
Projet Road Trip

Web Sémantique X Ingénierie des connaissances

Etudiantes:

Lynda Attouche
Nolwenn Peyratout

Contents

1	Overview	2
2	Modélisation	2
3	SHACL	3
4	Population	4
4.1	Construire un graphe de connaissances à partir de données non structurées (textes) .	4
4.2	Data lifing	5
4.3	Population manuelle	7
5	Alignement et liage	8
6	Exploitation du graphe de connaissances	8
7	Interface	12
8	Conclusion	13

1 Overview

Pour notre projet nous souhaitons pouvoir proposer un road trip à un utilisateur en fonction du pays qu'il souhaite visiter avec un certain nombre d'activités à réaliser. Pour ce faire nous allons utiliser différentes techniques afin de pouvoir modéliser notre ontologie en RDF et la peupler grâce à csv2rdf, des sparql micro-service ou encore du Named Entity Recognition.

2 Modélisation

Le modèle ontologique de notre application "**Road Trip**" est conçu pour représenter de manière exhaustive les subtilités de la gestion d'un road trip, en couvrant divers aspects allant des détails de l'itinéraire aux profils d'utilisateurs.

La classe centrale est le "RoadTrip", définie par une *owl:Class* représente un voyage avec ses différentes propriétés. Ces dernières sont définies comme des *owl:ObjectProperty* ou *owl:DataProperty*. Elles couvrent les informations nécessaires au voyage. En l'occurrence, le pays du road trip :*in*, *hasItinerary* indiquant l'itinéraire prévu au voyage, une date de début et de fin (*startingDate*, *endingDate*), les participants ou les voyageurs intéressés ou ayant déjà effectués le voyage. Aussi, pour notre application, il est agréable d'avoir des avis sur les voyages proposés, c'est donc pour cela que des reviews sont associées au road trip (:*hasReview*).

Pour répondre à des préférences spécifiques, deux types de voyages routiers spécialisés sont introduits : "**RuraleRoadTrip**" et "**UrbanRoadTrip**". Le premier met l'accent sur les activités rurales se trouvant dans nos datasets (les lacs, montagnes,..etc), tandis que le second se concentre sur les expériences urbaines (musée, monuments en ville,..etc). Ces voyages spécialisés héritent des propriétés de la classe générale de "RoadTrip", ce qui fournit un cadre flexible permettant d'adapter les voyages à différentes préférences. Cette sous-classe de voyage a été modélisée comme étant l'intersection un road trip caractérisée par la contrainte selon laquelle celui-ci doit contenir un minimum de **3 activités** rurales/urbaines.

Afin de modéliser les pays sans utiliser de vocabulaire existant, le modèle intègre une classe "**Country**". Elle regroupe les classiques informations mais essentielles à connaître lors d'un voyage. En particulier, les propriétés spécifiques aux pays comprennent :*hasCountryName*, *hasCapital*, *hasCurrency*, *hasLanguage* et *requiresVaccine*. Cela permet aux utilisateurs de planifier des voyages en tenant compte de pays spécifiques et de leurs caractéristiques uniques.

L'une des informations et propriété fondamentale des road trips est l'**itinéraire**. Celui ci indique le parcours à suivre lors du voyage. Nous avons décidé de le modéliser comme étant un ensemble de trajets contenant des activités touristiques à faire sur tout le long. L'itinéraire a été défini comme une *owl:Class* avec une restriction sur le nombre de trajets. En effet, un itinéraire a au moins 2 trajets. De plus, il comprend des propriétés telles que :*hasStartingPoint*, *hasEndingPoint* et *hasRoute*, qui fournissent une représentation structurée du voyage planifié.

Les **trajets** constituant un itinéraire sont pareillement représentés par une *owl:class* avec une restriction sur le nombre d'activités. En effet, dans notre cas, un trajet ou comme appelé dans l'ontologie "Route" est un trajet entre 2 activités et pas plus. Cette classe relie le tout, en incorporant des propriétés telles que *hasActivity*, *hasRuralActivity*, *hasUrbanActivity*, *hasDistance*, *has-*

Time, *:hasTransport*, *:origin*, and *:destination*. Cela permet d'encapsuler l'ensemble de l'itinéraire, en tenant compte des différents types d'activités, des distances et des modes de transport.

