Jérémy Surget | Sylvain Jiang

Documentation Technique

Data Engineering

Web app des produits reconditionnés Apple

Notre projet consiste à récolter les produits reconditionnés mis en ventes par Apple dans le but de les stocker et les afficher via une web application avec le framework Flask. Nous avons aussi utilisé Elastic Search pour créer un moteur de recherche et de suggestion sur notre site web. Pour comparer avec un autre site de déconditionné tel que Back Market nous avons utiliser Dash pour créer les graphiques de comparaison que nous avons directement intégré dans notre application. Docker nous a permis de virtualiser un serveur de base de données et un Elastic Search.

Flask

Pour faire tourner tout notre framework nous avons utilisé le Framework python Flask qui permet de créer des serveurs web avec python.

Notre fichier « web.py » est le principal et regroupe toutes les routes de l'application. Nous avons défini une route en cas d'erreur 404, cette route renvoi un template personnalisé d'une page « erreur 404 »

```python
@app.errorhandler(404)
def not_found(e):
    return render_template("404.html")
```

En parlant de template, Jinga 2 est un moteur de rendu de template qui fonctionne avec Flaks pour nous permettre de customiser nos pages HTML avec du code, tel que des boucles for ou if par exemple. Nous l'avons utilisé notamment pour afficher nos produits ou bien vérifié si une liste était vide ou pas.

```
<div class="result-number">
  {% if res %}
  {% if res['hits']['hits']|length == 0 %}
  <p>Nous n'avons rien trouvé 🤔</p>
  {% else %}
  <p>Nombre de resultats {{res['hits']['hits']|length}}</p>
  {% endif %}
  {% else %}
  <p> Il y a actuellement {{products|length}} produits sur le refurb Apple</p>
  {% endif %}
</div>
```

Pour ce qui est de du style de notre site web, nous avons utilisé un template HTML et CSS pour notre page d'accueil issu du site https://colorlib.com/. Nous ensuite adapté le reste de nos pages à ce style la. L'animation sur la barre de recherche est aussi un template CSS que nous avons trouvé sur codepen https://codepen.io/tommysmall92/pen/oPjaNW?page=1&

Scraping

Récupérer les données

Pour récupérer les données du site Apple nous avons utilisé Scrapy pour scraper leur page de reconditionnés.

Notre spider applescrap.py permet de récupérer notamment le titre, le prix, l'image etc… des articles Apple concernant les Iphones, Ipad, Macbook, iMac. Nous utilisons les propriétés css de scrapy pour récupérer les données car plus précise et facile que HtmlXPathSelector.

Notre spider parcours toutes les pages de présentation de produit et pas seulement la pages d'accueil des produits. Pour cela nous utilisons un callback pour chaque page, ou notre scraping sera exécuté.

```
    def parse(self, response):
        all_links = {
            name:response.urljoin(url) for name,url in zip(
            response.css(".refurbished-category-grid-no-js").css("li").css("a::text").extract(),
            response.css(".refurbished-category-grid-no-js").css("li").css("a::attr(href)").extract()
            )
        }
        for link in all_links.values():
            yield Request(link, callback=self.refurbished_product)
            print(link)
```

« self.refurbushed_product » est notre méthode qui scrap les champs qui nous intéressent pour chaque article.

Les pipelines

Une partie importante a été de mettre en place les pipelines qui permettent d'ingérer et de traiter toutes les données scrapé au moment du scraping.

Nous avons 3 pipelines :
- Uniformisé le texte
- Stocker dans MongoDB
- Indexer dans Elastic Search

La première sert simplement à supprimer les espaces des champs que nous scrapons.

La deuxième permet de stocker les items dans mongoDB qui est la base de données que nous avons utilisé dans le projet et dont nous parlerons plus loin.

Enfin la troisième pipeline nous permet d'indexer toutes les données dans ES pour ensuite pouvoir faire notre moteur de recherche que nous détaillons plus loin.

Tous les données scrapé sont stocké dans un document json pour avoir une trace au cas ou. Nous avons défini ce paramètre dans les settings de notre crawler.

```
FEED_FORMAT= "json"
FEED_URI= 'apple.json'      You, 3 days ago • crawling run from python script
```

Lancement de la spider

Dans le but d'automatiser complètement l'application, nous exécutons le crawler depuis notre script python.

