :

- le **domaine** du site Internet, par exemple « www.test.com » ;
- l'**url** du formulaire sur lequel il travaille ;
- l'**ensemble des champs** de ce formulaire, et les **valeurs** déjà saisies ;
- le nom du champ ayant le **focus** ;
- et éventuellement diverses informations comme le temps d'édition…

Ces interrogations se font sous la forme de requêtes nommées en JPQL[9]. La requête suivante est par exemple utilisée dans le contexte syntaxique pour récupérer les instances de formulaires correspondantes, c'est-à-dire dont la valeur associée au champ ayant le focus commence par une certaine chaîne de caractères :

```
SELECT fi
FROM FieldInstance fdi
JOIN fdi.form fi
JOIN fdi.field fd
JOIN fd.form f
WHERE LOWER(fdi.value) LIKE LOWER(:value)||'%'
AND fd.name = :fieldName
AND f.idDB = :formId
```

Le résultat de telles requêtes est stocké dans une **liste typée** selon la sélection effectuée *(ici des FieldInstance)*.

[9] **JPQL** : **J**ava **P**ersistence **Q**uery **L**anguage. Langage de requêtes orienté objets indépendant de la plateforme et défini dans la spécification JPA

L'utilisation des requêtes nommées en JPQL apporte une certaine couche d'abstraction quelque soit le SGBD[10] employé et facilite la manipulation des données au travers d'objets automatiquement mappés en entrée et en sortie de la base, là où il aurait fallu originellement traiter avec des requêtes SQL complexes.

En effet, ici on récupère une bonne fois pour toutes l'objet correspondant à notre *FormInstance*, et grâce au mode *LAZY* nous avons des proxies chargeant à la demande les entités tels que les *FieldInstance*s.

Tandis qu'avec une requête SQL classique, nous aurions soit récupéré notre *FieldInstance*, puis récupéré par d'autres requêtes les éléments nous intéressant, soit du charger l'ensemble des données en une seule requête lourde à traiter et à manipuler.

II.B.5 CONSTRUCTION DE LA STRUCTURE DE RESULTATS ATTENDUE

Nous avons vu que pour faciliter la manipulation des données, ce sont des objets mappés aux tables de la base de données qui sont traités.

Néanmoins, l'API souhaitée est basée sur des structures de données standards *Maps*, *Sets* et *Lists* pour les usages externes. C'est cette API qui est interrogée pour le préremplissage et la complétion en une seule passe.

Les données attendues sont organisées au sein d'une structure de donnée basée sur des tableaux associatifs. Un aperçu de la structure voulue donne :

```
{
 syntactic = {
  differentFields = {
   <valeur 1> = {
    <id instance 1> = {
     currentFieldName = <champ 2>,
     date = <date soummission 1>,
     values = {
      <champ 1> = <val champ 1>,
      <champ 2> = <val champ 2>,
      <champ 3> = <val champ 3>
     }
    },
    <id instance 2> = {
     currentFieldName = <champ 3>,
     date = <date soummission 2>,
     values = {
      <champ 1> = <val champ 1>,
      …
     }
    }
   },
   <valeur 2> = {…}
  },
  sameField = {…}
 },
 activity = {…},
 semantic = {…}
}
```

[10] **SGBD** : **S**ystème de **G**estion de **B**ases de **D**onnées. Exemples : MySQL, Oracle, SQLServer, PostgreSQL

Par niveau, on retrouve l'organisation suivante :

1. La distinction entre résultats issus de recherche **syntaxique**, **sémantique** ou par **activité** ;
2. La distinction des résultats **au sein d'une catégorie de recherche**. Par exemple pour une recherche syntaxique, ceux issus du même champ que le champ courant (préremplissage) et ceux issus de champs différents (auto-complétion) ;
3. La distinction par **valeur du champ** correspondant dans les suggestions et la valeur du champ courant ;
4. La distinction par **identifiant d'instance** dans la base ;
5. Les **informations de l'instance** de formulaire : champ correspondant, date de soumission et valeurs enregistrées.

Du côté du client, la réalisation d'une telle structure de données est a priori triviale puisqu'il s'agit d'une simple imbrication d'objets JavaScript.

Du côté du cœur de l'application, nous avons choisi de nous appuyer sur l'interface *Map* et ses implémentations. Cet appui sur les *Map* du paquetage *java.util*, plutôt qu'une encapsulation au sein de classes nouvelles, permet d'une part de s'appuyer sur une structure de données classique, et d'autre part alléger le code.

Néanmoins pour éviter des écritures lourdes et peu lisibles du genre :

```
Map<String, Map<Integer, Map<String, Map<String, String>>>> myMap =
   new HashMap<String, Map<Integer, Map<String, Map<String, String>>>>();
```

Et du fait que le langage Java ne permet pas l'utilisation d'alias, il a été choisi d'étendre l'interface *Map* et ses implémentations. Une instanciation d'un tableau associatif de valeurs prenant ainsi la forme bien plus exploitable de : `ValuesMap myMap = new ValuesLinkedHashMap();`

Toujours afin de faciliter l'exploitation des structures, et lorsque cela était possible, les clefs ont quant à elles été basées sur des énumérations et non de simples String :

```
enum ResultsCategory {ACTIVITY, SEMANTIC, SYNTACTIC};

Map<ResultsCategory, FieldsCategoriesMap>
```

II.B.6 TRI DES RESULTATS ET REMPLISSAGE DE LA STRUCTURE

Comme nous l'avons vu précédemment, ces données doivent être triées selon leur pertinence. Pour cela, il est nécessaire de trier la liste de résultats obtenue en sortie de la requête nommée. Le contenu de cette liste triée est, dans l'ordre de parcours, ensuite inséré dans la structure correspondante. L'ordre sera préservé dans cette dernière lorsqu'elle est basée sur l'implémentation *LinkedHashMap*.

Mais comment trier exactement ces données ?

À chaque instance de formulaire figurant dans la liste résultante est associé un tableau de scores. Le nombre de cases de ce tableau dépend de la catégorie de recherche qui le concerne. Par exemple :

- Pour les résultats d'une recherche **syntaxique** visant le **préremplissage** sont attendus deux valeurs portant dans l'ordre sur :
  - la **similarité** entre l'entrée utilisateur et l'instance de la liste,
  - puis sur la **fréquence** d'apparition de cette instance dans la liste.
- Tandis qu'une recherche **syntaxique** portant sur la **complétion** est attendu uniquement la **fréquence**.

Évidemment, les méthodes de calcul des scores remplissant ces tableaux ne sont pas fixées en dur. Elles sont définies au sein d'un fichier *measures.properties* de la forme suivante :

```
syntactic_sameField = adnosco.core.api.measures.Similarity,adnosco.core.api.measures.Frequency
syntactic_differentFields = adnosco.core.api.measures.Frequency
```

On appelle alors les méthodes de score par réflexivité en se basant sur les informations de ce fichier.

Les scores calculés et associés aux instances, les listes sont triées par un appel à la méthode : `Collections.sort(lrecords, new ComparatorClass());`[11]

Une fois les listes de résultat triées, il ne reste plus qu'à insérer les données dans la structure correspondante.

Ainsi, de par la nature des *LinkedHashSet* et des *LinkedHashMap*, nous nous retrouvons avec les données accessibles par clefs, et parcourable par ordre décroissant de pertinence.

