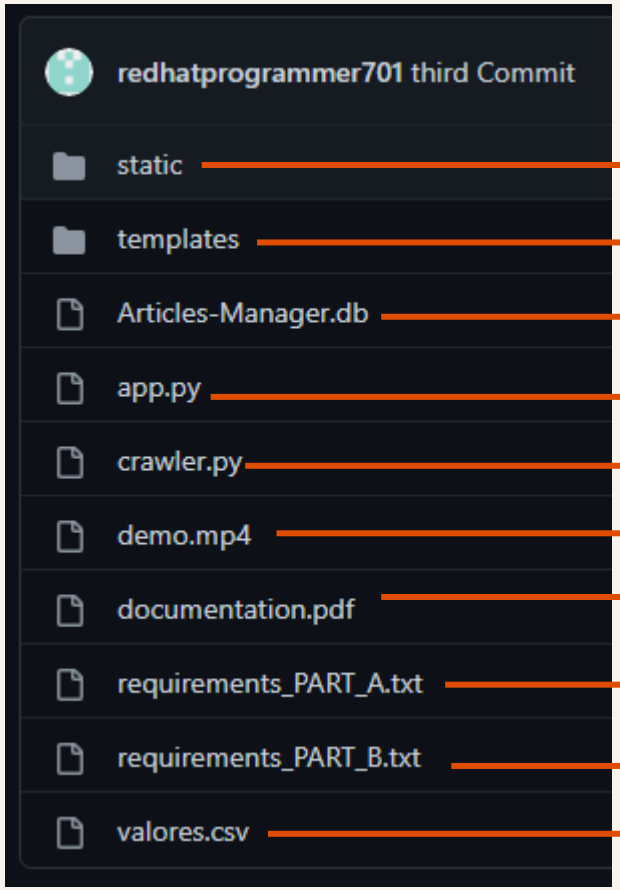
Documentación

Case Study - Junior Python Developer - Dauphin

Castleberry Media

Rafael Eduardo Monsalve Arboleda

Estructura del Repositorio



Capeta con archivos estáticos, para este caso contiene a `styles.css` que se encarga de estilizar la pagina principal.

Contiene a `index.html`, el cual se encarga de mostrar la estructura de la página principal.

Base de datos que contiene la tabla ARTICULOS.

Archivo Python con el código del Aplicativo Flask

Archivo Python con el código de Web Scraping

Video donde se presenta el funcionamiento del aplicativo Flask

Documentación de los programas realizados

Librerías requeridas para La Parte A

Librerías requeridas para La Parte B

Archivo generado por crawler.py con los datos de los artículos.

PARTE A - CRAWLER

Partiendo de los links de cada tag suministrados por la función `extract_tags()`, Se crea la función `extract_information()` que se encarga de extraer la información requerida, a continuación se dará un recorrido a lo largo del contenido de esta función:

Primera parte de la función

```
def extract_information():
    """""""
    article_titles=[]
    article_resume =[]
    article_tags=[]
    article_links=[]
    article_dates=[]

    try:
        for link in extract_tags()[1:]:
            driver.get(link)
            sleep(2)

            src = driver.page_source
            soup = BeautifulSoup(src,'lxml')
            wrapper = soup.find('main',{'class':'body-container-wrapper'})
            titles = wrapper.find_all('a',{'class':None}) #titles
            resume = wrapper.find_all('p',{'class':None}) #Resumen
```

Definición de la Función
Y de las listas que almacenarán la información

Ciclo For para extraer la información de cada link a partir de la función `extract_tags()` y se inicia el webdirver de Selenium

Se extraen los datos con BeautifulSoup a partir de etiqueta y clase.

Segunda parte de la función

Se extraen los nombres de los tags tomando la parte final de cada URL y eliminando la separación por guion. También se extraen las fechas a partir de etiqueta y clase con BeautifulSoup.

```
        for i in range(len(titles)):
            tag = link.split('/')[-1].replace('-', ' ')
            article_tags.append(tag) #tags
        dates = wrapper.find_all('span',{'class':'blog-index__post-date'}) #dates

        for (title, date, resume_) in zip(titles, dates, resume):
            article_titles.append(title.get_text())
            article_links.append(title['href'])
            date_ = date.get_text().strip()
            date_obj = datetime.strptime(date_, '%b %d, %Y')
            date_formmatted = date_obj.strftime('%Y-%m-%d')
            article_dates.append(date_formmatted)
            article_resume.append(resume_.get_text())

    except InvalidArgumentException:
        print('Invalid URL')
        exit()

    driver.close()
    return(article_dates, article_titles, article_resume, article_tags, article_links)
```

Excepción por si no encuentra o hay algún error con la URL.

