Ingénierie des connaissances
Web sémantique

Projet Final
Rapport

DIBE Sana
FANG Keyan

Cas d'usage

Notre objectif est d'aider les voyageurs qui cherchent à planifier un voyage culturel tout en respectant un budget restreint. Les voyageurs sont souvent confrontés au choix difficile entre dépenser plus d'argent pour un voyage plus luxueux et visiter moins d'attractions, ou dépenser moins d'argent pour plus de restrictions sur le voyage.

Notre solution consiste à combiner ces deux éléments en permettant aux utilisateurs de planifier un voyage culturel abordable tout en maximisant les attractions culturelles qu'ils peuvent visiter. Nous visons à rendre les voyages culturels accessibles à un public plus large en proposant une variété de destinations économiques pour les billets d'avion ainsi que des activités culturelles variées. De plus, nous souhaitons intégrer les hébergements ainsi que les restaurants dans notre application pour offrir une expérience de voyage complète et sans stress aux utilisateurs. Cela permettra aux voyageurs de planifier et de réserver tout leur voyage, y compris leur hébergement et leur nourriture, en un seul endroit, sans avoir à naviguer sur plusieurs sites ou à gérer des transactions multiples. Nous croyons que cela améliorera considérablement l'expérience utilisateur et renforcera notre engagement en faveur de voyages culturels abordables et accessibles.

Notre but est de rendre les voyages culturels accessibles à un public plus large, sans sacrifier la qualité et l'expérience de voyage. Nous espérons ainsi encourager les gens à explorer de nouvelles cultures et à découvrir l'héritage historique du monde.

Modélisation ontologique

Lors de la construction du modèle pour notre produit, nous avons pris en compte le cas d'utilisation spécifique de planification de voyages culturels pour les utilisateurs. Nous avons donc modélisé les vols et les voyages de manière séparée, en accordant une importance particulière au voyage culturel. Ce dernier inclut des éléments clés tels que les informations sur les voyageurs, les vols, les aéroports, l'hébergement, la ville culturelle et les sites culturels.

Nous avons également pris en compte la réalité du monde en classant les sites culturels en catégories telles que les sites archéologiques, les tombes, les monuments, etc.

Pour assurer la qualité des informations incluses dans notre modèle, nous avons utilisé des contraintes SHACL pour restreindre la modélisation des voyages. Par

exemple, pour chaque HistoricalTrip, il doit y avoir au moins un voyageur et exactement un vol.

