Projet Web Sémantique

BENATHMANE Ayoub

MISSAOUI Ahmed

Lien de démonstration : http://youtu.be/MRghuUw4XDw

I. Sujet et objectifs

Dans le cadre du mini projet de ce module, nous proposons la réalisation d'une application dont le sujet est intitulé : "**Guide de Voyage touristique**", Elle s'appuie sur les standards du web : HTML, CSS, et les standards du web sémantique : RDF, SPARQL, SKOS et OWL.
Notre application répond aux attentes des voyageurs voulant visiter des villes dont ils ne connaissent rien à propos. En effet, nous présentons à nos utilisateurs des descriptions de villes, la liste des monuments , hôtels , et restaurants . Elle les aidera ainsi à mieux voyager. Notre applications offre également un système de recommandation suggérant la liste des endroits à visiter et à ne pas manquer.

- **une description de ce que nous avons fait/pas fait et pourquoi**

L'application offre 5 services aux utilisateurs :

1. **Vol :** selon le type de vol, l'application vous recommande des restaurants, des monuments, des hôtels dans la ville de destination
2. **Ville :** il faut choisir une ville parmi une liste prédéfinie de 7 villes, pour avoir une brève description, quelques informations et images sur cette ville. Si la ville recherché n'est pas dans cette liste, vous pouvez l'insérer dans le champs texte et faire une recherche directement sur DBpedia.
3. **Monument :** à partir d'une ville sélectionnée, l'application affiche une liste des monuments de cette ville avec quelques informations supplémentaires (Description et image).
4. **Hôtel :** à partir d'une ville sélectionnée, l'application affiche une liste des Hôtels de cette ville avec quelques informations supplémentaires (Adresse, Note, Site web et image)
5. **Restaurant :** à partir d'une ville sélectionnée, l'application affiche une liste des Restaurants de cette ville avec quelques informations supplémentaires (Adresse, Note, Site web et Numero de téléphone)

II. Ontologie

Pour la construction de notre ontologie nous avons choisi d'utiliser OWL. Ce dernier permet de construire des ontologies sémantiquement plus expressives qu'avec RDFS. Car il propose de nouvelles types de propriétés (transitive,..) mais aussi des restrictions sur des propriétés lors de définition des classes.

Pour la documentation de notre ontologie nous avons choisi d'utiliser SKOS bénéficiant des propriétés tel que "skos:defintion"… Recommandé par M. CORBY, nous avons séparé la documentation de l'ontologie afin de ne pas avoir OWL et SKOS dans le même fichier et la liaison est assuré par des URIs.

Pour plus de sémantique nous avons défini des règles d'inférence en utilisant le langage Rule. Ces règles ont pour objectif de recommander des hôtels , des monuments , et des agence en se basant sur le type de vol. Par exemple si le vol est de type loisir vers une ville X alors on le recommande un ensemble de monuments de cette ville.

Les classes OWL :

- Ville : *Représente l'objet ville, elle a une restriction sur la propriété "contientMonument" qui réfère sur la classe "Batiment" et elle est une sous classe de la classe "Emplacement"*
- Emplacement : *représente l'objet emplacement, qui représente un lieu (ville ou pays) et elle est disjointe à la classe "Batiment"*
- Batiment : *Représente l'objet bâtiment, elle a une restriction sur la propriété "dans" qui réfère sur la classe "Emplacement", et elle est disjointe à la classe "Emplacement"*
- Hotel : *Représente l'objet hôtel, elle est une sous classe de la classe "Batiment"*
- Vol : *Représente l'objet vol*
- VolAffaire : *Représente l'objet vol de type affaire, elle est une sous classe de la classe "Vol"*
- VolLoisir : *Représente l'objet vol de type loisir, elle est une sous classe de la classe "Vol"*
- AgenceLocation : *Représente une Agence Location*
- AgenceLocationVoiture : *Représente une Agence Location des voitures, elle est une sous classe de la classe "AgenceLocation"*
- Restaurant : *Représente l'objet Restaurant, elle est disjointe à la classe "Monument" et une sous classe de la classe "Batiment"*
- Monument : *Représente l'objet Monument, elle est une sous classe de la classe "Batiment"*Pays : *Représente un pays*URL : *Représente un lien pour designer une image ou un site web*