II.B.7 WEBSERVICE REST ET SERIALISATION JSON

II.B.7.a Choix des bibliothèques

Nous avons donc notre API qui est interrogeable et qui nous fournit des résultats via des structures de données standards. Mais puisque nous voulons réaliser un service web REST, pour transférer les données il est nécessaire de configurer leur sérialisation dans des chaînes de caractères.

De plus, sachant que le client principal est réalisé en JavaScript, il parait judicieux d'utiliser le format JSON en lieu et place de XML. Celui-ci représente en effet les données avec la syntaxe d'un objet Javascript.

Moxy et Jackson sont deux bibliothèques dédiées à la sérialisation JSON et recommandées en utilisation avec Jersey. Celles-ci se basent par-dessus JAXB et en reprennent donc les annotations.

[11] ComparatorClass étant une classe implémentant l'interface Comparator.

Le mapping XML via JAXB d'une HashMap donne généralement une sérialisation XML de la forme :

```
<MaMap>
  <entry><key>clef1</key><value>valeur1</value></entry>
  <entry><key>clef2</key><value>valeur2</value></entry>
</MaMap>
```

Mais Moxy, tel qu'employé avec Jersey, reprend par défaut ce format dans la sérialisation JSON : `{{"key": "clef1", "value" : valeur1"}, {"key": "clef2", "value" : "valeur2"}}`. Il devrait être normalement possible de personnaliser le rendu via l'annotation XmlAdapter, mais celle-ci bien que reconnue ne fonctionnait pas.

Le format fourni par Moxy est donc inexploitable.

A contrario Jackson produit une sérialisation JSON parfaite des HashMap, soit `{"clef1":"valeur1", "clef2": "valeur2"}` pour notre exemple, mais aura des difficultés à produire un XML correct lorsque les clefs sont composées uniquement de nombres tels que les *identifiants d'instances de formulaires*.

Et une fois encore il est difficile (mais pas impossible) de personnaliser le rendu fourni par Jackson qui nous crée des objets XML incorrects tel que : `<7>Mavaleur</7>`.

Au final, le service est réalisé avec le trio de bibliothèques suivant : Jersey, Grizzly et Jackson-JAXRS.

II.B.7.b Conservation de la notion d'ordre côté client

Côté service nous avons les données structurées et ordonnées sérialisées au format JSON avant d'être envoyées au client. La représentation à ce moment-là respecte l'ordre qui a été affectée aux données.

Côté client, les données reçues sont désérialisées par le biais de la méthode *JSON.parse()*. Mais sans indication supplémentaire aucun ordre de parcours n'a été associé. Certaines machines virtuelles JavaScript iront parcourir l'objet généré dans l'ordre où les données ont été insérées, mais le cas général est l'ordre alphabétique des clefs.

Le problème est heureusement aisément soluble via les annotations propres à Jackson.

L'*Annexe F* présente des exemples d'annotations utilisées afin de conserver la notion d'ordre.

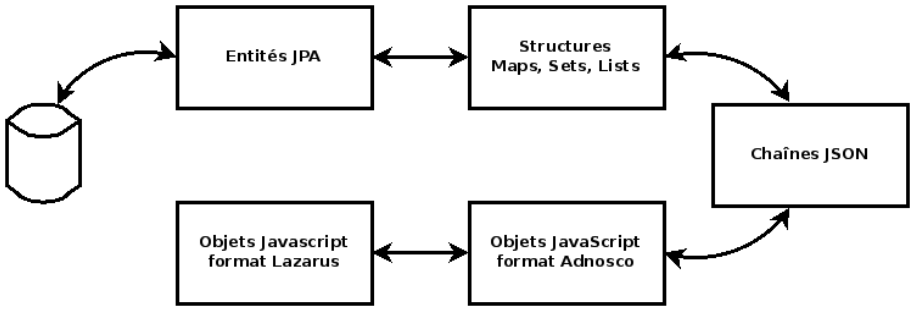


FIGURE 9 FORMATS DE DONNEES ET CONVERSIONS

II.B.8 REALISATION DU CLIENT SOUS FORME D'EXTENSION

Le **client principal** d'Adnosco se présente sous la forme d'une **extension** de navigateur, mais rien n'empêche techniquement l'emploi d'un autre client sous une forme différente.

L'idée d'employer une extension vient de **Lazarus**[12], extension dédiée à la restauration de formulaires. Trois avantages majeurs ressortent de celle-ci :

- Une bonne **intégration** dans l'interface, la liste de suggestions n'occultant ni celles prévues par le site, ni celle du navigateur ;
- L'extension est en mesure d'**intercepter** des données qui n'ont pas été soumises par le biais de l'évènement correspondant. Plus généralement, elle est en mesure d'intercepter des données de formulaires **à tout moment** ;
- Un « **socle** » a été réalisé par ses auteurs, donnant au développeur un accès à la même API quelque-soit le navigateur sous-jacent.

Concernant le « socle », typiquement l'exécution d'une extension sous Chrome peut être divisée en zones :

- Les **pages** chargées dans les onglets *(script content.js)*, où l'arbre DOM de la page est entièrement manipulable mais où il n'est pas possible d'accéder à certains appels bas niveau en direction du navigateur. Chaque page dispose de sa propre console de debuggage.
- L'**arrière-plan** *(background.js)*, permettant d'accéder à des fonctions de plus bas niveau. La console de debuggage associée est celle de l'extension.

Ces zones ne peuvent normalement pas communiquer directement entre elles si ce n'est en l'explicitant clairement dans chacune d'entre elles par un traitement asynchrone.

Le « **socle** » encapsule ces appels, fournissant une API identique peu importe le navigateur sur lequel le code fonctionnel s'exécute.

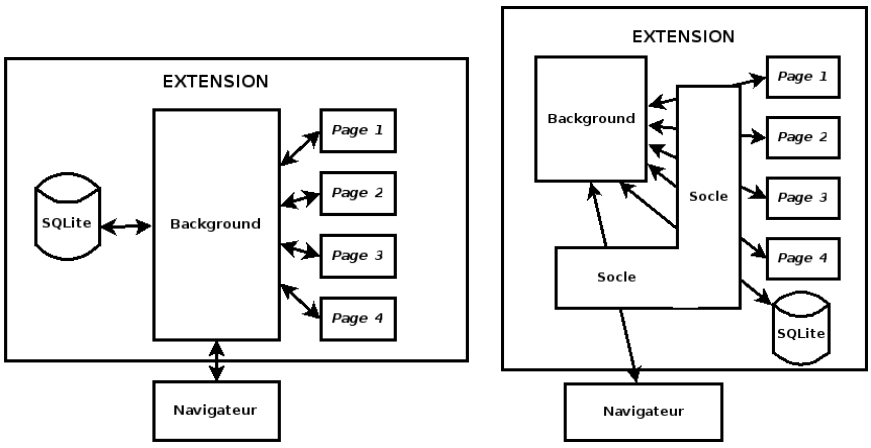


FIGURE 10 STRUCTURE D'UNE EXTENSION CHROME CLASSIQUE ET STRUCTURE AVEC « SOCLE » DE LAZARUS

[12] Site web de Lazarus : http://getlazarus.com/. Extension créée par Karl Dearden et Seth Wagoner

Afin d'en comprendre le fonctionnement, le client dans un premier temps s'appuie immédiatement sur le code de **Lazarus** qui est alors adapté pour requêter l'**API d'Adnosco** et exploiter les données retournées au lieu d'exploiter sa base de données SQLite locale.