Se realiza un ciclo para agregar cada elemento a las lista vacías definidas anteriormente. Para cada uno se utiliza la función `.get_text()` para obtener su contenido (titulo, resumen, fecha), para el caso de los links se utiliza [‘href’] .A las fechas se le agrega un formato con la ayuda de la modulo `datetime`.

Se Cierra el webdriver y se retornan las listas con la información requerida.

Para finalizar se almacenan las listas que retorna la función `extract_information()` en un diccionario para almacenarlo en un DataFrame y posteriormente guardarlo como 'valores.csv'

```
info = extract_information()

info_data = {
    "id_": [i for i in range(len(info[1]))],
    "Date": info[0],
    "Title": info[1],
    "Resume": info[2],
    "Tag": info[3],
    "Link": info[4],
}

df = pd.DataFrame(info_data)
df.to_csv('valores.csv', index=False)
```

Ejecución de la función y se almacena en la variable `info`.

Almacenamiento de las listas retornadas por la función en un diccionario.

Creación de un DataFrame a partir del diccionario con la ayuda de la librería Pandas, luego se crea 'valores.csv' con la información obtenida.

PARTE B – Aplicativo con Flask

Se realizará una recorrido a lo largo del código para el aplicativo con Flask al igual que con la PARTE A.

Primera parte de la Aplicación

Se crea una aplicación con el framework Flask y un motor de bases de datos con SqlAlchemy, para este caso se escogió SQLite.

La tabla ARTICULOS se implementa a partir de una clase llamada `Article` la cual hereda las características del método `declative_base()` almacenado en la variable `Base`, Se añaden los campos requeridos con su respectivo tipo de dato.

```
app = Flask(__name__)
engine = create_engine('sqlite:///Articles-Manager.db', connect_args={
'check_same_thread': False})

Base = declarative_base()

class Article(Base):
    __tablename__ = 'ARTICULOS'

    id_ = Column(Integer, primary_key=True, autoincrement=True)
    Date = Column(String(50), nullable=False, default=datetime.utcnow)
    Title = Column(String(250), nullable=False)
    Tag = Column(String(250), nullable=False)
    Resume = Column(String(250), nullable=False)
    Link = Column(String(400), nullable=False)

Base.metadata.create_all(engine)

df = pd.read_csv("valores.csv")
df.to_sql('Articulos', engine, if_exists='replace', index=False)

session = sessionmaker()
session.configure(bind=engine)
s = session()
```

Con la ayuda de pandas se leen los datos del archivo 'valores.csv', previamente creado, y se inyecta esta información en la base de datos(esta base de datos fue previamente creada) con el 'engine' de SQLite.

Se utiliza el método `sessionmaker()` para crear una sesión que permitirá hacer peticiones a la base datos.

Segunda parte de la Aplicación

Se define la ruta raíz de la aplicación y los métodos GET para mostrar los datos y POST para recibir los datos de entrada en la barra de búsqueda.

```python
@app.route('/', methods=['GET', 'POST'])
def index():
    results = s.query(Article).all()
    if request.method == 'POST' and 'tag' in request.form:
        tag = request.form["tag"]
        search = "%{}%".format(tag)
        results = s.query(Article).order_by(desc(Article.Date)).filter(Article.Tag.like(search)).all()
        return render_template('index.html', results=results, tag=tag)
    return render_template('index.html', results=results)

if __name__ == "__main__":
    app.run(debug=True)
```

Se Realiza una petición a la sesión creada previamente donde se solicitan todos los datos y se almacena en la variable `results`

La siguiente condición se emplea para asociar la entrada de datos en la barra de navegación con los datos de la tabla ARTICULOS. Para ello la etiqueta <input> de HTML se nombre como 'tag' y utiliza el método POST (véase `index.html`). Así pues al ingresar el nombre del tag del articulo(o una palabra asociada) se buscará en la base de datos un elemento asociado.

Esta petición busca por el valor que se ingresa en la barra de búsqueda y ordena por fecha de forma descendiente.

En la imagen se presenta el diseño final de la aplicación, Se invita a ver el video demo.mp4 que encuentra en el repositorio para apreciar mejor su funcionamiento.