Les propriétés OWL :

- dans : *propriété de type transitive, permet de designer si un "Bâtiment" (hotel, monument, restaurant ou agence) situe dans une "Ville", et si cette ville est dans un "Pays" alors le raisonneur OWL infère automatiquement qu'il se situe dans ce "Pays"*
- siteWeb : *propriété de type URL*
- imageUrl : *propriété de type URL*
- contientMonument : *propriété de type ressource, indique qu'un "Emplacement" contient un "Monument", elle est inverse à la propriété "estContenueDans"*
- estContenueDans : *propriété de type ressource, indique qu'un "Monument" se situe dans un "Emplacement", elle est inverse à la propriété "contientMonument "*
- destination : *propriété de type ressource, indique qu'un "Vol" à pour destination une "Ville"*
- depart *: propriété de type ressource, indique qu'un "Vol" à pour départ une "Ville"*
- recommenderMonument : *propriété de type ressource, indique que pour un "Vol" à destination d'une ville, on recommande une ensemble de "Monument"*
- recommenderHotel : *propriété de type ressource, indique que pour un "Vol" à destination d'une ville, on recommande une ensemble de "Hotel"*
- recommenderAgence : *propriété de type ressource, indique que pour un "Vol" à destination d'une ville, on recommande une ensemble de "Agence" de location*
- nom, description, departement, codePostale, localisation, adresse, telephone, note : *Propriétés de type DatatypeProperty.*

III. Construction du modèle RDF

Notre Modèle RDF est construit en utilisant des données issues de divers sources.
Voulant mettre en évidence l'aspect Linked Data du web sémantique , nous avons récupérer des données de DBpedia décrivant les viles et ses monuments.
Notre application exploite des données en format CSV. Ces données décrivent les vols, les hôtels , les restaurants , les agences de location.
Ces données sont construites de deux façon différentes :
Pour les hôtels , les restaurants , les données ont été récupérées du site le plus connu du tourisme "Tripadvisor " .Et cela , en développant une outils se connectant à ce site et récupère les données puis construit les fichiers CSV pour chaque ville. Ce crawler est développé en utilisant PHP (en s'inspirant du projet de l'année dernière avec madame Elena Cabrio).

Pour les vols et les agence , nous avons définit les fichier CSV manuellement.

Tous ces données servent de base pour la construction de notre modele RDF . Cette construction est réalisé en utilisant un programme java développé avec le projet apache Jena. Ce module appelée RDF_genrator est fourni dans la fichier livré.

Ce dernier génère les modèle RDF en utilisant le format N-Triple.
il peut être lancé avec la commande :
 mvn exec:java -Dexec.args="construct-Model outputPath "

Inférences

1. Relation transitive :

Si x propriété y et y propriété z alors inférer x propriété z, exemple si Volluxauto est situé à Nice, et Nice est situé en France alors Voluxauto est situé en France.

Données d'entrée :

```
tourisme:France a tourisme:Pays .
tourisme:Nice a tourisme:Ville;
tourisme:contientMonument dbpediaresource:
Carnavalet_Museum ;
tourisme:dans tourisme:France .
tourisme:Volluxauto a tourisme:AgenceLocationVoiture ;
tourisme:dans tourisme:Nice .
```

Requete Sparql :

```
PREFIX tourisme: <http://www.polytech.semantique/tourisme#>
        select * where {
        ?x tourisme:dans ?y
```

}

Résultat :

<result>
<binding name='x'><uri>http://www.polytech.semantique/tourisme#Volluxauto</uri></binding>
<binding name='y'><uri>http://www.polytech.semantique/tourisme#France</uri></binding>
</result>

2. Relation inverse :

Dans notre ontologie, nous avons défini que la relation "contientMonument" est l'inverse de "estContenueDans", ex : si Paris contientMonument Tour Eiffel, alors on déduit que Tour Eiffel estContenueDans Paris, et pour tester cela :

Données d'entrée :

tourisme:Nice a tourisme:Ville;
tourisme:contientMonument dbpediaresource:
Carnavalet_Museum ;
tourisme:dans tourisme:France .