À l'avenir, une **nouvelle version** du client non basée sur le code de Lazarus devrait être développée.

Il a été choisi de travailler sur la version **Chrome** de l'extension car ce navigateur possède un peu d'avance sur les formulaires HTML5 contrairement à **Firefox** *(qui est en revanche plutôt en avance sur CSS3)*.

De plus, le développement d'extensions y est un peu plus facilité vu qu'aucun reboot n'est nécessaire, à la manière des extensions *JetPack* de Firefox.

Néanmoins certaines difficultés ont été rencontrées durant l'adaptation de Lazarus pour notre besoin :

- La structure de la base de données, différente de la nôtre, **confond** ensemble les notions de **formulaires** et d'**instances de formulaires**.
- Tout comme les données représentées en base, les objets JavaScript présentent des **données** qui sont fortement **redondantes** et/ou n'exploitent pas les fonctions d'aggrégation.
  Au cas où, nous avions prévu des vues et un trigger en insertion adaptant notre base à ce qu'attend Lazarus. Pour l'instant elles ne servent pas, il est possible d'ignorer toutes ces données pour notre traitement.
- La base ne comporte **aucune contrainte de clef primaire, clef étrangère ni d'index**.
  Ce n'est pas grave puisque nous remplaçons les appels de la base de données locale par un appel vers notre service web.
- La notion d'**identifiant** employée dans le système est erronée, il s'agit en réalité d'un **hash**. Bien que la valeur générée n'ait en effet que peu de chance de provoquer une collision, le risque n'est pas nul pour autant.
  Nous avons donc distincts la notion d'identifiant et la notion de hash, tous deux déterminés dans le cœur d'Adnosco.
- Lazarus **purge** ses données de plus de quinze jours pour limiter le ralentissement qu'il provoque. Ce ralentissement est toutefois vraisemblablement causé par l'absence de clefs primaires et étrangères normalement utilisées lors des jointures. Cette purge limite aussi le risque de collisions des hash.
  La purge automatique a été désactivée.
- Lazarus est capable de **sauvegarder automatiquement** des données, même lorsque le formulaire n'a pas été soumis, afin d'éviter de la perte de données.

Initialement, cette sauvegarde automatique écrasait la dernière réalisée en raison du système d'id/hash originel.

Mais avec notre système d'id, chaque sauvegarde est considérée comme une nouvelle instance, ce qui alourdissait inutilement la base d'instances représentant des saisies intermédiaires.

Le choix a alors été fait de désactiver pour le moment cette fonctionnalité. Elle nécessite la réalisation d'un appel spécifique dans l'API et le service Web, et non de se baser sur la soumission de données classique.

- La liste de suggestions de Lazarus est sur **une seule colonne**, ne laissant que peu de place pour des informations supplémentaires telles que la date de soumission de l'instance de formulaire correspondant à une entrée.

  En jouant sur les paramètres de construction du menu et un peu CSS, il a été possible de rajouter une seconde colonne à droite de la suggestion.

  Un séparateur a aussi été ajouté afin de différencier visuellement les résultats de préremplissage et d'autocomplétion.

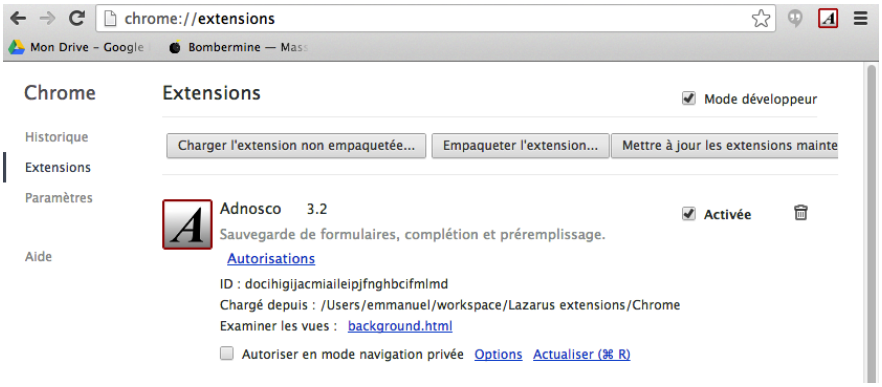


FIGURE 11 ENTREE DE L'EXTENSION DANS LE NAVIGATEUR

À ce niveau du développement, les problèmes suivant demeurent :

- Le **préremplissage** de formulaires **ajoutant/supprimant dynamiquement des champs** et de formulaires strictement identiques mais d'**url différente**.

  Le problème sera réglé de lui-même au passage sémantique qui établira les équivalences entre formulaires par le biais des qualifications sémantiques et ontologies.
- Certains formulaires n'utilisent pas l'évènement *submit (e.g. airfrance.fr)*, ce qui fait que le client n'est pas en mesure d'intercepter les données pour la sauvegarde.

  Ce problème peut être résolu en grande partie par la mise en place d'un système de sauvegarde automatique inspiré de celui de Lazarus.

  Il est envisageable aussi, mais beaucoup plus contraignant à mettre en place, la création d'une liste de sites à problèmes et permettant d'y adopter un comportement prédéfini pour intercepter les données.

