DATAENGINEERING

DOCUMENTATION TECHNIQUE

RÉALISÉ PAR BENJAMIN NAHUM
&
FAISSAL KOUTTI

Sommaire

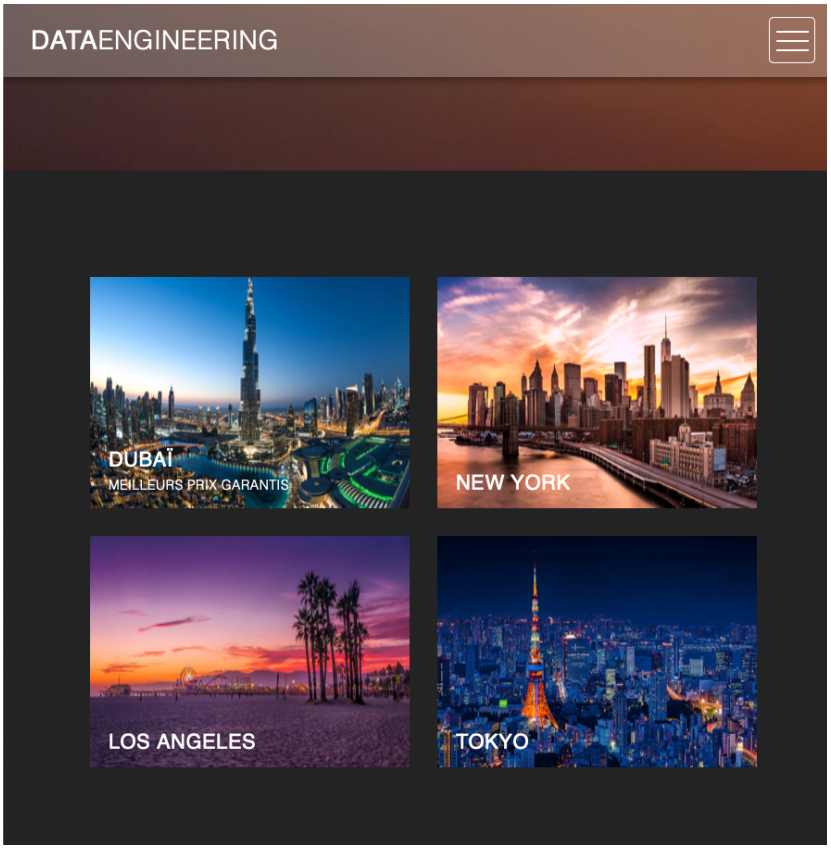
Objectifs

Notre projet aura pour but de prélever les hôtels de luxe de différentes destinations mondiales, et ce, sur plusieurs semaines. Une fois extraites, nous les stockons dans nos base de données dans l'optique d'afficher à la fois grâce à la framework Flask mais aussi Dash:

- les hôtels organisés de la manière la plus optimale, avec la possibilité pour l'utilisateur de trier ses recherches sur notre site

- des études sur les différentes destinations, des graphiques et des cartes intéressantes montrant les diverses fluctuations de prix entre les différentes mégapoles, mais aussi en fonction du temps et surtout du site source où l'on a prélevé les données ! (Hotels.com et Expedia.fr)

Les destinations seront réparties sur plusieurs pages du site, et les différentes recherches sont affinées grâce à ElasticSearch et MongoDB. Quant aux études réalisées à travers de diverses formes de graphiques, nous avons choisi d'implémenter Dash dans Flask afin d'intégrer directement les plot au sein même de notre site.Afin de faire partager notre projet sur plusieurs serveurs machine, nous utilisons Docker.



4

Scrapy

La première étape du projet consiste à scraper les données des deux sites. Nous utilisons Scrapy afin de prélever les données qui nous intéressent sur leur site.
Ainsi, nous avons deux spiders nommées hotelscom.py et expedia.py où se situent nos requêtes .css.
Nos spiders parcourent plusieurs blocs du site pour prélever des informations éparpillées dessus.

```
def parse(self, response):
    requetes = response.xpath("//li[starts-with(@class, 'hotel vip')]")
    requetes2 = response.xpath("//li[starts-with(@class, 'hotel')]")
    requetes3 = response.xpath("//li[starts-with(@class, 'hotel sponsored')]")
    requetes4 = response.xpath("//li[starts-with(@class, 'hotel sponsored vip')]")
    #requetefinale = response.xpath("//ol[starts-with(@class, 'listings infinite-scroll-enabled')]")
    dateArrive = response.css("#q-localised-check-in").css("::attr(value)").extract()
    dateDepart = response.css("#q-localised-check-out").css("::attr(value)").extract()
    id = response.css(".widget-query-group.widget-query-destination").css("input::attr(value)").get()
    site = "hotels.com"
    for i in requetes:
        yield{
        'site' : site,
        'id' : id,
        'title' : i.css(".p-name").css("a::text").get(),
        'dateArrive' : dateArrive,
        'dateDepart' : dateDepart,
        #'adresse' : i.css(".address").css("::text").get(),#adresse de l'hotel
        'localisation' : i.css(".location-info.resp-module .map-link.xs-welcome-rewards").css("::text").extract()  ,#localisation
        'prix' : i.css(".price-link").css("ins::text").get(),#prix pour <ins>
        'prixbis' : i.css(".price-link").css("strong::text").get(),#prix pour <strong>
        'avis' : i.css(".small-view").css("span::text").get(),#nombre d'avis
        'nombreNuits' : i.css(".price-info").css("::text").get(),#nombre de nuits
        #'etoiles' : i.css(".star-rating-text").css("span::text").get(),#etoiles
        'notes' : i.css(".guest-reviews-badge").css("::text").get(),#notes
        'images' : i.css(".u-photo.use-bgimage.featured-img-tablet").css("::attr(style)").extract()#Images
        #response.css("#listings .property-image-link").css("img::attr(style)").getall`all()#ImagesBis
        }

    for i in requetes2:
        yield{
        'site' : site,
        'id' : id,
        'title' : i.css(".p-name").css("a::text").get(),
        'dateArrive' : dateArrive,
```