Requête Sparql :

PREFIX tourisme: <http://www.polytech.semantique/tourisme#>
select * where {
?x tourisme:estContenueDans ?y
}

Résultat :

<result>
<binding name='x'><uri>http://dbpedia.org/resource/Carnavalet_Museum</uri></binding>
<binding name='y'><uri>http://www.polytech.semantique/tourisme#Nice</uri></binding>
</result>

3. Relation de restriction :

Dans notre ontologie, nous avons défini que la classe Batiment à une restriction sur la propriété "dans" qui peut prendre en valeur un emplacement, par exemple : si un Batiment X est situé dans un endroit E, alors on déduit que E est un Emplacement avec notre règle, et pour tester cela :

Donnée d'entrée :

tourisme:SansDans a tourisme:Monument ;
tourisme:dans tourisme:Cannes .

Requête Sparql :

PREFIX tourisme: <http://www.polytech.semantique/tourisme#>
select * where {
?x a tourisme:Emplacement
}

Résultat

<result>

<binding name='x'><uri>http://www.polytech.semantique/tourisme#Cannes</uri></binding>
</result>

4. Les règles Rule :

L'idée des ces règles, c'est implémenter un système de recommandation, par exemple si un vol de type loisir à pour destination une ville X, le programme lui recommande des monuments à visiter dans cette ville et si un vol de type affair à pour destination une ville X, le programme lui
recommande des hotels dans cette ville. Nous l'avons aussi implémenter pour recommander des Agences de location où des Hôtels dans la ville de destination.

Règle Rule :

```
<rule>
<body>
<![CDATA[
prefix tourisme: <http://www.polytech.semantique/tourisme#>
        construct {
                ?x tourisme:recommenderMonument ?y
        } where {
                ?x rdf:type tourisme:VolLoisir
                ?x tourisme:destination ?place
                ?y tourisme:dans ?place
                ?y rdf:type tourisme:Monument
        }
]]>
</body>
</rule>
```

Donnée d'entrée :

```
tourisme:Vol6 a tourisme:VolLoisir ;
dc:identifier "6" ;
tourisme:depart tourisme:Paris ;
tourisme:destination tourisme:Nice .
dbpediaresource:
Carnavalet_Museum
a tourisme:Monument ;
tourisme:dans tourisme:Nice ;
tourisme:description "Le musée Carnavalet est le musée ...." ;
tourisme:imageUrl tourisme:IMAGE_Monument_Carnavalet_Museum ;
tourisme:localisation "48.8574 2.36214" ;
tourisme:siteWeb tourisme:URL_Monument_Carnavalet_Museum .
```

Requête Sparql :

```
PREFIX tourisme: <http://www.polytech.semantique/tourisme#>
        select * where {
        ?x tourisme:recommenderMonument ?y
```

}

Résultat :

<result>
<binding name='x'><uri>http://www.polytech.semantique/tourisme#Vol6</uri></binding>
<binding name='y'><uri>http://dbpedia.org/resource/Carnavalet_Museum</uri></binding>
</result>

IV. Déploiement & Exposition du modèle RDF Par SPARQL Endpoint (Corese)

Une fois nous avons le modèle RDF , nous avons charger ces données avec notre ontolgie et les regeles d'inference dans Corese Server. Nous avons trouvé des difficultés de déploiement dans corese server, que nous avons surmonté . Ci dessous les difficultés rencontrée.