Les **activités**, quant à elles, servent à décrire les diverses attractions touristiques suggérées dans l'itinéraire du road trip. Définie par une classe *activity*, elle saisit divers aspects *hasActivityName*, *hasCountry*, *hasLocation*, *hasBudget*, *hasRating* et *id*. De plus, les activités sont en outre classées en "**RuralActivity**" et "**UrbanActivity**", ce qui permet d'établir des distinctions en fonction de la nature des activités concernées.

Le modèle comprend également une représentation des **utilisateurs**, classés en tant qu'individus, membres d'un groupe ou associés à une agence de voyage. Les propriétés des utilisateurs comprennent *hasCompleteName*, *hasVaccinationStatus*, *hasPassportExpirationDate*, et *userReview*, fournissant une base pour la planification de voyages personnalisés basée sur les profils des utilisateurs.

De plus, un thésaurus qui représente et définit les différents **moyens de transport** utilisés dans le contexte de votre projet a été défini. Il comprends 3 niveaux hiérarchiques, un **TravelMode** représentant l'essence du thésaurus ayant deux top concepts *:PublicTransport* et *:PrivateTransport*. Chacun d'entre eux incluent des moyens de transport relatifs au type, eg. comme transport privé nous retrouvons à un niveau plus bas que ce dernier la voiture.

3 SHACL

En ce qui concerne les contraintes en SHACL, nous avons défini différentes contraintes sur un ensemble de classe pour garantir une bonne intégrité des données. Pour bien définir la classe *Roadtrip*, nous nous assurons que le roadtrip se fait dans au moins un pays, nous pourrions avoir un roadtrip se faisant dans deux pays différents. De plus, nous assurons que le roadtrip a un unique itinéraire, une date de début et de fin. Nous imposons aussi des contraintes sur la propriétés des participants. En effet, un roadtrip doit au moins avoir un participant de type *User*, donc soit une personne individuelle, un groupe ou toute une agence. Avec ces participants, une autre contrainte est défini. Pour que le roadtrip soit valide au niveau des participants, la date d'expiration de leur passeport devrait être inférieure à la date de fin du road trip et cela en utilisant *shacl:lessThan*. Pareillement, nous vérifions que les participants aient le status de vaccination requis par le pays du roadtrip (le pays d'entrée) et cela a été fait avec une contrainte sur la propriété *:requiresVaccine* et en vérifiant que les vaccins requis par un pays sont inclus dans la liste des vaccins des participants en utilisant *shacl:in*.

Pour une bonne définition du road trip nous devons assurer une bonne définition de l'itinéraire et activités. Pour le premier, nous assurons que la classe itinéraire a un point de départ et d'arrivée, qui sont tous deux des activités. De plus, un itinéraire comme nous l'avons modélisé doit contenir au moins deux trajets, donc une vérification est nécessaire.

Concernant ces trajets, nous définissons des contraintes qui assurent qu'un trajet doit contenir deux activités, ni plus ni moins et que ces activités sont soit rurales ou urbaines (en utilisant *shacl:in*). Un trajet devrait avoir des propriétés essentielles comme une distance, la période de trajet, et le mode transport que celui-ci implique.

Les activités contenues dans un trajet doivent avoir un unique nom, pays et localisation. Aussi, dans le cas où un budget est valable (si l'activité n'est pas gratuite), il est unique.

Pour la localisation, elle doit être bien défini avec une unique adresse, longitude, et latitude.

Comme nous avons modélisé la classe Country de zéro, nous devons assurer la bonne intégrité de nos données sur les pays. Pour se faire, nous assurons que le pays a un unique nom, capitale et devise. Nous assurons aussi qu'une langue est associée à cette classe. Nous ajoutons une contrainte sur les vaccins requis dans le pays, s'ils existent ils devraient être de type string. Dans ce cas, nous n'avons pas une contrainte sur le nombre de vaccins ou autre car un pays peut ne pas avoir des vaccins pour entrer sur le territoire comme il peut avoir plusieurs.

Pour chacune des contraintes sur le nombre et cardinalité des propriétés, nous avons utilisé `shacl:minCount` et `shacl:maxCount`.