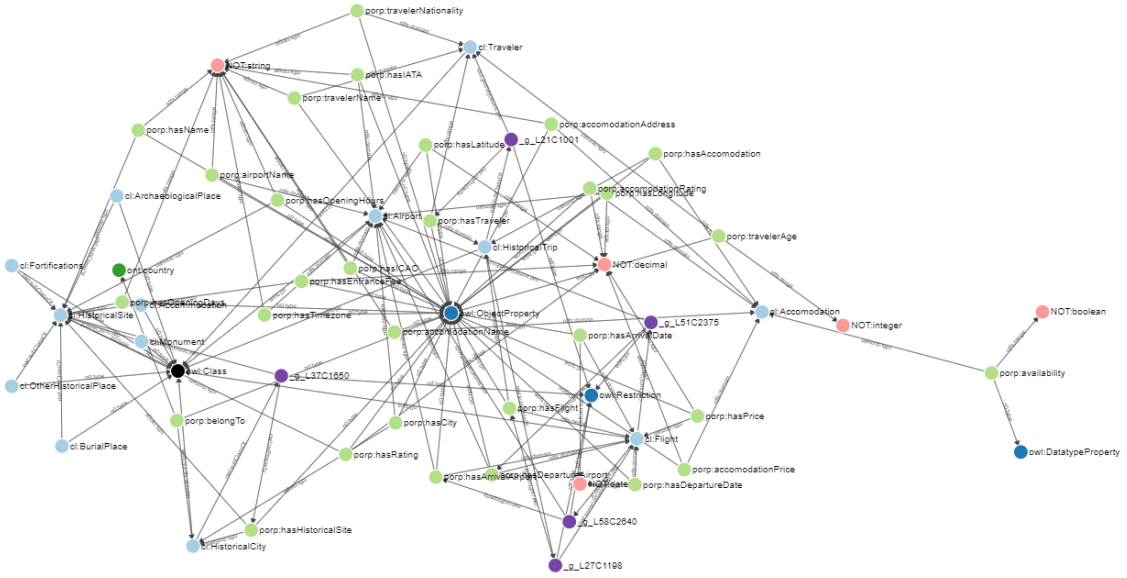

Figure 1. Ontologie de voyage historique

Nous avons mis en place une stratégie pour faciliter la planification des voyages historiques de nos clients en fournissant des informations efficaces et économiques sur les vols et les aéroports. Pour ce faire, nous avons utilisé OpenTripMap pour récupérer et normaliser les informations sur les 10 meilleures destinations historiques du monde: Paris, Florence, Tokyo, Milano, Washington, Edinburgh, Rome, Berlin, Londres, Moscou et Sao Paulo. Comme nous n'avons pas trouvé de CSV avec des informations complètes sur les sites historiques, même celui de l' UNICEF était incomplet, nous avons construit le nôtre en effectuant une collecte de données personnalisée. Cela va permettre aux voyageurs de comparer les options et de faire des choix éclairés.

Les informations sont obtenues grâce à des API de différentes compagnies aériennes et sont ensuite consolidées pour fournir une vue complète de chaque vol disponible. Chaque vol est défini par son origine et sa destination, ses attributs (aller simple ou aller-retour), l'heure de départ et le coût du billet. Avec l'aide de la technologie SHACL, nous veillons à ce que chaque vol soit défini de manière unique et que le prix du billet soit toujours raisonnable et abordable, compris entre 0 et 1500.

Les contraintes pour les données de notre application Web sont définies dans le fichier shaclConstraints.ttl pour les classes suivantes : HistoricalTrip, HistoricalCity, HistoricalSite, Flight et Traveler. Pour chaque classe, le code établit les propriétés requises et facultatives, telles que le nombre minimum et maximum de propriétés, le type de données, la longueur et la plage de valeurs, ainsi que d'autres conditions associées. Par exemple, "HistoricalTrip" doit avoir une propriété "hasFlight", tandis que "Traveler" doit avoir une propriété "travelerName" qui est une chaîne de caractères de longueur comprise entre 3 et 20 et une propriété "age" qui est un entier compris entre 0 et 200.
La classe "Flight" doit avoir une propriété "hasPrice" avec un minimum de 1 comptage, un maximum de 1 comptage et un type de données xsd:decimal.

Les contraintes sont utilisées pour garantir que les données correspondent à la forme définie dans l'ontologie.