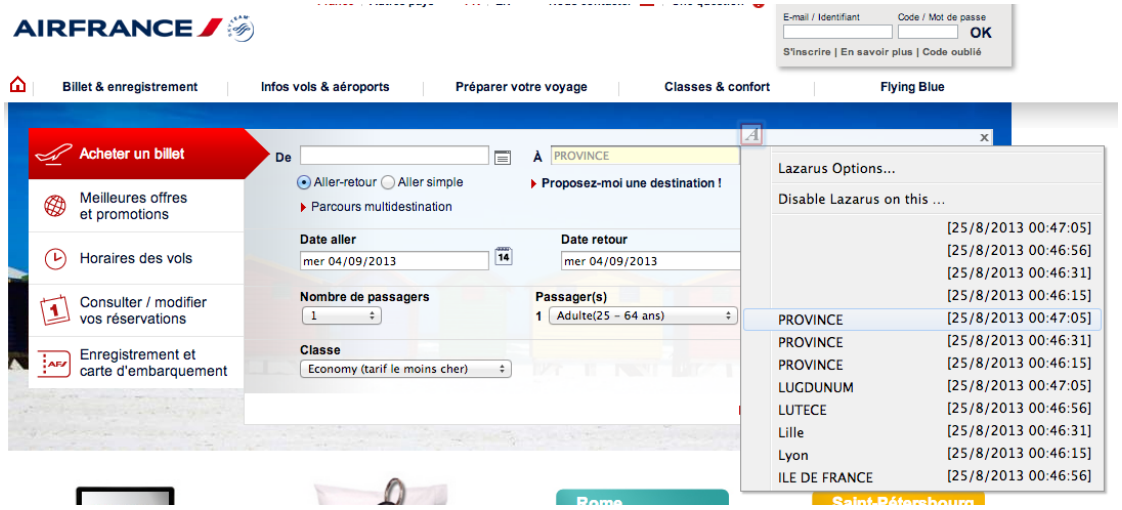


FIGURE 12 ABSENCE DE RESULTATS DE PREREMPLISSAGE SANS LA SAUVEGARDE AUTOMATIQUE

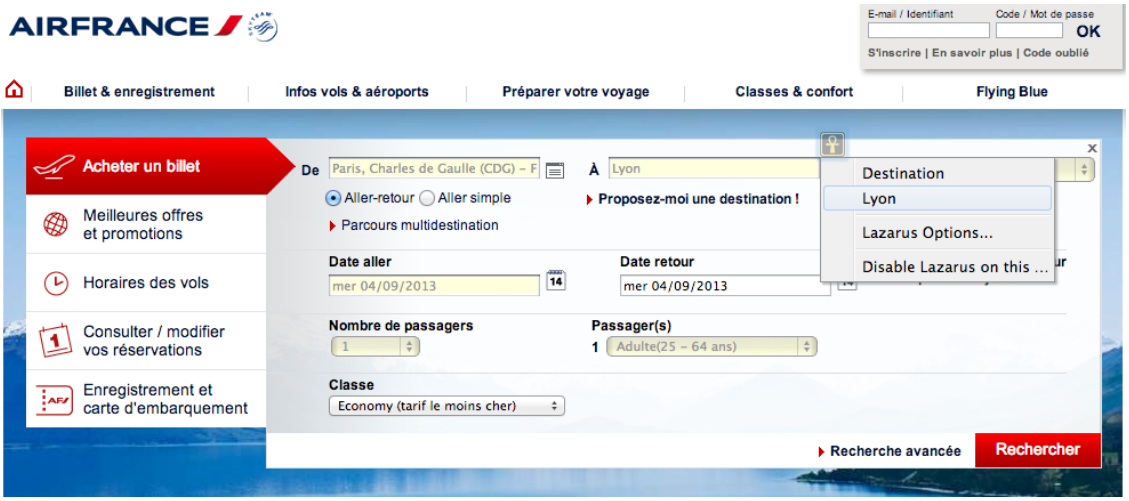


FIGURE 13 LAZARUS EST EN REVANCHE CAPABLE DE RESTAURER UNE PARTIE DE LA SAISIE AVANT SOUMISSION

III BILAN ET PERSPECTIVES

III.A POINT SUR L'AVANCEMENT FINAL

III.A.1 PARTIE RECHERCHE

En ce qui concerne la partie recherche, la démarche a été posée. Elle nécessite encore de la programmation, et de l'analyse afin de déterminer une bonne méthode d'alignement entre nouveaux formulaires et ontologies connues, mais les grandes lignes sont présentes.

III.A.2 MODELE EXPERIMENTAL

L'application fonctionne comme attendu au niveau syntaxique et peut être utilisée dans un usage courant.

L'analyse est bien avancée sur la partie sémantique et est déjà implémentée dans la base de données.

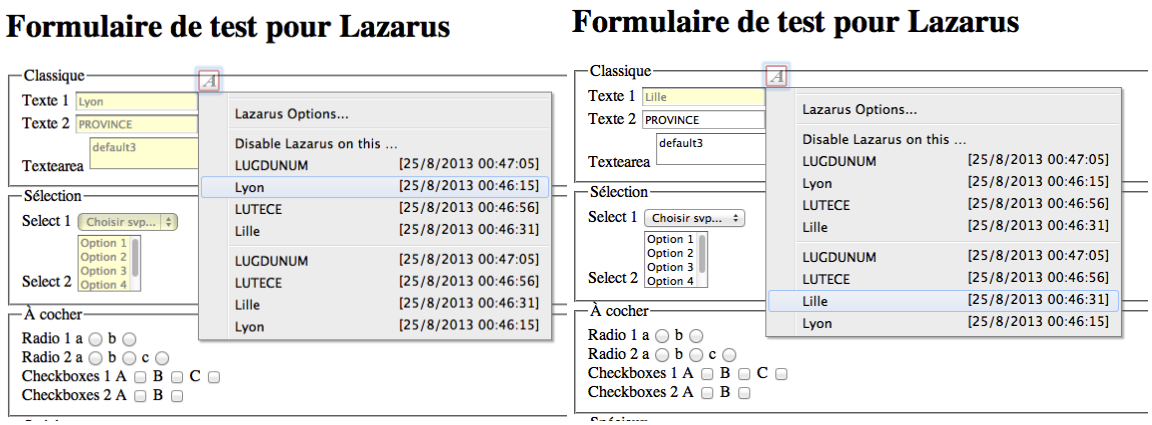


FIGURE 14 SUGGESTIONS EN REMPLISSAGE ET EN COMPLETION

III.B LIVRABLES

Les livrables attendus sont :

- Le cœur d'Adnosco avec un installateur ;
- Le client sous forme d'extension Chrome ;
- Le script SQL de création de la base de données ;
- Ce rapport ;
- Une machine virtuelle avec les environnements de démonstration et de développement distincts ;
- Les scripts et le jeu d'essai de la partie recherche ;
- Des documentations de développement, utilisation et installation.

À l'heure de la rédaction de ces lignes, je suis actuellement en train de rédiger la documentation d'installation et utilisation ainsi qu'à la préparation de la machine virtuelle.

III.C PERSPECTIVES

L'application est fonctionnelle d'un point de vue syntaxique. Parmi les prochaines étapes on peut estimer pouvoir rajouter :

- Le passage au **niveau sémantique**, en supposant les formulaires qualifiés à la main
- Le passage au **niveau activité**
- La **gestion d'utilisateurs**
- Le **chiffrement** du flux de données échangé entre l'extension et le cœur
- Une interface **d'interrogation des données utilisateurs**, toujours sous forme de service Web
- L'établissement d'un service dédié à la **sauvegarde automatique** à la façon de Lazarus.

III.D RETOUR SUR EXPERIENCE PERSONNELLE ET PROFESSIONNELLE

Lors de ce stage j'ai pu découvrir ce à quoi s'apparente le monde de la recherche.

J'ai eu l'occasion notamment de devoir chercher et lire de nombreux articles en lien avec le sujet à traiter. Et bien que familiarisé avec l'anglais courant et l'anglais technique du jargon informatique, ces lectures m'étaient parfois assez difficiles, l'anglais scientifique m'étant difficile à comprendre. Difficulté que j'avais aussi parfois pour trouver les bons mots clefs dans la recherche d'articles.

Plusieurs réunions et compte rendus ont rythmé mes semaines, fixant le cap et les tâches à effectuer. Celles-ci ont permis d'améliorer mon esprit de synthèse et de restitution.

En revanche, j'ai pu constater qu'il m'était encore difficile d'estimer le temps qu'il m'est nécessaire pour effectuer une tâche. De la même manière, basculer couramment entre les parties recherche et expérimentale me ralentissait car devant me remémorer du contexte à chaque fois.

D'un point de vue technique, j'ai pu aussi approfondir certaines technologies et pratiques étudiées lors de ces dernières années, telles que le mapping JPA ou la création et la mise en place de services REST. Tout ceci m'était connu, mais je n'avais pas eu jusqu'à présent l'occasion de m'interroger sur certaines problématiques pouvant être rencontrées durant leur emploi.

Ce fût aussi pour moi l'occasion de mieux comprendre les mécanismes inhérents au fonctionnement des extensions de navigateurs Internet, ayant jusqu'à présent réalisé uniquement des clients JavaScript intégrés à des pages web.

Enfin, d'un point de vue personnel, pour moi ce fût une très bonne expérience, malgré quelques soucis de santé que j'ai pu rencontrer durant cette période. Je ne sais pas si le monde de la recherche me convient vraiment, mais je reste dans l'envie de continuer d'en apprendre davantage sur la sémantique et sur la gestion des données utilisateur.

