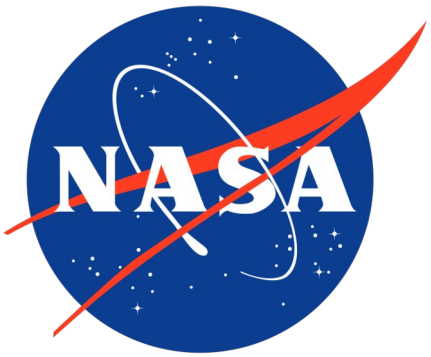
Introducción a la Programación - primer semestre del 2024.

Trabajo práctico: galería de imágenes de la NASA



Integrantes:

- Dejesús, Daian Ernesto.
- Gonzalez, Franco Javier.
- Samudio, Lucas Leonel.

El trabajo práctico consta en modificar un repositorio alojado en Git Hub para que sea funcional. El objetivo, es que consuma una API de la NASA para muestre en el navegador los resultados de búsqueda ingresados por el usuario. Para ello, se debió hacer uso de distintas plataformas y entornos de trabajo, principalmente Python, Git, Visual Studio Code y Django, junto con el navegador web.

En principio, para obtener el mínimo de funcionalidad, se tuvieron que completar las funciones de Python que estaban definidas pero no desarrolladas. El primer desafío, y tal vez el más difícil, fue intentar saber como era la estructura del código, disperso en varios módulos, para entender su funcionamiento. Para ello se armaron cuadros sinópticos que fueron debatidos y modificados mediante la colaboración entre los estudiantes. Una vez terminado este paso, se procedió a completar el código hasta hacerlo funcional. En el módulo "views.py" se completaron las siguientes funciones:

```
def getAllImagesAndFavouriteList(query):
    images = transport.getAllImages(query) #en formato json
    mappedImages = [] #lista de objetos de nasacards
    for image in images:
        mappedImages.append(mapper.fromRequestIntoNASACard(image))
    favourite_list = []
    return mappedImages, favourite_list
```

Esta función llamada "getAllImagesAndFavouriteList(query)" toma una lista de elementos en formato "Json" en la variable "images" mediante el llamado de la función "getAllImages(query)" que se encontraba definida en el módulo "transport.py". Ambas funciones trabajan en base a lo buscado por el usuario en el navegador.
Luego, se utilizó un ciclo "for" para recorrer la lista "images" y pasar del formato "Json" al formato "nasacard" a través de la llamada a la función "fromRquestoIntoNASACard()", definida en el módulo "mapper.py". Posteriormente, la función retorna dos listas, una formada por objetos de nasacards llamada "mappedImages" y otra llamada "favourite_list". Esta última lista no se encuentra funcional y se retorna vacía, ya que no se completó el código para su funcionamiento.
En síntesis, la función getAllImagesAndFavouriteList(query) toma un argumento ingresado por el usuario y retorna dos listas, mappedImages y favourite_list, luego de haber consumido la API.

La siguiente función modificada fue la que se muestra a continuación:

```
def home(request):
    # llama a la función auxiliar getAllImagesAndFavouriteList() y obtiene 2 
listados: uno de las imágenes de la API y otro de favoritos por usuario*.
    # (*) este último, solo si se desarrolló es opcional de favoritos; caso 
contrario, será un listado vacío [].
    predeterminado="space" #en la pantalla de galería, se mostrará el resulado de 
"space" de forma predeterminada.
    mappedImages, favourite_list = getAllImagesAndFavouriteList(predeterminado)
    return render(request, 'home.html', {'images': mappedImages, 'favourite_list': 
favourite_list} )
```