4 Population

4.1 Construire un graphe de connaissances à partir de données non structurées (textes)

Pour réaliser un graphe de connaissances à partir de données non structurées nous avons souhaité réaliser du Named Entity Recognition sur des textes de blog.

Pour ce faire nous avons commencer par trouver un site internet, un blog, qui propose des articles sur des voyages dans différents pays. Nous avons donc pu scraper ces articles grâce à la librairie python scrapy afin de ne récupérer que les paragraphes de certaines pages du blog que nous avons choisis. Nous avons choisis des pages de Nouvelle Zélande, d'Espagne ou encore de Grèce car ce sont des pays où nous avons pas beaucoup d'activités, du data lifting.

A la suite de ce scraping, nous avons utilisé spacy pour réaliser du NER sur chaque paragraphe. Nous allons récupérer uniquement des entités du type LOC qui contiennent un mot d'une liste de mot définis. Nous ne récupérons que des entités de ce type car elles contiennent les activités que nous cherchons. Pour ce qui est de la liste de mot, nous nous sommes rendues compte que spacy pouvait nous renvoyer des activités erronées comme Africa. Ainsi nous avons créé une liste avec des mots comme lake, mount, crater ..., et nous vérifions si l'entité renvoyée comprend un mot de cette liste.

Lorsque notre entité correspond à tout ces critères nous pouvons l'ajouter à notre graphe, pour ce faire nous utilisons rdflib. Cette librairie nous permet de construire à la main un graphe. Nous savons que toutes les activités que nous récupérons sont de type activité et qu'elles sont liées au pays de l'article. Ainsi nous pouvons définir deux triplets, le premier est activité est de type Activity et le second est activité est dans le pays Country. Nous avons aussi fait du liage de données dans ce même temps, mais nous expliquons cela dans la section 5. Nous pouvons par la suite exporter ce graphe afin de l'ajouter à notre population, il se trouve dans le dossier ontology/NER.ttl.

4.2 Data lifing

La population du dataset par la méthode du Data lifting a été faite en utilisant des données extraites de csv liées aux activités et un autre aux pays. De plus, nous avons créé un micro service permettant de faire des requêtes à une api (Google Maps).

- **CSV à rdf** Nous avons eu à notre disposition, un premier fichier csv ¹ comportant des activités (leur nom), leur localisation ainsi que des images représentatives.

Nous avons converti ce csv grâce à l'outil présenté en cours csv2rdf ² et en faisant en sorte de respecter l'ontologie que nous avons défini. En particulier, chaque activité a été défini comme class:Activity avec un nom et un pays. Ce dernier est aussi défini comme une URI, comme c'est le cas dans l'ontologie.

Example. 1.

```
<http://example.org/activities/MachuPicchu> a <http://example.org/roadtrip/Activity>;  
<http://example.org/roadtrip/hasActivityName> "Machu Picchu";  
<http://example.org/roadtrip/hasCountry> <http://example.org/activities/MachuPicchu#country> .  
  
<http://example.org/activities/MachuPicchu#country> a <http://example.org/roadtrip/Country>;  
<http://example.org/roadtrip/hasCountryName> "Peru" .
```

Pareillement, nous avons enrichi notre base de données à l'aide d'un fichier csv contenant les pays, leur indicatif, capitale, devise, région et leur fuseau horaire. Nous avons preprocesser ce fichier en ne gardant que les pays pour lesquels nous avons des activités. Puis, en le convertissant nous avons décidé de ne sélectionner seulement les colonnes pays, capitale et devise comme décrit dans l'ontologie. Ci-dessous un exemple du résultat obtenu,

Example. 2.

```
<http://example.org/countries/Peru> a <http://example.org/roadtrip/Country>;  
<http://example.org/roadtrip/hasCountryCapital> "Lima";  
<http://example.org/roadtrip/hasCountryName> "Peru";  
<http://example.org/roadtrip/hasCurrency> "PEN" .
```

¹Activity csv: <https://github.com/nghia-t-nguyen/travel-bot-project/blob/main/Travel-Destinations.csv>

²csv2rdf: <https://github.com/Swirrl/csv2rdf/>

- **Micro-service**

Après avoir obtenu les différentes activités qui se trouvent dans un pays, nous voulons d'un premier lieu récupérer plus d'informations concernant ces attractions. En l'occurrence, leur exacte localisation, le type d'activité, le budget, et l'ouverture en temps réel. Pour se faire, nous avons choisi de travailler avec l'API Google Maps ³. Au début, le choix de recourir à l'API de Google s'avérait prometteur, étant donné que celle-ci renfermait une quantité important de données qui satisfaisait pleinement nos attentes, en plus d'une documentation détaillée.