Fichier web.py

```
scraper = Scraper()
scraper.run_spider()
```

Fichier scraper.py

```
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings
from newcrawler.newcrawler.spiders import applescrap
import os


You, 3 days ago | 1 author (You)
class Scraper:
    def __init__(self):
        settings_file_path = 'newcrawler.newcrawler.settings'
        os.environ.setdefault('SCRAPY_SETTINGS_MODULE', settings_file_path)
        self.settings = get_project_settings()
        self.process = CrawlerProcess(self.settings)
        self.spider = applescrap.AppleSpider

    def run_spider(self):
        self.process.crawl(self.spider)
        self.process.start()  # le script se bloque ici jusqu'a la fin      You, 3 days ago • crawling run from
```

Ainsi dès que l'application est lancée le scraping se lance. La base de données ainsi que ES sont vidés complètement et remplacé par les nouvelles données. Cela permet d'actualiser au cours de la journée les données présentes sur le refurb d'Apple étant donnée que certains produits deviennent indisponible au cours de la journée.

Nous pourrions améliorer ce système en gardant les données à chaque scraping mais en comparant avec leur ID si les produits sont disponibles ou pas et ainsi passer un champ « available » à True ou False en fonction. L'affichage se ferait en fonction de se champ et on pourrait imaginer de griser les produits dont le champ available est à False.
A la fin de la journée tout serait sauvegarder dans une autre collection pour laisser la place au données du lendemain et pour faire un historique des produits régulièrement disponibles sur Apple.

MongoDB

Nous avons utilisé MongoDB pour stocker nos données. C'est une base de données no_SQL ce qui l'avantage d'être particulièrement souple dans le modèle de données car tout est organisé en document. Comme nous avons changé plusieurs fois notre scraping Mongo s'adaptait parfaitement de par sa souplesse au niveau du stockage contrairement au BDD SQL.

Dès que la spider est lancée, nous créons une collection « refurbApple » si celle-ci n'existe pas et nous ajoutons des documents dont les ID sont les IDs des produits sur le site Apple ainsi qu'un hash du jour, car nous avons remarqué que certains produits pouvaient avoir le mêmes ID.

Pour notre page de recherche nous si rien n'est recherché nous affichons tous les documents de la BDD

```
def refurbComparaisonPage():
    documents = collection_product.find({})
    response = []
    for document in documents:
        response.append(document)
```

La liste des documents est ensuite passés au template HTML pour nous permettre de l'afficher

```
    return render_template("refurb_page.html", products=response)
```

Ainsi nous pouvons sélectionner les différents champs que nous voulons afficher pour les produits.

```
<span class="card__category">{{product['currentPrice']}}</span>
<h3 class="card__title">{{product['title']}}</h3>
<span class="card__by"><i>{% if product['save'] != None %}{{product['save']}}
{% endif %}</i></span>
```

Elastic Search

Pour notre moteur de recherche et de suggestions nous avons utilisé Elastic Search qui permet d'indexer des données pour faire de la recherche rapide dessus.

Recherche

Dans notre pipeline nous indexons chaque item dans ES pour permettre une recherche dessus. Ainsi, nous créons un index appelé « product » et nous lui passons sous forme de dictionnaire les champs des items.

```
self.es_client.index(
    index="product", doc_type='product', id=item['id'], body=item_dict)
```

Pour nous permettre d'afficher produits en fonction de la recherche que l'utilisateur a effectuée, nous faisons une requête à ES avec comme query de la requête ce que l'utilisateur a entré dans la bar de recherche. Nous passons ensuite le résultats au template pour faire le mêmes rendu que énoncé plus haut.

```
if request.method == 'POST':
    search_term = request.form["input"]
    query = es_client.search(
        index="product",
        size=30,
        body={
            "query": {
                "multi_match": {
                    "query": search_term,
                    "fields": [
                        "title",
                        "currentPrice",
                    ]
                }
            }
        }
    )

    #es_client.search(index="suggest_product", body=suggest, size=10)
    return render_template('refurb_page.html', res=query)
```

L'option « fields » nous permet de préciser les champs sur lesquels nous voulons effectuer la recherche. Ici nous ne recherchons que sur le titre et le prix du produits, « size » permet de préciser la taille maximale de la réponse.

Suggestions

Elastic Search possède une fonctionnalité de « Search-As-You-Type » qui permet de suggérer des produits à l'utilisateur pendant que celui-ci est en train de taper dans la barre de recherche. Cette fonctionnalités a été un peu plus compliquée à mettre en place.

Tout d'abord dans la pipelines nous avons du créer un nouvel index spécialement pour cela. Cet index possède un mapping spécifique où nous précisons le type de notre champ titre pour que celui-ci possède la fonction « Search-As-You-type ».