Esta función renderiza en el navegador las listas arrojadas por la función previamente descrita, mostrando en galería por defecto el resultado de la búsqueda de la variable "predeterminado", cuyo argumento es "space". Aquí surgió una duda, ya que en la imagen del readme se mostraba

como debía quedar al iniciar la galería donde se veía los resultados de “space”, mientras que en la consigna decía que los resultados de “space” se debían mostrar si el usuario realizaba una búsqueda sin ingresar ningún valor. Se optó por realizar ambas acciones, una en la función home(request) y otra en la función search(request), como se describe más abajo.
La funcion search(request) no solo devolverá las listas de resultados de acuerdo a lo ingresado (o no) por el usuario, sino que también se implementaron un sistema de traducción de los vales de búsqueda de español a inglés y un sistema de paginación. Para explicar esta función la dividiremos en secciones

```
def search(request):
        translator = Translator()
        search_msg = request.POST.get('query', '')
#sección de busqueda:
        if not search_msg:
                mappedImages, favourite_list = getAllImagesAndFavouriteList("space")
        else:
#sección de traduccion:
            try:
                translated_search_msg = translator.translate(search_msg, src = "es",
dest='en').text # Momentaneamente traduce de español a inglés.
            except Exception as e:
                print(f"Error translating search query: {e}")
                translated_search_msg = search_msg    # Si no puede traducir, lo
devuelve al mensaje original.
            mappedImages, favourite_list =
getAllImagesAndFavouriteList(translated_search_msg)
#sección de paginación:
    #paginacion de django en resultados de busqueda.
    paginator=Paginator(mappedImages, 6) # El numero de la función es la cantidad de
elementos mostrados en pantalla.
    page_number = request.GET.get('page')
    page_obj = paginator.get_page(page_number)

    # si el usuario no ingresó texto alguno, debe refrescar la página; caso
contrario, debe filtrar aquellas imágenes que posean el texto de búsqueda.
    return render(request, 'home.html', {'images': page_obj, 'favourite_list':
favourite_list} )
```

En la sección de búsqueda se agregó un if. Si el usuario no ingresa ningún valor, de lo tomará como predeterminado el valor “space”, sino será el ingresado por el usuario.
En la sección de traducción, se agregó la librería "googletrans" de python, que implementa funcionamientos de la Google Translate API, la cuál sirve para detectar y traducir distintos idiomas. Se importó desde el modulo 'googletrans' la función “Translator()” en los módulos "views.py" y en "mapper.py", los cuales han sido utilizadas en las funciones "search()" y en "fromRequestIntoNASACard()" respectivamente para realizar las traducciones a los valores ingresados por el usuario.

En "search", si la variable “search_msg” es distinta a vació mediante 'try' se intentará traducir la palabra en idioma ingresado en español para traducirla al idioma destino en inglés , el cual es el idioma nativo de la API pública de la NASA. En caso de no poder traducirlo, imprimirá en pantalla

que no fue posible la traducir y devolverá la variable a su valor original, el cual fue el ingresado por el usuario. En caso de que el usuario no haya ingresado texto alguno, se buscará por el valor predeterminado "space" tal como está definido en el if.

En "fromRequestIntoNASACard", se intentará, mediante dos variables creadas llamadas "translated_title" y "translated_description", la traducción del título y de la descripción de las imágenes solicitadas por el usuario para devolverlas en el idioma español. En caso de no lograrlo, se imprimirá por pantalla que hubo un error al traducir y devolverá los valores originales en lugar de los traducidos.

Si bien quedó funcional y logró la traducción, se encontró la dificultad de que se ralentiza demasiado la renderización de los resultados. Por esta razón, se optó por dejar el código comentado. Esta parte del código se encuentra en el módulo mapper.py

```
# def fromRequestIntoNASACard(object):
#     translator = Translator()
#
#     try:
#         translated_title = translator.translate(object['data'][0]['title'], src='en', dest='es').text
#         translated_description = translator.translate(object['data'][0]['description'], src='en', dest='es').text
#     except Exception as e:
#         print(f"Error translating title or description: {e}")
#         translated_title = object['data'][0]['title']
#         translated_description = object['data'][0]['description']

#     nasa_card = NASACard(
#         title=translated_title,
#         description=translated_description,
#         image_url=object['links'][0]['href'],
#         date=object['data'][0]['date_created'][:10]
#     )

#     return nasa_card
```

En la sección de paginación, se importaron las librerías Django “paginator” y “shortcuts”. Esta funcionalidad de Django permite agregar el sistema de paginación a los resultados de búsqueda. En la variable “paginator” se llamó a la función “Paginator(mappedImages, 6)” para devolver como resultado la cantidad de elementos a mostrar en el navegador, en este caso seis. Luego, se utilizaron las variables “page_number y page_obj” para mostrar los controles de paginación y para renderizar los resultados en la página respectivamente.