CONCLUSION

Ca stage m'a permis de découvrir en quoi consiste un travail de recherche, tout en mettant en œuvre les connaissances acquises jusqu'alors durant le cursus de master sur l'application servant à démontrer la faisabilité du projet Adnosco.

Ma tâche était d'une part d'en apprendre davantage sur les méthodes d'alignements et la qualification sémantique tout en réfléchissant à un moyen de tirer profit des informations déjà qualifiées, et d'autre part de poser les premières pierres de l'outil qui servira à la qualification sémantique des formulaires.

À l'heure de la rédaction de ce rapport, il me reste encore à réaliser la documentation d'installation et à compléter la documentation de développement pour les futurs repreneurs de l'application. Bien que celle-ci ne qualifie pas encore sémantiquement les formulaires en l'absence d'interface, les briques essentielles sont en place pour les futurs repreneurs.

Concernant mes perspectives d'avenir, je n'ai pas candidaté à une thèse, préférant ne pas m'engager fermement, pour des raisons personnelles. J'ai toutefois beaucoup apprécié en apprendre davantage sur le monde de la sémantique, et avoir un stage suffisamment diversifié entre technique et théorie.

Si mon stage est validé, et ainsi donc mon année, je me mettrais à la recherche d'un travail dans le développement Java ou PHP adapté à mon niveau de formation. Mon souhait étant depuis longtemps de partager du savoir dans mes domaines de connaissance, dès que l'occasion se présentera je tenterai éventuellement de m'inscrire à une thèse. En conclusion, ce stage a été une expérience enrichissante et c'est à partir d'aujourd'hui même que l'ensemble de mes choix va déterminer ma future carrière professionnelle.

BIBLIOGRAPHIE

DOCUMENTATION DEVELOPPEMENT

HTML

- **W3C, *HTML: The Markup Language (an HTML language reference)*, 28 mai 2013.** [http://www.w3.org/TR/html-markup]
  Référence du W3C pour la production de contenu HTML
- **W3schools.com, *HTML Reference (HTML5 Compliant),* 2013.** [http://www.w3schools.com/tags]
  Référence des balises et attributs HTML par W3Schools.

JAVA

- **Developpez.com, Section dédiée au langage Java.** [http://java.developpez.com]
  FAQ très complète et beaucoup de bonnes pratiques recommandées.
- **Bibliothèque Hibernate, *documentation*.** [http://www.hibernate.org/docs]
  Documentation officielle d'Hibernate.
- **Wikibooks, *Java Persistence*, Mai 2013.** [http://en.wikibooks.org/wiki/Java_Persistence]
  Cours sur l'utilisation des annotations JPA (notamment pour Hibernate)
- **Bibliothèque Jersey, *User Guide*.** [http://jersey.java.net/documentation/latest/user-guide.html]
  Guide utilisateur de Jersey.
- **Stack Overflow.** [http://stackoverflow.com]
  Site de questions/réponses relatives au développement. J'ai pu y trouver différents sujets sur la configuration et la faisabilité de certaines approches avec les bibliothèques utilisées.

JAVASCRIPT

- **Tout JavaScript.com, *La référence javascript, 2005.*** [http://www.toutjavascript.com/reference]
  Référence un peu ancienne mais servant d'aide mémoire pour la syntaxe de base.
- **Mozilla Developer Network, *JavaScript Reference*, 15 août 2013.** [https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference]
  Référence plus complète et à jour sur le langage.
- **Bibliothèque jQuery, *API Documentation*.** [http://api.jquery.com]
  Documentation de la bibliothèque jQuery.

- **Firebug, *Command Line API*, juillet 2013.**
  [https://getfirebug.com/wiki/index.php/Command_Line_API]
  Documentation de l'API de la ligne de commande pour le débugage HTML/JavaScript. C'est de cette API que s'inspirent celles fournies avec les navigateurs.
- **Google Chrome – Google Developers, *Command Line API Reference*, août 2013.**
  [https://developers.google.com/chrome-developer-tools/docs/commandline-api]
  Documentation de la ligne de commande de Chrome.

EXTENSIONS CHROME

- **Google Chrome – Google Developers, *Getting Started: Building a Chrome Extension*, 2013.** [http://developer.chrome.com/extensions/getstarted.html]
  Documentation officielle pour débuter dans la création d'extensions Chrome.
- **DANGIN, Sylvain, *Introduction à la programmation d'une extension Google Chrome*, Developpez.com, 10 décembre 2009.**
  [http://sylvain-dangin.developpez.com/tutoriels/general/introduction-programmation-extension-google-chrome]
  Un tutoriel d'introduction à la création d'extensions Chrome.
- **dkey, *Extensions pour Chrome*, Site du Zéro, 8 janvier 2013.**
  [http://www.siteduzero.com/informatique/tutoriels/extensions-pour-chrome]
  Un autre tutoriel d'introduction.

DOCUMENTATION RECHERCHE

- **EUZENAT, Jérôme, *D2.2.3: State of the art on ontology alignment*, 2004. [1]**
  État de l'art sur l'alignement entre ontologies, présentant différent types de mesures de similarités locales et globales, et différents systèmes les implémentant.
- **EUZENAT, Jérôme, *Towards a methodology for evaluating alignment and matching algorithms*, 2005. [2]**
  Présente les méthodologies utilisées pour le projet OAEI (Ontology Alignment Evaluation Initiative). Apport de mesures d'évaluation telles que Précision, Rappel et F-Mesure, et comment agréger les mesures.
- **BILKE, Alexander et NAUMANN, Felix, *Schema Matching using Duplicates*, 2005. [3]**
  Présente comment comparer des ensembles de données en ne se basant pas sur des noms de colonnes opaques (de sources hétérogènes), mais sur les instances.
- **MARIE, Anan et GAL, Avigdor, *Boosting Schema*, 2008. [4]**
  Approche façon « machine-learning » pour combiner les outils d'alignements et donne des matchers possibles sur : termes, valeurs, composition/hiérarchie, précédence, terme et valeur, combinaison pondérée.

- ***KONDRAK, Grzegorz, N-gram similarity and distance*, 2005. [5]**
  Présente des mesures de similarité et de distance basées sur N-Gram. La distance d'édition (ou de Levenstein) est un cas particulier de ces mesures.
- **BENNANI, Nadia, DUCHATEAU Fabien, EGYED-ZSIGMOND, Elöd et LAMARRE Philippe, *Adnosco: a system to farm users' data*, 2013. [6]**
  Brouillon de l'article du projet Adnosco.
- **VIVIANI, Marco, BENNANI, Nadia et EGYED-ZSIGMOND, Elöd, *G-Profile: A Hybrid Solution for Extended Identity Management in the Field of Personalized Service Provision*. [7]**
  Article sur le projet G-Profile consistant à la gestion des données d'identification d'un utilisateur au sein d'un seul service.
- ***Hungarian Algorithm*, Wikipedia, 6 août 2013. [8]**
  [http://en.wikipedia.org/wiki/Hungarian_algorithm]

ANNEXE A - ALGORITHME HONGROIS

Soit une matrice M de $n \times n$ réels positifs représentant des scores entre paires de deux ensembles de dimension n chacun.

L'algorithme Hongrois est un algorithme d'optimisation combinatoire résolvant les problèmes d'affectation en temps polynomial. Il permet de déterminer l'ensemble des paires n'impliquant qu'une seule fois les éléments de chaque matrice pour un score total minimal.