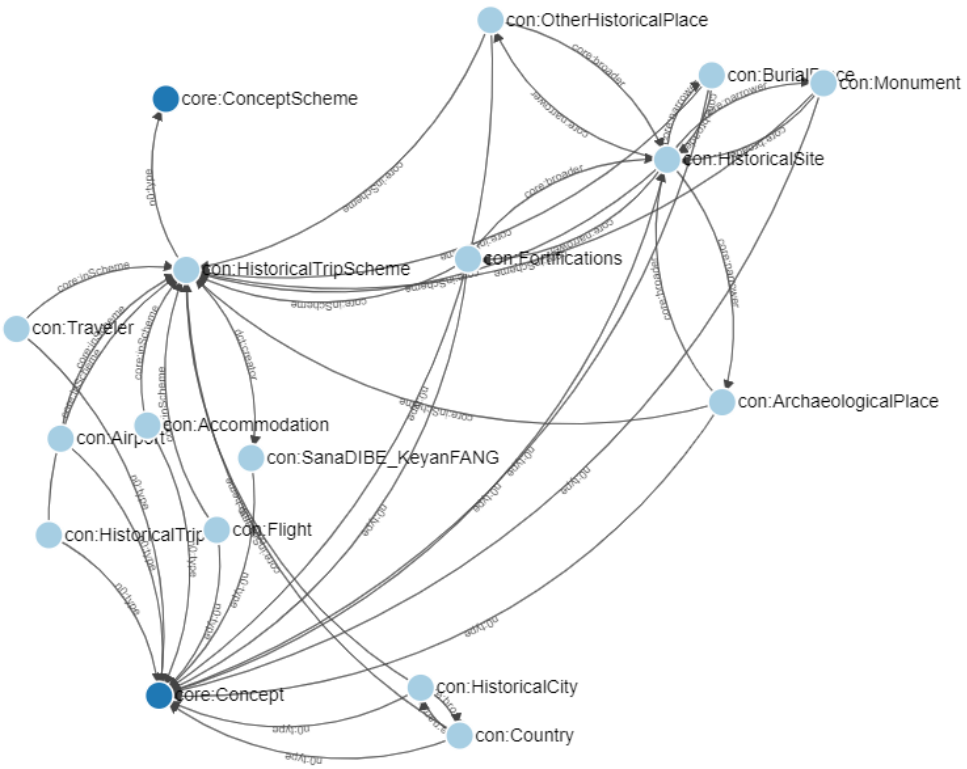

Figure 2. graphe de concept skos

Graphe de connaissances

Comme indiqué précédemment, notre graphe de connaissances comporte principalement les relations entre les voyageurs et les voyages, ainsi que les différents aspects du voyage, tels que les vols, les aéroports, l'hébergement et les sites culturels à visiter.

Nous pouvons améliorer notre ontologie "historicalTrip" en utilisant les données provenant de Wikipedia et de DBpedia. En effectuant une analyse de la section "histoire" de Wikipedia, nous sommes en mesure d'extraire des informations pertinentes concernant les individus associés à un site historique, puis en utilisant DBpedia Spotlight pour les annoter. Nous pouvons ajouter ces triplets à notre ontologie, en utilisant la propriété "hasRelatedPerson", pour la classe "historicalSite". De plus, nous pouvons également ajouter les noms et résumés de ces individus pour une meilleure compréhension.

Dans le cadre de ce projet, nous avons présenté un exemple avec le Mémorial de la Shoah. Nous avons débuté avec une brève description du mémorial enregistrée dans le fichier "Mémorial de la Shoah", puis en utilisant l'annotateur DBpedia Spotlight, nous avons réussi à extraire les URLs correspondantes depuis DBpedia et à identifier les personnalités associées au mémorial, telles que Léon Poliakov et François Hollande.

Un autre exemple concerne le Mémorial des Martyrs de la Déportation, où nous avons réussi à récupérer les informations sur Charles de Gaulle à partir du texte original. En plus de cela, nous pouvons également ajouter d'autres triplets tels que "hasFamousEvent" ou "onceKnownAs" à l'avenir.

```python
import rdflib
from rdflib import Namespace, Literal, URIRef
from rdflib.namespace import RDF

g = rdflib.Graph()
g.parse("monument.ttl", format="turtle")

# Create a namespace for the trip and its properties
trip = Namespace("http://www.historical-trip.org/class#")

# Loop over each monument
for monument, people in related_person.items():
    monument_uri = URIRef(f"http://www.historical-trip.org/class#{monument.replace(' ', '_')}")
    g.add((monument_uri, RDF.type, trip.HistoricalSite))

    # Loop over each person in the monument
    for person in people:
        person_name = person["hasPersonName"].replace(" ", "_")
        person_uri = URIRef(f"http://www.historical-trip.org/class#{person_name}")

        # Add the person to the graph
        g.add((person_uri, RDF.type, trip.Person))
        g.add((person_uri, trip.hasAbstract, Literal(person["hasAbstract"])))
        g.add((person_uri, trip.hasAbstract, Literal(person["hasPersonName"])))
        g.add((monument_uri, trip.hasPerson, person_uri))

# Write the enriched graph to a new file
g.serialize("enriched_monument_from_text", format="turtle")
```

Figure 3. codes pour lifter des personnages liés avec monuments

Comme vous pouver voir dans la figure ci-dessus extraite de notre notebook monument_history_text.ipynb, ce bout de code utilise la bibliothèque rdflib pour créer un graphe. Il commence par analyser notre fichier Turtle existant appelé "monument.ttl" dans le graphe; ce fichier était le résultat du csv2rdf lifting. Un namespace "trip" est défini pour les classes et les propriétés de l'ontologie. Ensuite, il parcourt chaque monument et les personnes associées dans le dictionnaire "related_person". Il crée des URIs pour chaque monument et personne et les ajoute au graphe en tant que nœuds avec les types RDF appropriés.
Des propriétés telles que "hasAbstract" et "hasPersonName" sont ajoutées aux nœuds personne en tant que littéraux. Le nœud monument est lié à son nœud personne associé avec la propriété "hasPerson". Enfin, le graphe enrichi est sérialisé et écrit dans un nouveau fichier "enriched_monument_from_text" au format Turtle.