- Les difficultés rencontrés

1. Corese server n'applique pas les règles Rules, pour résoudre ce problème, M. CORBY nous a conseillé d'exécuter les règles dans la GUI Corese (ou bien par programme) et sauvegarder le contenu du graphe kg:rule dans un fichier .rdf et le charger.

La requete :

construct
from kg:rule
where { ?x ?p ?y }

2. Problème d'encodage : DBpedia retourne des données encoder en UTF-8, et corese server utilise ANSI, ce qu'a permet d'afficher des caractères non compréhensible, et pour résoudre ce problème, nous avons utilisé l'option "*-Dfile.encoding="UTF-8*" pour obligé le serveur à utilisé cet encodage.

Ayant les données RDF et corrigeant tous ces problèmes avec Corese server , nous avons pu déployé notre ontologie, nos données et les règles en utilisant la commande :

java -Dfile.encoding="UTF-8" -jar corese-server.jar -load "tourisme.nt;tourisme.owl;inference.rdf" -o

Explication:

- tourimse.nt : Les données RDF formant N-Triple.
- tourimse.owl : notre ontologie OWL.
- inference.rdf: Les inférences realisées par Corese en se basant sur les règles passées. Il représente le contenu du graphe Kg:rule.

V. Exploitation des Connaissances (Client Web)

Après le déploiement du modèle dans Corese qui sera notre SPARQL Endpint , nous avons développé une application web en utilisiant Node.js.
Cette application peremet de mettre en valeur nos donnée RDF, et de montrer l'intérêt à développer notre application . Cette dernière assurerar un bon voyage pour nos utilisateur en leur fournissant des informations utiles sur les monuments ou les restaurants à visiter.

Elle permet aussi de faire des **requêtes en temps réel sur DBpedia** pour la récupération d'information une ville non existante dans notre base de connaissance.
Cette fonctionnalité utilise le syntaxe service de SPARQL pour requêter DBpedia. La requête est:

```
SELECT  * WHERE {

{service <http://dbpedia.org/sparql>
 { select * where
        {
?x rdf:type dbo:Museum;dbo:abstract ?description;dbo:thumbnail
?imageLink;dbpprop:pushpinMap 'Paris'@en;dbpprop:website ?website;grs:point
?localisation.
FILTER(langMatches(lang(?description), 'FR'))
       }
    }
}
UNION

 {service <http://dbpedia.org/sparql>
 {
  select * where
    {
?x rdf:type dbo:Museum;dbo:abstract ?description;
dbo:thumbnail ?imageLink;dbo:location 'Paris'@en;
dbpprop:website ?website;grs:point ?localisation.
FILTER(langMatches(lang(?description), 'FR'))
    }
   }
 }
}
```

L'application Web est fournie dans le dossier livré. Elle est nommée “Guide_WS_client” . Elle est lancer avec les commandes suivantes:

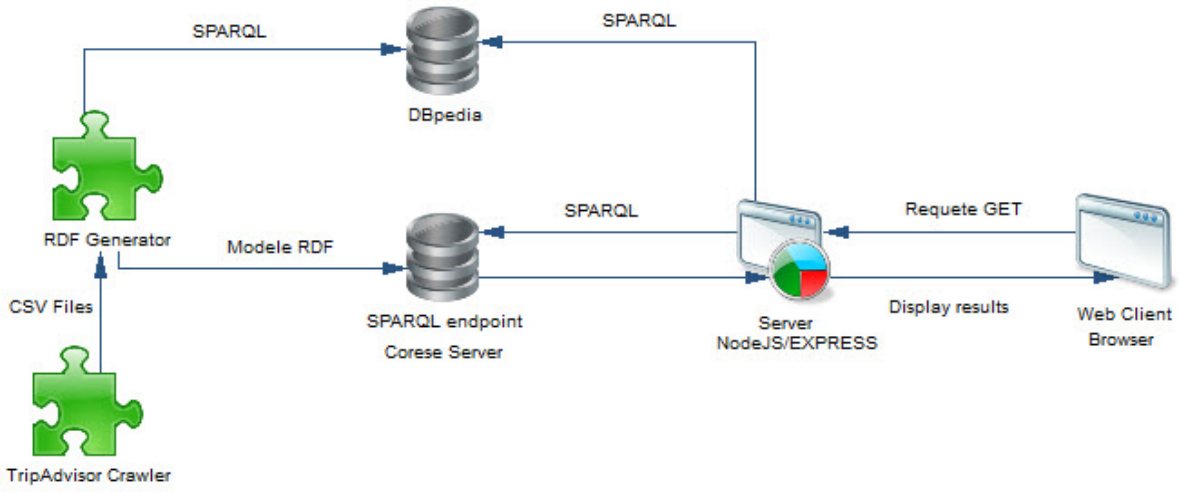
1. npm install
2. node server.js
3. Vous pouvez vous connecter sur l’adress : http://localhost:8081