Il peut être adapté pour une recherche de score maximum en prenant les valeurs opposées à celles initiales, et en remplaçant les valeurs originellement à zéro par une valeur m supérieure à la valeur maximale de la matrice.

Si la matrice M n'est pas carrée, il suffit de l'adapter en ajoutant des lignes ou des colonnes de zéros.

La matrice $M = \begin{bmatrix} a_1 & a_2 & a_3 & a_4 \\ b_1 & b_2 & c_3 & c_4 \\ c_1 & c_2 & c_3 & c_4 \\ d_1 & d_2 & d_3 & d_4 \end{bmatrix}$ illustrera l'algorithme dans notre exemple.

1. Pour chaque ligne de la matrice soustraire la valeur minimale de celle-ci.
   La matrice a désormais au moins un zéro par ligne.
2. Même chose avec les colonnes. La matrice a désormais au moins un zéro par ligne et par colonne :
   $$M = \begin{bmatrix} 0 & a_2 & a_3 & a_4 \\ b_1 & b_2 & c_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ d_1 & 0 & 0' & d_4 \end{bmatrix}$$
3. Parcourir tous les zéros et les marquer en rouge s'ils ne sont pas la même ligne ou même colonne qu'un zéro déjà marqué.
   $$M = \begin{bmatrix} 0 & a_2 & a_3 & a_4 \\ b_1 & b_2 & c_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ d_1 & 0 & 0' & d_4 \end{bmatrix}$$
   Si n zéro *(ici 4)* sont sélectionnés, on arrête l'algorithme.
4. Couvrir chaque colonne ayant un zéro marqué.
   $$M = \begin{bmatrix} 0 & a_2 & a_3 & a_4 \\ b_1 & b_2 & c_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ d_1 & 0 & 0' & d_4 \end{bmatrix}$$
   a. Si un zéro est non couvert, le marquer d'un '.

i. Si un zéro est marqué sur la même ligne, découvrir la colonne et couvrir la ligne.

$$M = \begin{bmatrix} 0 & a_2 & a_3 & a_4 \\ b_1 & b_2 & c_3 & 0 \\ c_1 & c_2 & c_3 & 0 \\ d_1 & 0 & 0' & d_4 \end{bmatrix}$$

ii. Sinon passer à l'**étape 5**.

b. Si tous les zéros sont couverts, passer à l'**étape 6**.

5. Si le nombre maximal de zéros indépendants n'a pas été sélectionné :

   Construire la suite z_i telle que :

   a. z_0 : unique $0'$non couvert
   b. z_1 : zéro marqué sur la colonne de z_0
   c. $z_{i \mid i\ pair}$: $0'$ sur la ligne de z_{i-1}
   d. $z_{i \mid i\ impair}$: $0'$ sur la colonne de z_{i-1} s'il existe, sinon la suite s'arrête.

   La suite z_i comprend un *0'* de plus que de zéros marqués. Substituer directement ces *0'* aux zéros marqués de la suite.

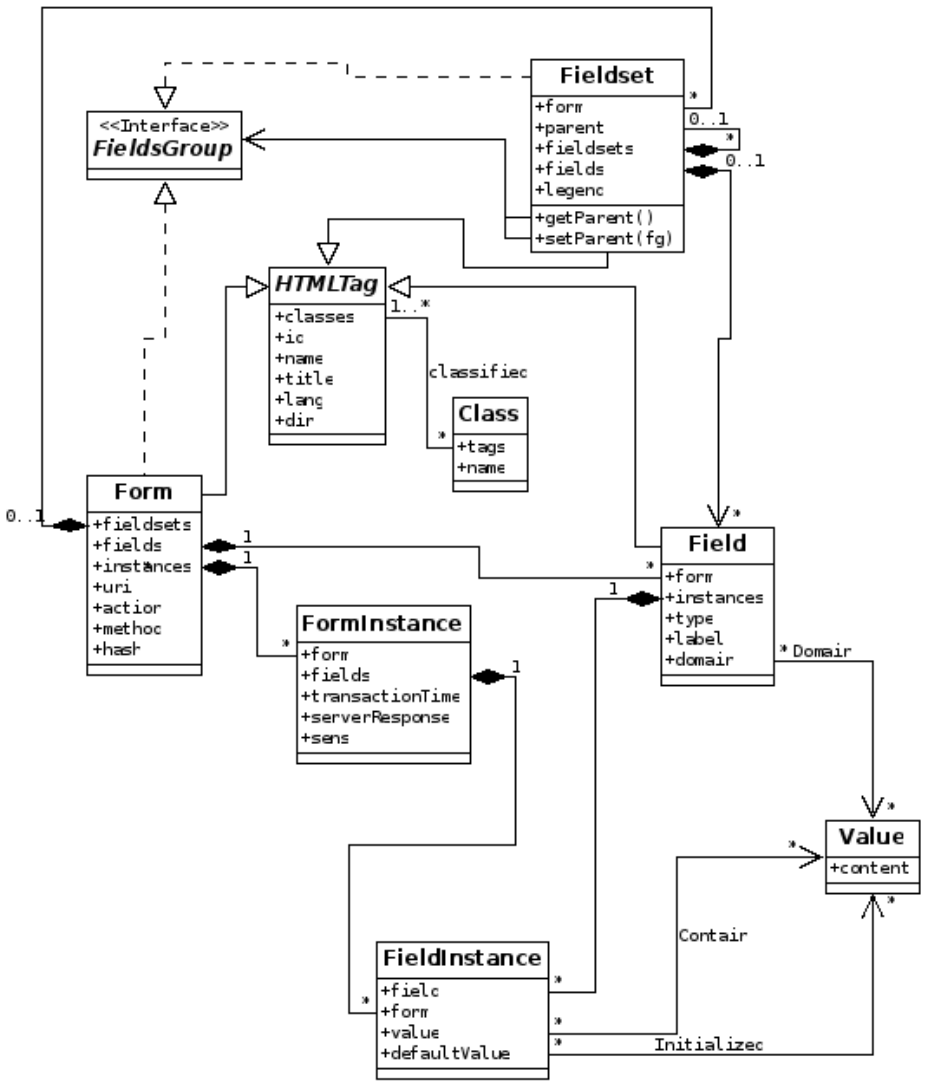
   Retourner à l'**étape 3**, en gardant les nouveaux zéros marqués mais en effaçant les primes et les couvertures.

6. Si tous les zéros ont été couverts :

   a. Trouver la valeur minimum de la sous-matrice des éléments non-couverts en **étape 4**.
   b. Ajouter cette valeur à toutes les lignes couvertes, et la retirer à toutes les colonnes non-couvertes.
   c. Retourner à l'**étape 3**, en conservant le marquage des zéros, la couverture les lignes et colonnes, et les primes.

ANNEXE B – STRUCTURE D'UN FORMULAIRE

Une analyse des informations sémantiquement intéressantes dans un formulaire donne le diagramme de classes suivant :



À noter que dans l'implémentation actuelle, les valeurs multiples sont implémentées sous la forme d'une sérialisation JSON dans le champ value de FieldInstance. C'est le modèle choisi par l'extension Lazarus ayant servie de base pour le client.

Une page Web n'étant pas immuable dans le temps, celle-ci est identifiée par l'ensemble **{url, {ensemble de champs}}**. Un hash est généré et indexé pour faciliter la recherche d'un formulaire.