Pour ce faire nous avons créés des boucles passant dans les différentes requêtes du type response.xpath(---) afin de prendre toutes les données en continu. Après avoir attribuées les données scrapées des deux sites sur leurs spiders respectives dans le yield, nous stockons les données dans des fichiers CSV. Nous fusionnons les 2 fichiers des deux sites et nous les transformons sur le notebook. pour une utilisation plus pratique.

5

Les pipelines ont ensuite été utilisés pour traiter les différents items que nous avons obtenu en scrapant. Lorsqu'un article est envoyé dans le pipeline d'items, il est scrapé par la spider et traité à l'aide de plusieurs composants. Nous les avons ainsi utilisés dans le fichier pipelines.py du projet newscrawler pour l'encodage de nos strings, le stockage dans Mongodb (ci dessous)

```
class MongoPipeline(object):

    collection_name = 'scrapy_items'

    def open_spider(self, spider):
        self.client = pymongo.MongoClient()
        self.db = self.client["hotelscom"]

    def close_spider(self, spider):
        self.client.close()

    def process_item(self, item, spider):
        self.db[self.collection_name].insert_one(dict(item))
        return item
```

6

Flask/routes

Le framework Flask a été utilisé pour obtenir des serveurs web. Ainsi, le fichier views.py contient les différents accès des pages de notre site. C'est celui ci qui permettra d'implémenter notre Dash au sein même de notre serveur Flask

```
app = Flask(__name__)
dash_app = dash.Dash(__name__, server=app, routes_pathname_prefix= '/dash/')
dashboard.GraphDash(dash_app=dash_app)
```

Par la suite, les différents fichiers html seront conduits grâce au routing, contenant diverses requêtes en ElasticSearch et en Mongo

```
@app.route('/dubai.html', methods=['GET','POST'])
def dubai():
    ddubai = collection.find({"id": "Dubaï"})
    response =[]

    for document in ddubai:
        response.append(document)

    if request.method == 'POST':
        query = es_client.search(
        index="hotels",
        body={"query": {"term" : { 'id': 'Dubaï'} }
            }
        )
        [elt['_source'] for elt in query["hits"]["hits"]]
        return render_template('dubai.html', res=query)
    else:
        return render_template("dubai.html", hotels=response)
```

Une fois les requêtes effectuées, nous utilisons jinja2 pour interagir directement avec les fonctions python au sein du code html, ou encore de créer des boucles pour faciliter la lisibilité du code.

```
    <span class="card__title">Notes : {{hotel['notes']}}/10 sur {{hotel['nombreAvis']}} avis</span>
  </div>
</li>


{% endfor %}
```

Bases de données NoSQL: MongoDB et ElasticSearch

Pour stocker les données, nous avons opté pour la base de données noSQL MongoDB.
En effet, celle-ci est connue pour être flexible et particulièrement simple d'utilisation. L'affichage des données, maintenant disponibles dans notre dataFrame grâce au scraping, comprendra 3 étapes avant d'être mises à disposition sur notre site.

La première est la collection. Dans les phases de routing, nous créons diverses collections MongoDB et requêtes elasticSearch. en fonction du site, de la date, prix etc.

Ensuite, nous dirigeons nos requêtes vers la template souhaitée, préalablement créée et modifiée parmi les fichiers HTML.

```
lla = collection.find({"id": "Los Angeles"})
response2 =[]
for document in lla:
    response2.append(document)

if request.method == 'POST':
    query = es_client.search(
    index="hotels",
    body={query :{"query": {"term" : { 'id': 'Los Angeles'} }}
        }
    )
    [elt['_source'] for elt in query["hits"]["hits"]]
    return render_template('la.html', res=query)
else:
    return render_template("la.html", hotels=response2)
```

Enfin, nous pouvons afficher les données correspondantes dans les onglets attribués.

```
<span class="card__category">{{hotel['prix']}} €</span>
<h3 class="card__title">{{hotel['title']}}</h3>
<span class="card__title">Localisation : {{hotel['localisation']}}</span>
<span class="card__title">Site internet : {{hotel['site']}}</span>
```

Docker

```
version: '3'

services:
  elastic:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.4.0
    container_name: elasticsearch
    environment:
      - "ES_JAVA_OPTS=-Xms512m -Xmx512m"
      - discovery.type=single-node
    ports:
      - "9200:9200"
  mongo:
    image: mongo
    container_name: mongo
    environment:
      - MONGO_DATA_DIR=/data/db
      - MONGO_LOG_DIR=/dev/null
    volumes:
      - ./data/mongo:/data/db
    ports:
        - "27018:27017"


networks:
  default:
```

Notre dossier contient un fichier docker compose. L'utilisateur doit lancer la commande "docker-compose up -d" afin d'instancier toutes les bases de données et l'image principale.