VII. L'architecture de l'application

Notre application se décompose en deux parties :

❏ une partie Backend exposant les données sémantiques RDF avec un SPARQL Endpoint et stockées dans une Triple Store (Corese). Le données peuvent être récupère en utilisant des requêtes SPARQL à distance (HTTP).

❏ une partie Front End: affiche les connaissances extraites à partir des requêtes effectuées sur le SPARQL EndPoint dans des pages web.



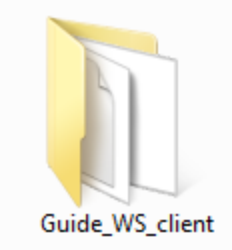
IX. Comment tester l'application facilement

1. Décompresser le fichier du projet
2. Copier la dernière version de corese serveur dans le dossier "**corese**"

3. Lancer la commande : "*java -Dfile.encoding="UTF-8" -jar corese-server.jar -load "tourisme.nt;tourisme.owl;inference.rdf" -o*" pour charger tous les fichiers necessaires

```
D:\WebSem\Guide_WS\Web_S-mantique_Project\corese>java -Dfile.encoding="UTF-8" -jar corese-server.jar -load "tourisme.nt;tourisme.owl;inference.rdf" -o
Server: tourisme.nt;tourisme.owl;inference.rdf
org.apache.commons.vfs.VfsLog.info(VfsLog.java:122) Using "C:\Users\BENATH~1\AppData\Local\Temp\vfs_cache" as temporary files store.
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.extractResourceDir(EmbeddedJettyServer.java:181) Overwritting directory webapp in file:///C:/Use
al/Temp/BENATHMANE
org.mortbay.log.Slf4jLog.info(Slf4jLog.java:67) Logging to org.slf4j.impl.Log4jLoggerAdapter(org.mortbay.log) via org.mortbay.log.Slf4jLog
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:119) ----------------------------------------
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:121) Corese/KGRAM endpoint started on http://localhost:8080/
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:122) ----------------------------------------
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:131) ----------------------------------------
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:132) Corese/KGRAM webapp UI started on http://localhost:8080
fr.inria.edelweiss.kgramserver.webservice.EmbeddedJettyServer.main(EmbeddedJettyServer.java:133) ----------------------------------------
```

4. Naviguer avec l'invite de commandes/Terminal vers le dossier "**Guide_WS_client**"



5. Lancer la commande "**npm install**" pour installer les modules nécessaires pour NodeJS, dans notre cas nous avons utilisé : express version 3.4.8 et request version 2.36.0, en cas de problème vous les télécharger manuellement en tapant la commande : npm install express ou/et request.
6. Dans le méme dossier, lancer la commande "node server.js"

```
BENATHMANE@BENATHMANE-PC /D/WebSem/Guide_WS/Web_S-mantique_Project/Guide_WS_clie
nt (master)
$ node server.js
```

7. dans votre navigateur, ouvrez : http://localhost:8081/ et bonne navigation :)