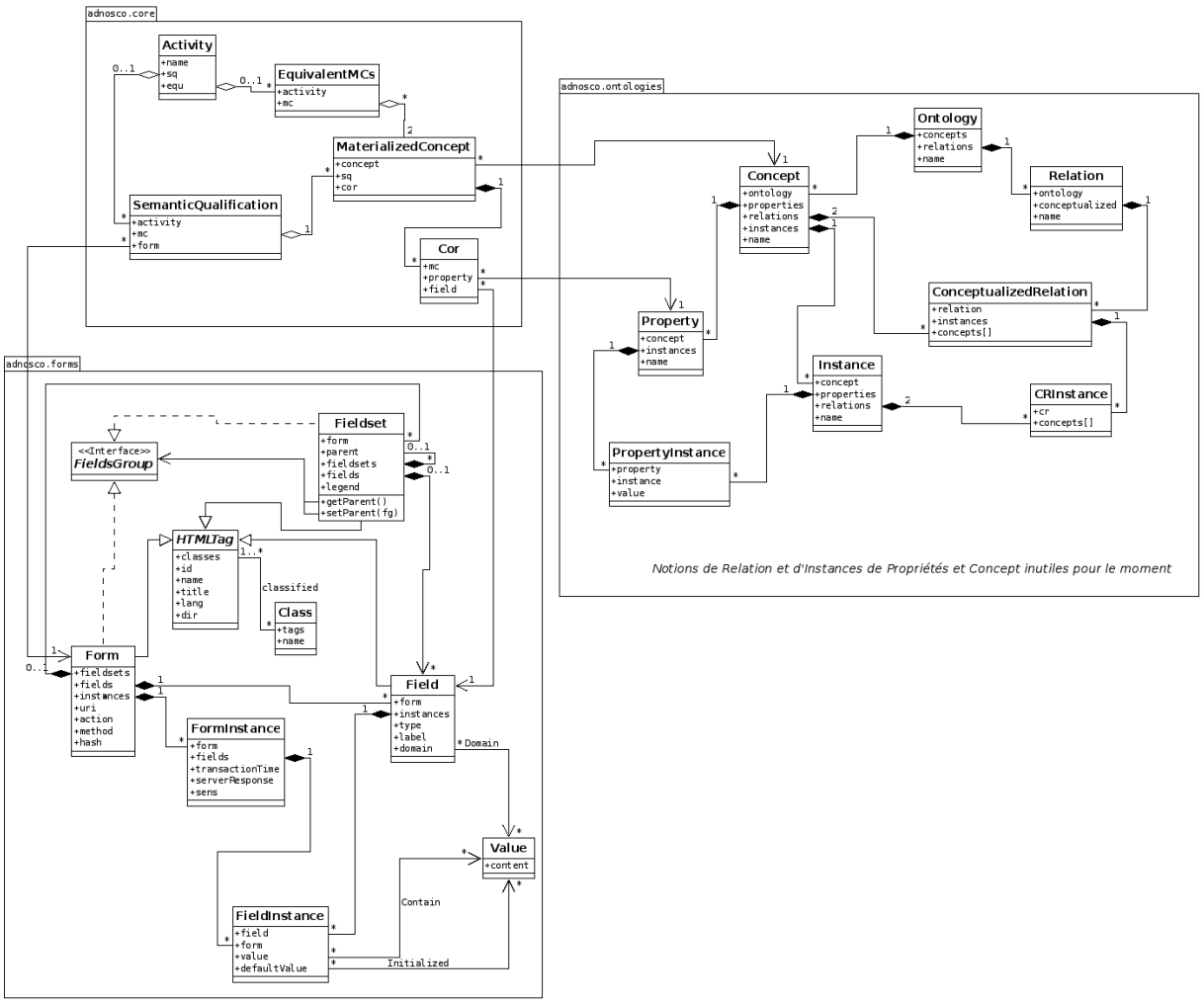
Les notions de groupes de champs, de classes HTML ne sont actuellement pas pris en compte… mais peuvent être à l'avenir être des informations utiles dans la compréhension sémantique d'un formulaire.

ANNEXE C – DIAGRAMME DE CLASSES DU CŒUR D'ADNOSCO

Le diagramme de classes suivant présente le cœur du projet tel que je le conçois après analyse.

Comme déjà précisé précédemment, certaines classes du paquetage des ontologies ne sont pas utilisées car pour l'instant inutiles dans le projet.

De même dans le paquetage des formulaires, la séparation des valeurs multiples, ainsi que l'utilisation des classes HTML ont été ignorés pour simplifier.