Source des données

Nos données sont divisées en deux parties : descriptions des villes et des monuments culturels, et des vols vers différentes destinations. Par conséquent, nous avons adopté deux méthodes différentes pour obtenir respectivement deux parties de données, et finalement les intégrer dans notre graphe de connaissances.

CSV

Nous n'avons pas réussi à trouver un fichier csv convenable pour l'extraction de données sur les sites historiques via csv2rdf. Alors, nous avons opté pour l'API OpenTripMap pour obtenir les informations sur les villes et les monuments culturels pertinents, et avons créé un programme opentrip.ipynb pour extraire ces données. Les informations ont été converties en fichier csv pour une extraction plus facile. Nous avons également utilisé l'API Places de Google pour ajouter plus de richesse à nos données. Enfin, nous avons associé les données à des ressources web de données telles que DBPedia et Wikipedia pour créer une ontologie complète.

Pour améliorer les données de l'API Amadeus, nous avons choisi un fichier CSV provenant de openFlight qui offre de nombreuses possibilités d'enrichissement des données (aéroports, avions, vols, etc.). Pour notre utilisateur qui a des difficultés à retenir plus de 70 000 codes aéroports IATA, nous nous sommes concentrés sur les aéroports pour obtenir le nom de l'aéroport et la ville.

API et SPARQL micro-service

En ce qui concerne les informations des vols, nous avons obtenu les données à travers l'API via le microservice SPARQL. Nous avons sélectionné l'API de l'entreprise prestigieuse Amadeus pour les services touristiques et avons utilisé les informations relatives aux aéroports et à de nombreux vols dans le monde comme données structurées pour construire un graphe de connaissances.

Pour cela nous avons créé un service amadeus4dev qui récupère les vols les moins chers pour aller à une destination donnée et avec un prix maximal à ne pas dépasser.