Cependant, lors des tests effectués vers l'API, les résultats étaient concluants, ce qui n'était malheureusement pas le cas lors des tests avec la requête fédérée. Concrètement, lorsque nous chargeons les données du fichier d'activités en format turtle et effectuons des appels à l'API avec une requête SPARQL correctement structurée, nous sommes confrontés à une erreur, comme illustré dans dans la figure 1

```
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:72 [] - Workflow Context:
env-corese-1 | default-graph-uri : "<(http://localhost/service/resp-graph-65a5018d5f3979.91985995> )"^^dt:list
env-corese-1 | st:remoteHost : "172.19.0.4"
env-corese-1 | st:service : "http://ns.inria.fr/sparql-template/user"
env-corese-1 | request : "[org.eclipse.jetty.server.Request:Request(POST //corese:8081/sparql)@448a3a69]"^^dt:pointer
env-corese-1 | url : <http://corese:8081/sparql>
env-corese-1 | user query: true
env-corese-1 | level: PRIVATE
env-corese-1 |
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:73 [] - {st:count="[Map: size=1]"^^dt:map, st:date="[Map: size=
1]"^^dt:map, st:host="[Map: size=1]"^^dt:map, st:hostname="[Map: size=1]"^^dt:map}
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:74 [] - {"http://ns.inria.fr/sparql-template/user"=59}
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:76 [] - {"172.19.0.4"=59}
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.SPARQLRestAPI.updateTriplesDirect:881 [] - updateTriplesDirect
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.SPARQLRestAPI.updateTriplesDirect:884 [] - DROP SILENT GRAPH
env-corese-1 | <http://localhost/service/resp-graph-65a5018d5f3979.91985995>
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:72 [] - Workflow Context:
env-corese-1 | st:remoteHost : "172.19.0.4"
env-corese-1 | st:service : "http://ns.inria.fr/sparql-template/user"
env-corese-1 | request : "[org.eclipse.jetty.server.Request:Request(POST //corese:8081/sparql)@258b4ac8]"^^dt:pointer
env-corese-1 | url : <http://corese:8081/sparql>
env-corese-1 | user query: true
env-corese-1 | level: PRIVATE
env-corese-1 |
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:73 [] - {st:count="[Map: size=1]"^^dt:map, st:date="[Map: size=
1]"^^dt:map, st:host="[Map: size=1]"^^dt:map, st:hostname="[Map: size=1]"^^dt:map}
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:74 [] - {"http://ns.inria.fr/sparql-template/user"=60}
env-corese-1 | 2024-01-15 09:57:33 INFO webservice.EventManager.log:76 [] - {"172.19.0.4"=60}
env-sparql-micro-service-1 | 127.0.0.1 - - [15/Jan/2024:09:57:33 +0000] "POST /sparql-ms/src/sparqlms/service.php?
root_url=http://localhost/service&querymode=sparql&service=roadtrip/getInformation HTTP/1.1" 200 1401 "-" "Jersey/3.0.4 (HttpURLConnection 19)"
env-sparql-micro-service-1 | 172.19.0.1 - - [15/Jan/2024:09:57:32 +0000] "POST /service/roadtrip/getInformation HTTP/1.1" 200 1438 "-" "Jersey/3.0.4
(HttpURLConnection 19)"
env-sparql-micro-service-1 | [Mon Jan 15 09:57:42.254155 2024] [mpm_prefork:error] [pid 1] AH00161: server reached MaxRequestWorkers setting, consider
raising the MaxRequestWorkers setting
```

Figure 1: Erreur Corese et micro-service 1

Après avoir passé un temps considérable à tenter de comprendre et résoudre le problème que ce soit de manière autonome ou avec l'aide du professeur, nous n'avons malheureusement pas réussi à lancer les requêtes sur l'intégralité de données comprenant 249 activités.