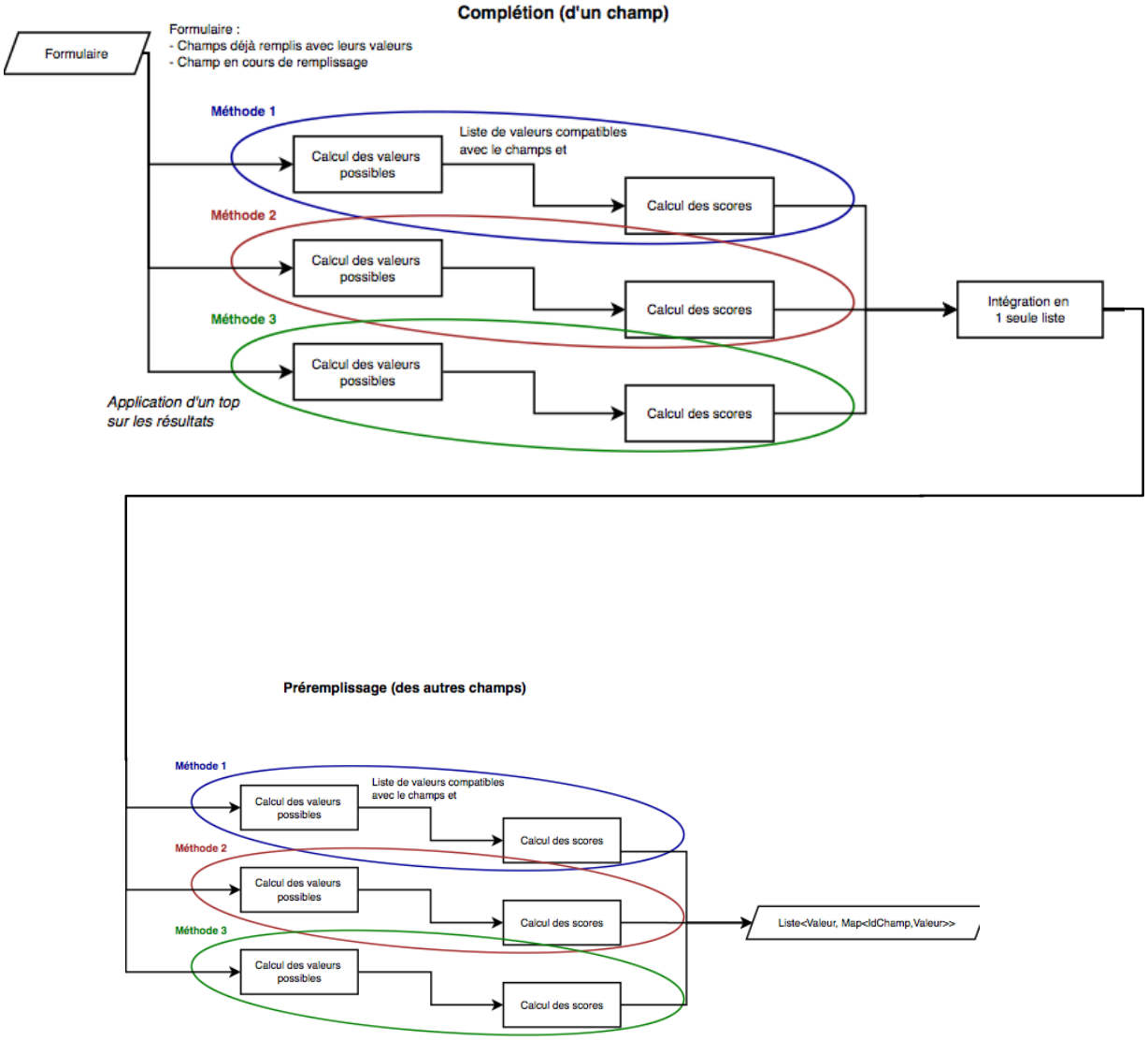


Pour les formulaires, la Modèle Logique de Données correspondant est composée des tables *Form*, *FormInstance*, *Field*, *FieldInstance*, *Domain*, *Fieldset* et *Type*.

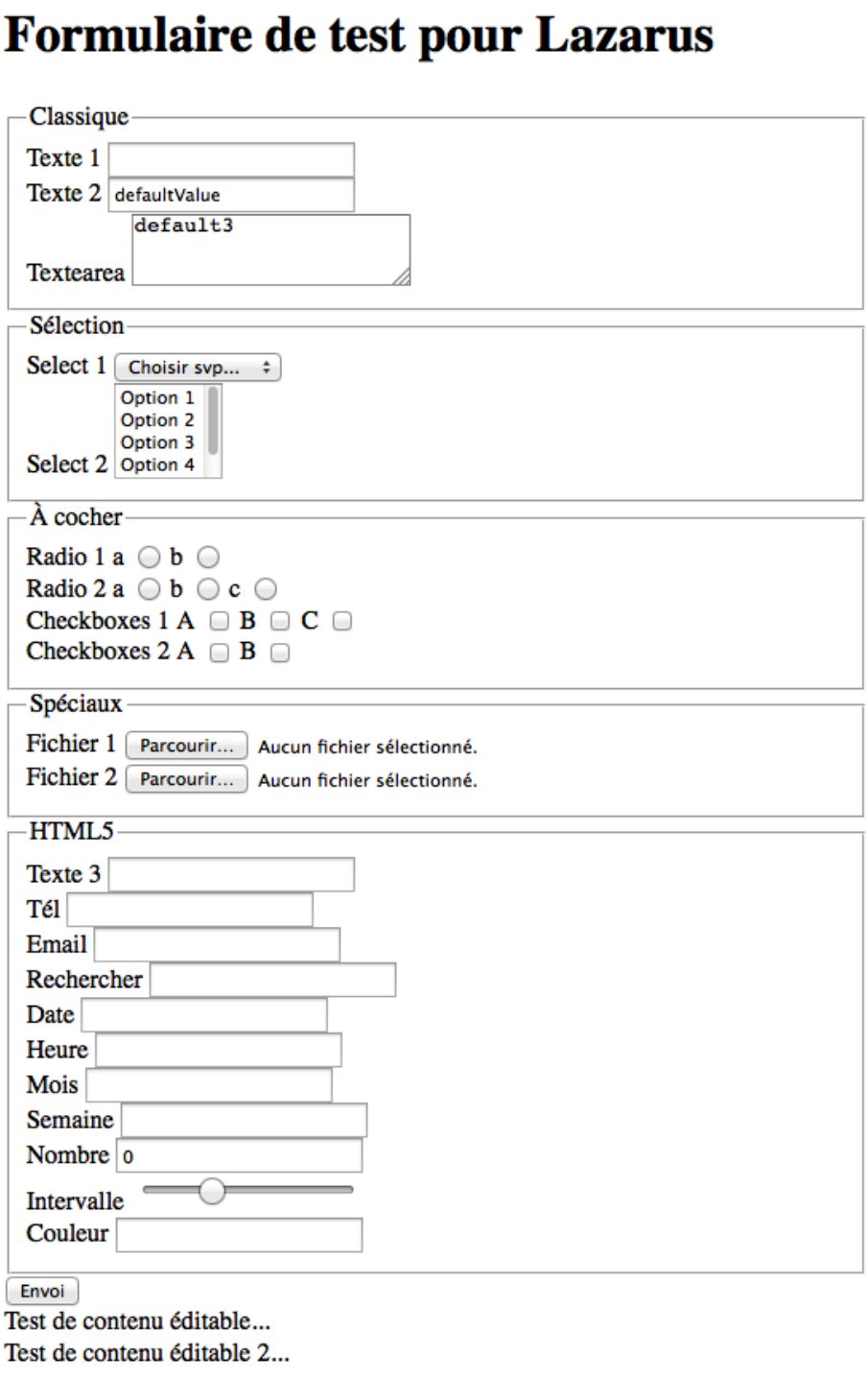
Les informations portées par les classes HTMLTag et FieldsGroup sont exprimées directement dans les tables auxquelles elles se rapportent.

ANNEXE D – SYNTHESE DES RESULTATS DE RECHERCHE LORS DE L'INTERROGATION DE L'API

Le schéma suivant présente les différents appels et fusions effectuées dans la génération d'une liste unique dédiée à l'autocomplétion et au préremplissage.



ANNEXE E – FORMULAIRE DE TESTS



ANNEXE F – EXEMPLE D'ANNOTATION POUR MAPPING JSON

CONSERVATION DE L'ORDRE VIA UN INDEX

INTERFACE MERE

```
@JsonSerialize(as=ValuesMapJSON.class)
public interface ValuesMap extends Map<String, InstancesMap> {}
```

INTERFACE DE RENDU JSON : UNE METHODE RETOURNANT UN INDEX, L'AUTRE LA MAP ELLE-MEME

```
public interface ValuesMapJSON {
 /** Clefs rangées dans l'ordre d'insertion si LinkedHashMap. */
 @XmlElement(name="sortedIndex")
 public Set<String> keySet();

 @XmlElement(name="map")
 @JsonSerialize(as=HashMap.class)
 public Map<String, InstancesMap> getMap();
}
```

CLASSE D'IMPLEMENTATION

```
public class ValuesLinkedHashMap
 extends LinkedHashMap<String, InstancesMap>
 implements ValuesMap, ValuesMapJSON {
  public Map<String, InstancesMap> getMap() {
   return new HashMap<String, InstancesMap>(this);
  }
}
```

SERIALISATION DES ENUMERATIONS SERVANT DE CLEFS

```
@JsonDeserialize(as=FieldsCategoriesLinkedHashMap.class)
public interface FieldsCategoriesMap extends Map<FieldsCategory, ValuesMap> {
 /** Catégorie d'entrées. */
 @XmlEnum
 public static enum FieldsCategory {
  /** Même champ du même formulaire. */
  @XmlEnumValue("sameField")
  SAME_FIELD("sameField"),
  /** Champs différents de tout formulaire (mais de même type). */
  @XmlEnumValue("differentFields")
  DIFFERENT_FIELDS("differentFields");

  private final String value;

  FieldsCategory(String str) {
   value = str;
  }

  @JsonValue
     public String toString() {
         return value;
     }
 };
}
```