El proceso de implementación del paginador fue complicado por varias razones. En primer lugar, la falta de experiencia en la utilización de funciones de Django, para lo cual se tuvo que consultar su documentación oficial, que aunque se encuentra disponible en ingles, es amigable para el lector. En segundo lugar, la falta de conocimientos en HTML para mostrar en pantalla los controles de paginación llevó mucho tiempo, al tener que adquirir los conocimientos básicos necesarios para su correcto funcionamiento. En un primer momento se logró que los controles aparezcan en pantalla, pero quedaban solapados por el footer, imposibilitando su uso. Estos problemas fueron

solucionados hasta dejarlo funcional, aunque no se pudo agregar los estilos deseados de CSS, pese al haber intentado con la etiqueta <style> y desde el archivo “style.css”, incluso utilizando los selectores de clase. Se atinó a agregar estilos básicos en linea.
El código HTML para su implementación está en “home.html” y es el siguiente:

```
<!-- Controles de paginación -->
<div class="pagination" style="text-align: center; margin: 20px 0;">
  <span class="step-links" style="display: inline-block;">
    {% if images.has_previous %}
      <a href="?page=1" style="margin-right: 5px; text-decoration: none; color: 
#007bff;">&laquo; Primero</a>
      <a href="?page={{ images.previous_page_number }}" style="margin-right: 5px; 
text-decoration: none; color: #007bff;">Anterior</a>
    {% endif %}

    <span class="current" style="margin: 0 10px; color: #555;">
      Página {{ images.number }} de {{ images.paginator.num_pages }}.
    </span>

    {% if images.has_next %}
      <a href="?page={{ images.next_page_number }}" style="margin-left: 5px; text-
decoration: none; color: #007bff;">Siguiente</a>
      <a href="?page={{ images.paginator.num_pages }}" style="margin-left: 5px; 
text-decoration: none; color: #007bff;">Último &raquo;</a>
    {% endif %}
  </span>
</div>
<!--Estos espacios se agregaron para que el footer no tape los controles de 
paginación  -->
<div>
  <br>
  </br>
  <br>
  </br>
  <br>
  </br>
</div>
</main>

{% endblock %}
```

Por el lado del sistema de login, se definieron las siguientes funciones:

```
def login_view(request):
  if request.method == 'POST':
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)

    if user is not None:
      login(request, user)
```

```
        # Redirige a la página de inicio después del login
        return redirect('home')
    else:
        messages.error(request, 'Usuario o contraseña incorrectos, intente 
nuevamente.')

    return render(request, 'registration/login.html')

def logout_view(request):
    logout(request)
    return redirect('home')
```

Para el login se importaron los módulos de Django contrib.auth y contrib. La función login_view (request) se comunica con la base de datos de SQL y verifica que el usuario esté almacenado. Antes de ello, se tuvo que modificar la linea de código de login.html de la siguiente manera:

```
<form action="{% url 'login' %}" method="POST" style="display: inline-block;">
```

Conjuntamente, en “urls.py”, se agregaron las siguientes rutas:

```
path('login/', views.login_view, name='login'),
path('logout/', views.logout_view, name='logout'),
```

Esto permite ingresar el usuario y la contraseña, que luego es autentincada por la variable “user” si se encuentra en la base de datos. Si el usuario se encuentra logueado lo redirige a home.html y con la sesión abierta. En caso de no encontrarse en la base de datos, vuelve automáticamente a la pestaña de inicio de sesión de login.html. Se intentó implementar un mensaje de error al fallar el inicio de sesión que se encuentra expresado en el código, pero no se encuentra funcional.
Para la implementación del login, hubieron grandes dificultades ya que hubo que investigar desde el principio su lógica de funcionamiento, existiendo varias opciones disponibles que resultaban complejas para su entendimiento.