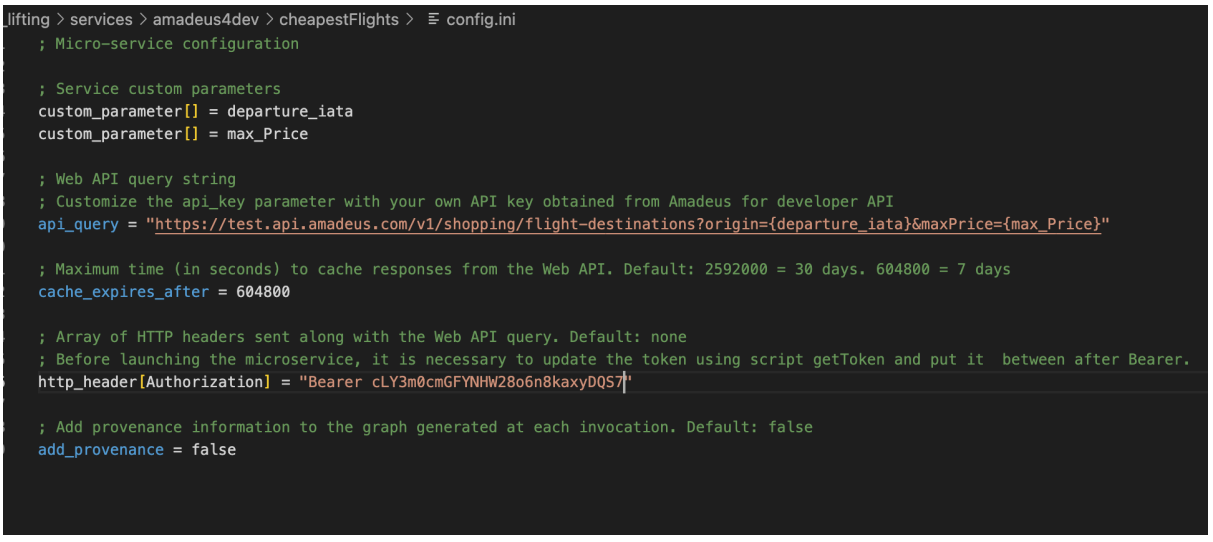

Figure 4. SPARQL micro-servicepour pour l'API Amadeus

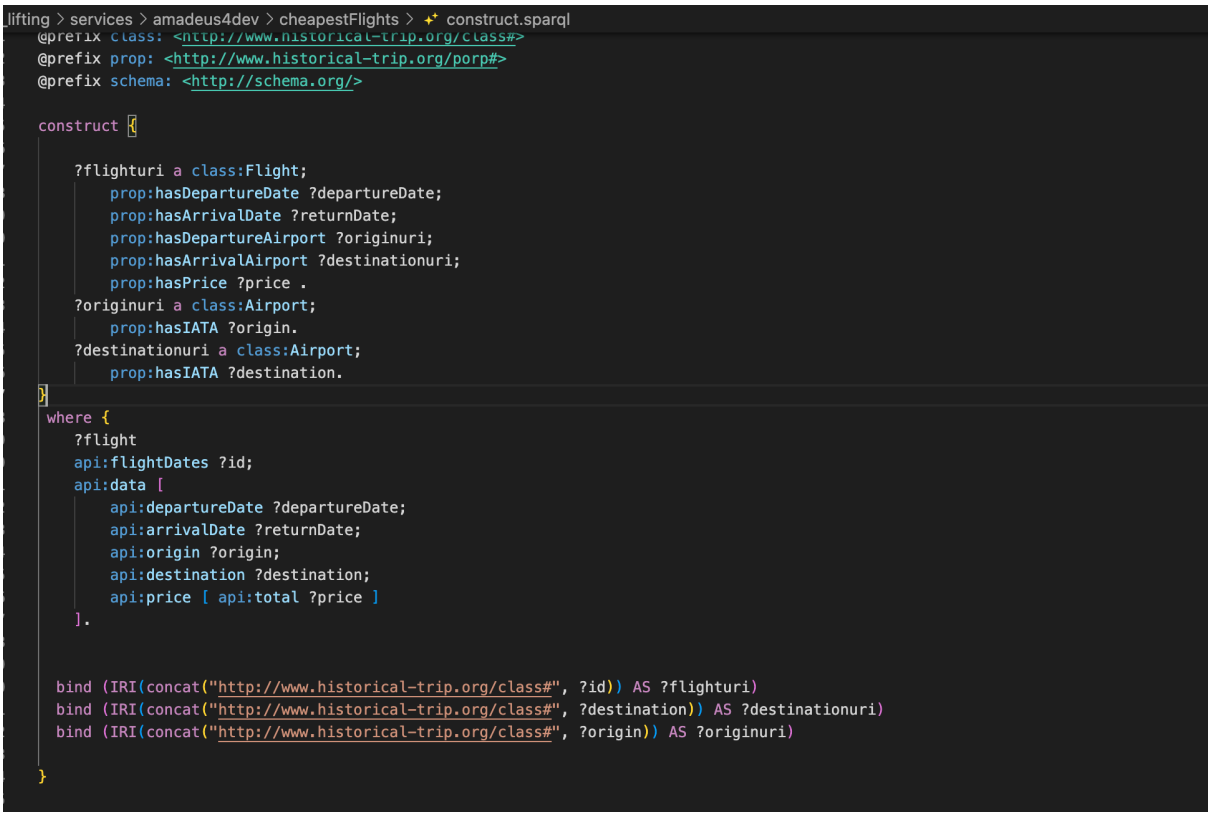

Figure 5. requête SPARQL CONSTRUCT

Notre requête SPARQL CONSTRUCT permet de créer des triplets en format en utilisant les préfixes "class", "prop" et "schema". La requête définit les triplets pour une ressource Vol, une ressource aéroport d'origine et une ressource aéroport de destination. La ressource Vol a des propriétés de date de départ, date d'arrivée, aéroport de départ, aéroport d'arrivée et prix. Les ressources aéroport ont des propriétés IATA. La requête utilise le préfixe api pour obtenir des données de l'API Amadeus, y compris la date de départ, la date d'arrivée, l'origine, la destination et le prix. L'IRI pour chaque ressource est créé en utilisant la fonction concat, qui combine une chaîne statique avec une valeur dynamique (par exemple, ?id).