La seule solution que nous avons envisagée pour surmonter cette difficulté a été de lancer des requêtes sur des sous-ensembles du dataset qui contiennent 30 activités au maximum avant de les regrouper manuellement dans un unique fichier.

Nous avons également l'intention de récupérer par le biais de la même API les itinéraires d'un roadtrip en spécifiant un point de départ, un point d'arrivée, des points de passage

³Google Maps API: <https://developers.google.com/maps/>

(stops) et un mode de transport (ex. Driving). La requête fonctionne correctement lorsqu'elle est testée avec POSTMAN. Or, quand celle-ci a été testée avec Corese, une autre erreur est survenue et est illustrée dans 2

```
2024-01-20 23:18:36 ["uuepvo98gla0j"] service NOTICE: Web API query string: https://maps.googleapis.com/maps/api/directions/json?
destination=Paris&mode=car&origin=Nice& waypoints=optimize:true%7Clyon&key=AIzaSyCLxr9s4hRmmunl_APc7RRt8B7KuS3m1n0
2024-01-20 23:18:36 ["uuepvo98gla0j"] Utils WARNING: ML\JsonLD\Exception\JsonLdException: Syntax error, malformed JSON. in /var/www/sparql-ms/vendor/ml/json-
ld/Processor.php:211
Stack trace:
#0 /var/www/sparql-ms/vendor/ml/json-ld/FileGetContentsLoader.php(143): ML\JsonLD\Processor::parse()
#1 /var/www/sparql-ms/vendor/ml/json-ld/JsonLD.php(142): ML\JsonLD\FileGetContentsLoader->loadDocument()
#2 /var/www/sparql-ms/vendor/ml/json-ld/JsonLD.php(411): ML\JsonLD\JsonLD::expand()
#3 /var/www/sparql-ms/src/common/Utils.php(224): ML\JsonLD\JsonLD::toRdf()
#4 /var/www/sparql-ms/src/sparqlms/service.php(244): frmichel\sparqlms\common\Utils::translateJsonToNQuads()
#5 /var/www/sparql-ms/src/sparqlms/service.php(171): frmichel\sparqlms\queryWebAPIAndGenerateTriples()
#6 {main}
2024-01-20 23:18:36 ["uuepvo98gla0j"] service ERROR: Exception: Cannot query the Web API or transform its response to JSON-LD. in /var/www/sparql-
ms/src/common/Utils.php:237
Stack trace:
#0 /var/www/sparql-ms/src/sparqlms/service.php(244): frmichel\sparqlms\common\Utils::translateJsonToNQuads()
#1 /var/www/sparql-ms/src/sparqlms/service.php(171): frmichel\sparqlms\queryWebAPIAndGenerateTriples()
#2 {main}
```

Figure 2: Erreur Corese et micro-service 2

L'erreur est interne à Corese, car lors de la requête, la bibliothèque JSON-LD utilisée génère une erreur à la lecture de la réponse de l'API. Par conséquent, nous n'avons pas pu récupérer directement les résultats via l'API en nous basant sur les retours avec le micro-service.

Afin d'obtenir tout de même des résultats pour nos itinéraires et disposer d'une base de données complètes, nous avons manuellement fait des requête (via le navigateur) et extrait les informations nécessaires. Compte tenu du temps nécessaire pour ce processus, nous avons limité le nombre de road trips à 2 et choisi deux pays : la France et les États-Unis.

En résumé, bien que des erreurs indépendantes de notre contrôle aient été rencontrées, nous avons pris des mesures pour collecter les données à partir de l'API, que ce soit en utilisant des sous-ensembles ou en effectuant la collecte manuellement. De plus, nous avons procédé à l'ajout manuel des propriétés et classes liées aux types d'activité et aux road trips (ruraux et urbains).

4.3 Population manuelle

Pour compléter notre ontologie, nous l'avons aussi populer manuellement afin d'y ajouter des utilisateurs mais aussi définir le type de chaque activité.