Voici à quoi ressemble notre pipeline complète pour ES.

```python
class IndexElasticSearch(object):
    def open_spider(self, spider):
        # verif ca fonctionne sur un autre pc ????
        self.es_client = Elasticsearch(
            hosts=["localhost" if ES_LOCAL else "elasticsearch"])

        mapping = {
            "mappings": {      You, a month ago • added suggest index and mapping
                "properties": {
                    "title": {
                        "type": "search_as_you_type"
                    }
                }

            }
        }

        self.es_client.indices.create(
            index='suggest_product', body=mapping)

    def process_item(self, item, spider):
        item_dict = dict(item)
        p_suggest = {
            "title" : item_dict['title'],
        }
        self.es_client.index(
            index="product", doc_type='product', id=item['id'], body=item_dict)
        self.es_client.index(index="suggest_product", body=p_suggest)
        return item
```

Ensuite dans le but de pouvoir suggérer sans quitter la page ou se trouve le champ de recherche nous avons du créer un « endpoint » qui renvoi les résultats pour la suggestions. Notre endpoint est une route dans notre application Flask sur laquelle nous allons envoyer une requêtes à chaque modification dans le champ de recherche, chaque modification va query via l'url de ce endpoint. Par exemple si l'utilisateur tape « lp » l'url va ressembler à « …/suggest_product/suggest?search=lp » nous pouvons donc récupérer ce que ES nous renvoi.

```python
@app.route('/suggest_product/suggest', methods=['GET', 'POST'])
def suggest_method():
    if request.method == 'GET':
        #permet de recuperer le query passé en url grace au JavaScript
        req = request.args.get('search')

        query = es_client.search(
            index="suggest_product",
            size=5,
            body={
                "query": {
                    "multi_match": {
                        "query": req,
                        "type": "bool_prefix",
                        "fields": [
                            "title",      You, 15 days ago • changed ES mapping name for better understanding
                            "title._2gram",
                            "title._3gram"
                        ]
                    }
                }
            }
        )
        return query
    else:
        return "La méthode devrait renvoyer un index ES à la suite d'un GET pas d'un POST"
```

Tout ce passe ensuite du coté de L'HTML et surtout du javascript.

```javascript
var titles = {}
var searchoption = null;
var doctype = $("#customsearch").data('doctype');

$("#customsearch").on('input', function(){
  autosuggest($(this).val());
 });
//requet GET à chaque nouvelle lettre tapées
// on a crée un url en endpoint -> /suggest_product/suggest qui nous retourne les query d'ES
function autosuggest(value){
  $.get("/suggest_product/suggest", {search: value},         function(data){
    console.log(data);
   showSuggestions(data);
  });
 }
//affiche les resultats renvoyés par ES
 function showSuggestions(data){
  var div = '';
  console.log(data.hits.hits);
  data.hits.hits.forEach(function(suggestion){
    div += ('<option value="'+ suggestion['_source']['title']+'">');

  });
  $('#suggestions').html(div);
 }      You, 15 days ago • changed ES mapping name for better understanding
```

À chaque modification dans la barre de recherches des options sont ajoutés dans le tag « datalist » qui nous permet d'afficher les suggestions sont changer de page.

```
<form action="" class="search-form" method="POST">
  <input name = "input" type="text" placeholder="Rechercher ..." id="customsearch" list="suggestions" autocomplete="off">
  <datalist id="suggestions">
    <!-- les options vont etre insérées via le script JS -->
  </datalist>
```

Dash

Dans le but de pouvoir montrer des comparaisons entre produits nous avons utilisé Dash.

Pour pouvoir utiliser Dash dans notre application Flask déjà existante nous passons simplement l'instance de l'application Flask en serveur à Dash et pour facilité le développement nous avons créer une class qui créer toute l'application Dash et nous n'avons plus qu'a l'instancier elle aussi au lancement de notre application.

Web.py

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

chart.py

```
class GraphDash:
  def __init__(self, dash_app):
    super().__init__()
    self.dash_app = dash_app

    with open('apple.json', 'r') as file:
```

Pour intégrer Dash dans le style de notre application nous l'affichons dans une iframe HTML, ce n'est probablement pas la meilleure solution, mais elle a l'avantage d'être efficace.

```
<div id="dashframe">
    <iframe src="http://127.0.0.1:2745/dash/"></iframe>
</div>
```

Docker

Nous avons utilisé docker pour virtualiser des serveurs MongoDB ainsi que Elastic Search.

Notre projet contient un fichier docker compose qui permet de monter et lancer les images MongoDB et ES, essentielles au bon fonctionnement de notre web app.