Application

Fonctionnalité

Nous visons à enrichir l'expérience utilisateur en ajoutant plusieurs fonctionnalités innovantes. Tout d'abord, nous permettons aux utilisateurs de trouver facilement les vols les plus économiques en fonction de leur destination souhaitée. En outre, si les utilisateurs ne spécifient pas leur destination, notre logiciel peut afficher toutes les informations sur les vols disponibles et sélectionner la tarification la plus abordable

en fonction de leur budget. Enfin, pour les utilisateurs qui cherchent à planifier un voyage culturel, ils peuvent accéder à des informations détaillées sur les différentes villes culturelles et les attractions touristiques, pour faire un choix éclairé en fonction de leurs préférences et intérêts.

Interface Web

En ce qui concerne l'interface Web, nous avons opté pour Streamlit. Il s'agit d'un cadre open-source pour la création d'applications interactives basées sur le web en utilisant le langage Python. Les avantages offerts par Streamlit incluent une utilisation facile, des mises à jour dynamiques, un déploiement rapide et un nombre minimal de lignes de code requises. En conséquence, Streamlit apparaît comme une solution viable pour la création d'interfaces Web simples et interactives.

Nous avons aussi utilisé la bibliothèques rdflib pour implémenter une interface utilisateur simple pour une application de recherche de vol vers une destination historique. L'utilisateur peut entrer un prix maximum, et l'application affichera une liste de vols en dessous du prix maximum, avec leurs dates de départ et d'arrivée, leur prix et la ville de destination. L'utilisateur peut sélectionner un des vols, et les informations seront affichées.

La fonction get_flights_by_price qui est dans app.py utilise SPARQL pour interroger un graphique RDF de données de vol stocké dans un fichier Turtle et récupérer les vols qui ont des prix inférieurs ou égaux au prix maximum spécifié par l'utilisateur.

La fonction extract_monument_data récupère des informations sur les monuments dans une ville donnée, y compris leur nom, les heures d'ouverture, la notation, l'URL wikidata et la description.