Nous souhaitions développer le fait qu'un road trip peut être réalisé par une personne individuelle ou par un groupe, mais aussi qu'une agence de voyage puisse consulter les différents road trips. Ainsi nous avons pu ajouter différents utilisateurs qui représentent ces différents cas, ils sont dans le fichier user.ttl.

Pour ce qui est du type des activités nous avons dû les renseigner manuellement. Ainsi chaque activité est de type RuralActivity ou UrbanActivity pour définir si elle est plutôt urbaine ou campagnarde. Pour ce qui est des activités récoltés en faisant du NER elles sont d'office classée en RuralActivity puisque nous les avons choisies comme cela.

5 Alignement et liage

Nous avons aligné notre ontologie avec wikidata, foaf, dbpedia et schema.org. Pour réaliser cela nous avons manuellement cherché à quels autres prédicats nos prédicats pouvaient être rattachés. Ainsi nous avons pu trouver quelques alignements dans les vocabulaires précédents. De plus nous avons aussi réalisé de l'alignement sur les concepts skos, avec skos:exactMatch. Nous avons pu l'utiliser dans notre thésaurus de nos moyens de locomotions dans transport.ttl.

Nous n'avons pas effectué d'alignement sur toute notre ontologie car cela nous prenait trop de temps à le faire à la main. Ainsi nous avons souhaité réaliser les autres parties du projet en priorité.

Pour ce qui est du liage des données nous avons liés une partie de de nos données grâce à DBpedia Spotlight. Cette librairie nous permet d'envoyer du texte et de trouver une URI de dbpedia qui correspond à certains éléments du texte. Nous avons réalisé ce liage en même temps que nous avons réalisé du Named Entity Recognition. Ainsi lorsque notre script reconnaît une activité nous cherchons l'URI dbpedia associée et nous créons un noeud sur le graph avec comme sujet l'activité, comme prédicat owl:sameAs et comme valeur l'URI de dbpedia. Pour ce qui est du choix des paramètres pour définir la confiance et le support, nous avons choisi respectivement 0,4 et 20. Ce choix s'est fait en réalisant plusieurs essais avec dbpedia Spotlight.

Lors du liage des données nous avons eu quelques soucis puisque dbpedia spotlight ne reconnaissait pas facilement tout type d'activités. En effet tous les lacs sont mal reconnus, dbpedia renvoie comme URI celle de lac et non celle du lac précisément. Pour contrer ce problème nous avons vérifié à la main toutes les URI que dbpedia spotlight nous renvoyait pour enlever les mauvaises URI.

6 Exploitation du graphe de connaissances

Pour exploiter notre graph de connaissances nous avons réalisé différents requêtes SPARQL afin de montrer le potentiel de notre ontologie.

En premier nous vérifions si un participant d'un road trip a les vaccins nécessaires pour entrer dans le pays :

```
PREFIX rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns
```

```
PREFIX ns1: http://example.org/roadtrip
```

```
ASK
```

```
WHERE {
```

```
?roadTrip a ns1:RoadTrip ;  
    ns1:hasItinerary ?itinerary ;  
    ns1:participants ?participant ;  
    ns1:in ?country .
```

```
?participant a ns1:Individual ;  
    ns1:hasVaccinationStatus ?vaccinationStatus .
```

```
?country ns1:requiresVaccine ?requiredVaccine .
```

```
FILTER (CONTAINS(str(?vaccinationStatus), str(?requiredVaccine)))
}
```

En second nous obtenons des activités avec une note supérieure à un certain seuil :

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ns1: <http://example.org/roadtrip#>

SELECT ?activity ?activityName ?rating
WHERE {
    ?activity rdf:type ns1:Activity .
    ?activity ns1:hasActivityName ?activityName .
    ?activity ns1:hasRating ?rating .
    FILTER(?rating > 4.5)
}
```

En troisième, c'est la requête que nous utilisons dans l'interface. Elle nous permet d'obtenir les détails des activités d'un roadtrip précis :

```
PREFIX act: <http://example.org/activity#>
PREFIX schema: <http://example.org/roadtrip#>
PREFIX country: <http://example.org/country#>
PREFIX transport: <http://example.org/transport#>

SELECT ?itinerary ?startPointItName ?endPointItName ?originName ?
    originRating ?originAddress ?destName ?destRating ?destAddress
    ?distance
WHERE {
    ?roadtrip a schema:RoadTrip;
        a ?type;
        schema:in ?country;
        schema:hasItinerary ?itinerary.
    ?itinerary schema:hasRoute ?route;
    schema:hasStartingPoint ?startPointIt;
    schema:hasEndingPoint ?endPointIt.

    ?route schema:hasTransport ?transport;
    schema:origin ?origin;
    schema:destination ?dest;
    schema:hasDistance ?distance.

    ?startPointIt schema:hasActivityName ?startPointItName.

    ?endPointIt schema:hasActivityName ?endPointItName.

    ?origin schema:hasActivityName ?originName;
```

```

OPTIONAL{?origin schema:hasRating ?originRating;}
OPTIONAL{?origin schema:hasLocation [schema:hasAddress ?
    originAddress].}

?dest schema:hasActivityName ?destName;
OPTIONAL{?dest schema:hasRating ?destRating;}
OPTIONAL{?dest schema:hasLocation [schema:hasAddress ?destAddress
    ].}

?country ?hasCountryName ?countryName.
filter(?countryName="France")
filter(?type=schema:UrbanRoadTrip)
filter(?transport = transport:Car)
}

```

Pour ce qui est de la requête que nous utilisons lors de l'appel à notre micro service pour récupérer les informations des activités nous faisons celle ci :

```

PREFIX act: <http://example.org/activities#>
PREFIX ex: <http://ns.inria.fr/sparql-micro-service/api#>
PREFIX : <http://example.org/roadtrip/>
PREFIX schema: <http://example.org/activity#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

```

CONSTRUCT {

```

[] a ?activityType;
:hasActivityName ?name;
:hasCountry ?country ;
:hasBudget ?budget ;
:hasLocation [a :Location;
:hasLongitude ?longitude;
:hasLatitude ?latitude;
:hasAddress ?address] ;
:hasRating ?rating;
:id ?id ;
owl:sameAs ?same .
}
where {
SELECT ?name ?country ?id ?latitude ?longitude ?rating ?open_now
    ?budget ?address ?activityType ?same {
[] schema:hasActivityName ?name;
    schema:hasCountry ?country;
    owl:sameAs ?same;
    a ?activityType.
    ?country :hasCountryName ?nameCountry

```

```

BIND (IRI (CONCAT ("http://localhost/service/roadtrip/getInformation
    ?place=", ?name)) as ?serviceURI)

```

```

SERVICE ?serviceURI{
    ?s ex:place_id ?id.
    OPTIONAL{ ?s ex:geometry ?geom.}
    OPTIONAL{?geom ex:location ?location.}
    OPTIONAL{?location ex:lat ?latitude.}
    OPTIONAL{?location ex:lng ?longitude.}
    OPTIONAL{ ?s ex:rating ?rating. }
    OPTIONAL{ ?s ex:opening_hours ?openH.}
    OPTIONAL{?openH ex:open_now ?open_now.}
    OPTIONAL{?s ex:maxprice ?budget.}
    OPTIONAL{?s ex:formatted_address ?address.}
}
filter CONTAINS(?address, ?nameCountry)
}
GROUP BY ?name}

```

Et finalement une requête qui nous permet d'afficher seulement les road trips de plus de 1000km :

```

PREFIX ex: <http://example.org/roadtrip#>

SELECT ?roadTrip (SUM(?distance) as ?totalDistance)
WHERE {
    ?roadTrip ex:hasItinerary ?itinerary.
    ?itinerary ex:hasRoute ?route.
    ?route ex:hasDistance ?distance.
}
GROUP BY ?roadTrip
HAVING (?totalDistance > 1000)

```

7 Interface

Pour ce qui est de l'interface nous avons voulu faire une interface web où un utilisateur pourra choisir le pays, ainsi que le type de roadtrip qu'il souhaite faire et cela lui affichera un itinéraire qui correspondra à ses attentes.

Pour faire cela nous avons fait une application grâce à Vue.js et corese server. Nous avons chargé nos données dans le serveur corèse en spécifiant que nous souhaitons faire de l'entaillement:

Example. 3.