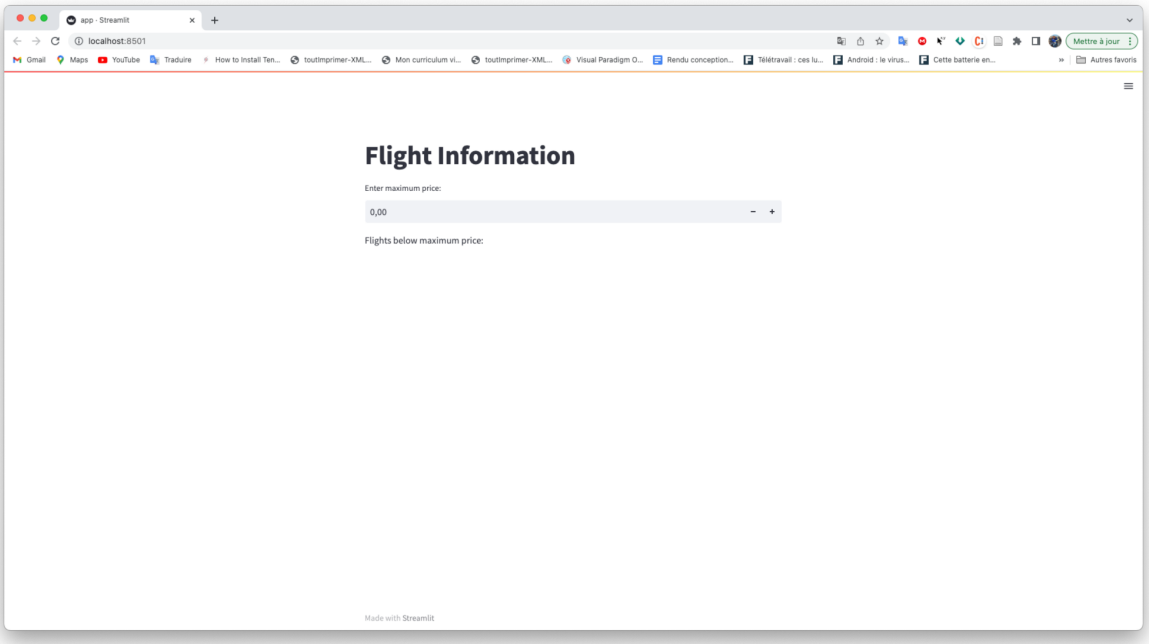

Figure 6. Interface Web page d'accueil

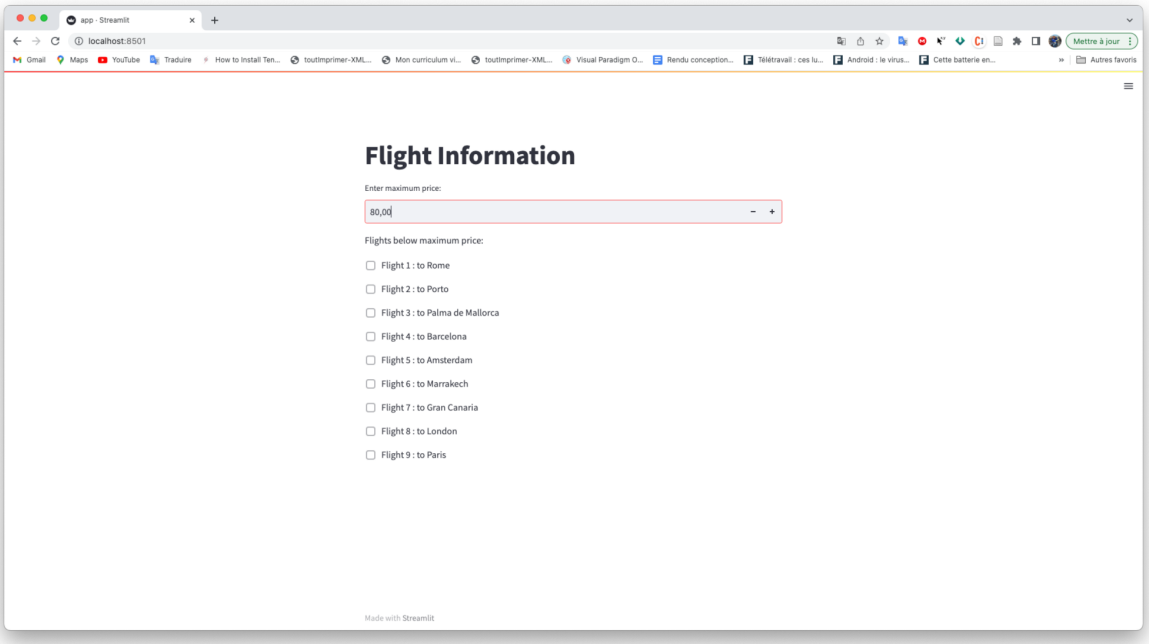

Figure 7. Page de recherche d'information du vol

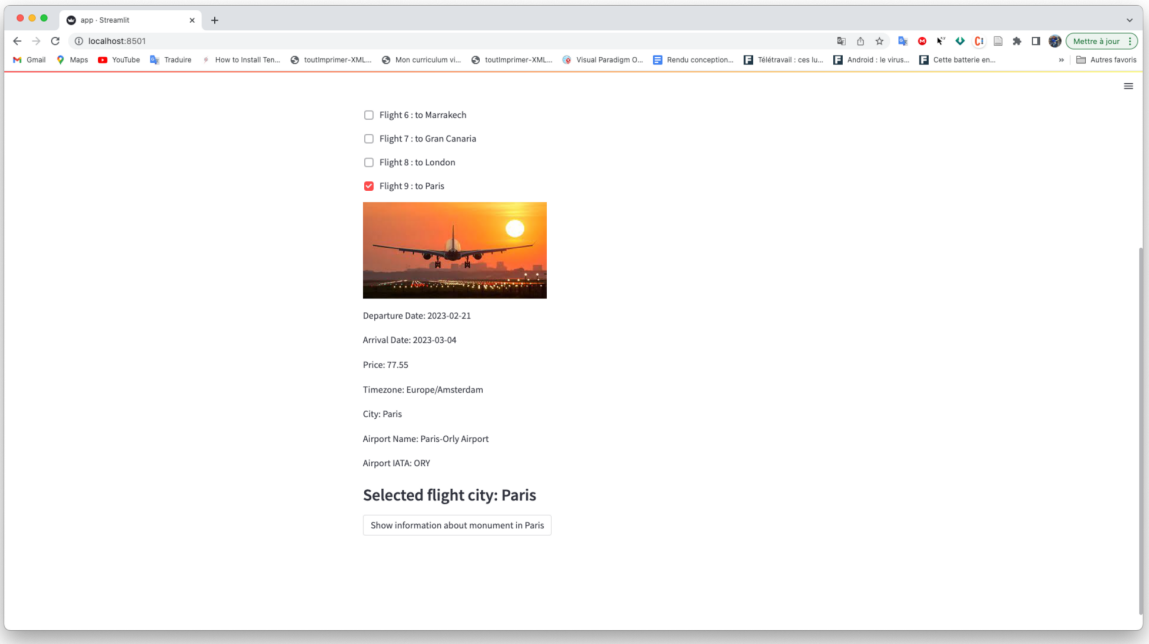

Figure 8. Séléction du vol

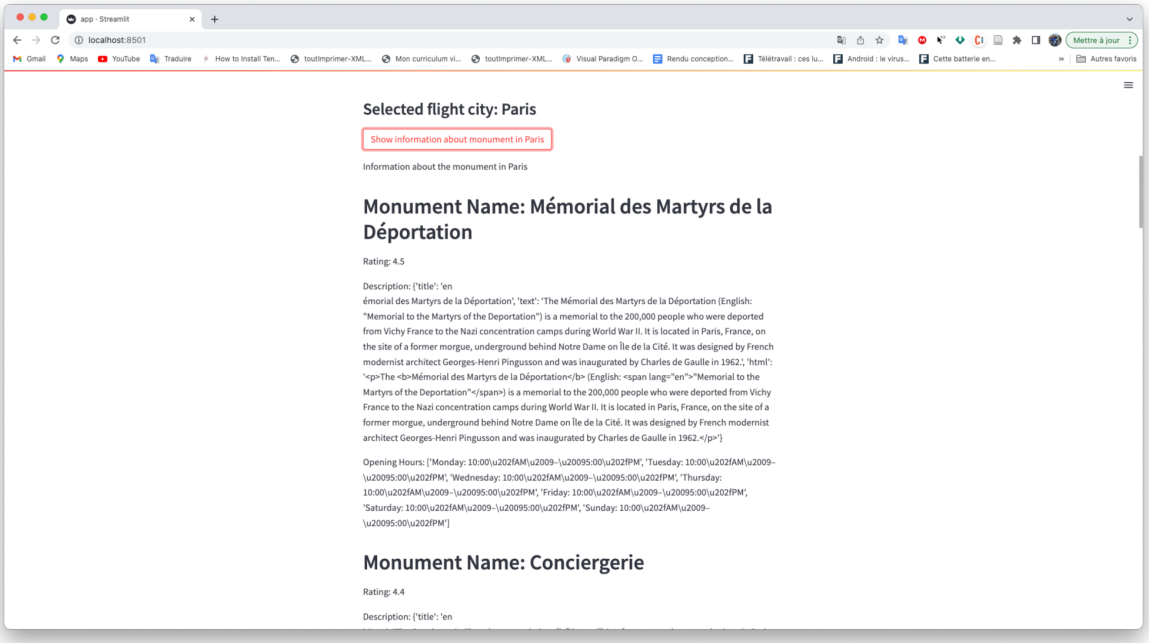

Figure 9. Affichage d'information du monument

Référence:

https://opentripmap.io/docs
https://developers.google.com/maps/documentation/places/web-service?hl=en

https://developers.amadeus.com/self-service/apis-docs
https://docs.streamlit.io/
https://rdflib.readthedocs.io/en/stable/

Github

https://github.com/sana-dibe/Projet-Final-Web-Semantique