```
java -jar corese-server-4.5.0.jar -su -e -l "./ontology/activitiesInfo.ttl" -l "./ontology/country.ttl" -l "./ontology/french\_itineraries.ttl" -l "./ontology/french\_roadtips.ttl" -l "./ontology/UnitedStates\_itineraries.ttl" -l "./ontology/UnitedStates\_roadtrips.ttl" -l "./ontology/roadTrip.ttl" -l "./ontology/transport.ttl" -l "./ontology/user.ttl"
```

Dans notre application nous pouvons appeler notre endpoint avec notre requête SPARQL, cet appel est de la forme: "http://localhost:8080/sparql?query=". A la suite de cela nous obtenons un résultat en html que nous parsons et stockons. Après cela nous pouvons les afficher en dessous de la barre de recherche.

Dans notre requête nous récupérons les activités de départs et d'arrivées ainsi que les notes de ces activités, les addresses mais aussi les distances entre chaque activités. Ces informations permettent de montrer à l'utilisateur des informations supplémentaires pour s'informer du voyage. Pour ce qui est des distances, nous n'affichons seulement que la distance totale du trajet. Nous pourrions faire de même pour la durée du trajet, mais nous n'avons pas souhaité l'afficher pour cette version du projet. Nous pouvons voir dans la figure 3 notre interface lorsque nous avons sélectionné un pays, un type de voyage ainsi que la situation de la personne qui souhaite faire ce voyage.

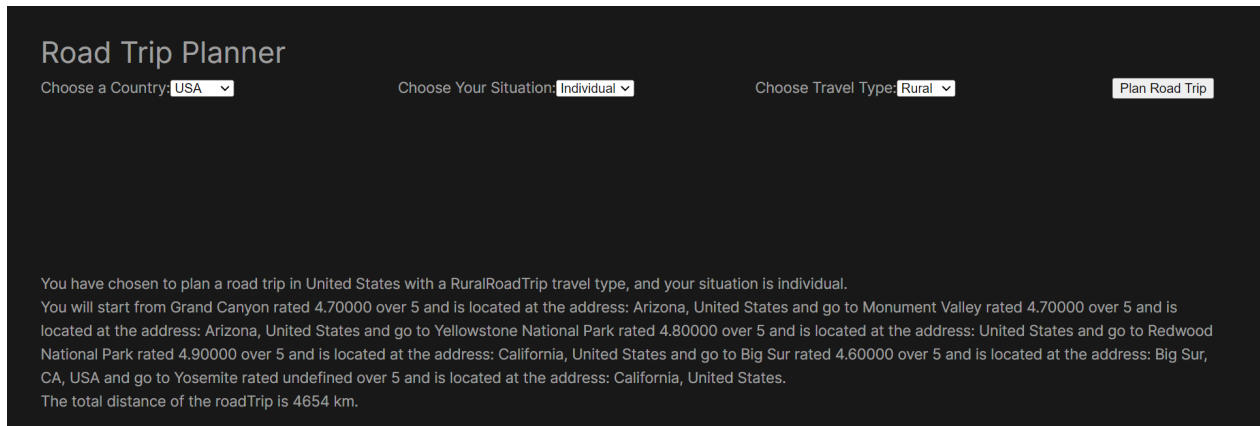


Figure 3: Interface web de notre application

Nous nous sommes rendu compte en lançant le serveur corèse que certaines fois les données renvoyées par le serveur étaient erronées, sans raison apparante. En effet il manque des '<' et donc nous ne pouvons plus parser correctement le résultat. Pour contrer cela, nous relançons le serveur jusqu'à ce que cela fonctionne.

8 Conclusion

Pour conclure, nous avons réussi à créer une ontologie complexe basée sur les road trips avec de nombreuses données issues de multiples sources et a pouvoir interragir avec celles-ci via une interface web qui récupère les informations de nos road trip et qui les traite afin de les afficher correctement pour un utilisateur.

Nous avons pu grâce à ce projet mettre en pratique différentes notions que nous avons vu comme du NER, utiliser des sparql micro service, traduire du CSV en RDF ou encore mettre en place des bases de web sémantiques avec du owl ou du SHACL. De plus nous avons pu lié notre ontologie ainsi que nos données à d'autres graphs du web comme dbpedia ou encore wikidata, afin de pouvoir rendre notre graph plus riche.