TP Final – Introducción a la programación

Universidad Nacional de General Sarmiento

Integrantes: Anadón Alejandra, Cazas Kevin y Pogonza Lautaro

Profesores: Argañaraz Omar y Nores Nancy

Comisión – 04

Índice de contenidos

Introducción

El presente trabajo tiene como finalidad la implementación de una aplicación web basada en el framework Django, cuya función principal será permitir a los usuarios realizar búsquedas de Pokémon y visualizar información relevante de cada uno.

Funcionalidad General

La aplicación se encargará de realizar consultas a una **API externa** (PokéAPI), desde donde obtendrá todos los datos necesarios sobre los Pokémon. Una vez recuperada la información, ésta será procesada y renderizada en el frontend utilizando el sistema de plantillas de Django. Los resultados de las búsquedas se mostrarán en tarjetas visuales (cards) dentro de una galería.

Módulos Que Completar

A continuación, se detallan los componentes del sistema que requieren desarrollo o modificación para que la funcionalidad de búsqueda y visualización esté completamente operativa:

views.py

Este módulo debe gestionar las peticiones del usuario (como búsquedas) y coordinar la comunicación con el módulo de servicios que consulta la API. También debe preparar el contexto necesario para renderizar correctamente los resultados en el template.

services.py

Este archivo será responsable de encapsular la lógica de comunicación con la API externa. Deberá implementar funciones que tomen un nombre o ID de Pokémon y devuelvan los datos relevantes en un formato procesable por Django.

home.html (template principal)

El archivo de plantilla necesita ser ajustado para iterar correctamente sobre los datos devueltos por la vista y generar dinámicamente una galería de tarjetas. Cada tarjeta debe mostrar los atributos antes mencionados de forma clara y estilizada.

Objetivo Inmediato:

El objetivo actual es completar la funcionalidad de búsqueda y visualización en la galería, asegurando que, al realizar una consulta, las tarjetas correspondientes con los datos de los Pokémon solicitados se muestren correctamente en la página de inicio.

Views.py

def home

Esta función es la que muestra la página principal. Muestra todas las tarjetas con los Pokémon disponibles. Si el usuario está logueado, también le muestra sus favoritos. Es la primera pantalla que ve cualquier usuario al entrar a la aplicación. También permite que cada persona vea sus favoritos si ya se identificó. Recibe el objeto `request` con la información de la solicitud HTTP, obtiene las imágenes a mostrar, verifica si hay un usuario autenticado y obtiene su lista de favoritos, y finalmente renderiza la plantilla `home.html` enviando estos datos.

```
def home(request):
 images = services.getAllImages()
 favourite_list = []
 if request.user.is_authenticated:
 favourite_list = services.getAllFavourites(request) return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
```

def search

Luego define una vista llamada **search** que maneja la búsqueda de imágenes en la aplicación web. Cuando el usuario realiza una búsqueda se extrae ese texto, se eliminan espacios innecesarios y se utiliza para filtrar imágenes. Si el usuario está autenticado, también se obtiene su lista de favoritos para mostrarlos junto con los resultados. Finalmente, la vista renderiza la página **home.html**, pasando las imágenes filtradas y los favoritos como contexto. Si el usuario no escribe nada, lo redirige de nuevo a la página principal. Es necesario porque esta función permite al usuario buscar imágenes específicas (por ejemplo, de un Pokémon) dentro del sistema

```
def search(request):
name = request.POST.get('query', '').strip()

# si el usuario ingresó algo en el buscador, se deben filtrar las imágenes por dicho ingreso.
if name != '':
images = services.filterByCharacter(name)
favourite_list = []
if request.user.is_authenticated:
favourite_list = services.getAllFavourites(request)

return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
else:
return redirect('home')
```

def filter by type

Después define a una vista llamada **filter_by_type** permite al usuario filtrar imágenes según el tipo de Pokémon seleccionado desde un formulario.

1. **Parámetros:** request: Es el objeto HTTP que envía el navegador al servidor.
2. Desde este objeto se extrae el tipo seleccionado por el usuario en el formulario: request.POST.get('type', '').strip().
3. También permite saber si el usuario está autenticado (request.user.is_authenticated) para obtener su lista de favoritos.
4. Devuelve la plantilla home.html con:
    a. images: lista de imágenes filtradas por el tipo seleccionado.
    b. favourite_list: la lista de favoritos del usuario (si está autenticado).
    c. Si no se seleccionó ningún tipo, se redirige a la página principal (home) para evitar errores o búsquedas vacías.

```
def filter_by_type(request):
type = request.POST.get('type', '').strip()
# debe traer un listado filtrado de imágenes, segun si es o contiene ese tipo.
if type != '':
images = services.filterByType(type)
favourite_list = []
if request.user.is_authenticated:
favourite_list = services.getAllFavourites(request)

return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
else:
return redirect('home')
```

Por último, hay varias funciones que gestionan las acciones relacionadas con los favoritos de un usuario autenticado en una aplicación web. Todas las funciones están protegidas lo que significa que solo pueden accederse si el usuario inició sesión.

def get all favourites by user

La función **getAllFavouritesByUser** obtiene todos los elementos marcados como favoritos por el usuario y muestra esa lista en una plantilla. Es necesario porque permite al usuario ver sus elementos favoritos en una vista dedicada.

```
@login_required
def getAllFavouritesByUser(request):
favourites = services.getAllFavourites(request)
return render(request, 'favourites.html', {'favourites': favourites})
```

def save favourite

La función **saveFavourite** se activa cuando se envía una solicitud POST, lo que indica que el usuario desea agregar un elemento a su lista de favoritos; se llama al servicio correspondiente y luego se redirige a la página principal. Es necesario porque permite que el usuario interactúe con el contenido y marque elementos que desea

guardar para acceder más tarde. Redirige a la página principal (home) después de guardar el favorito.

```
@login_required
def saveFavourite(request):
if request.method == 'POST':
services.saveFavourite(request)
return redirect('home')
```

def delete favourite

La función **deleteFavourite** permite eliminar un favorito del usuario, también mediante una solicitud POST, y luego redirige nuevamente a la página principal. Es necesario porque permite al usuario gestionar sus favoritos y eliminar los que ya no desea mantener. Redirige a la página principal luego de eliminar el favorito.

```
@login_required
def deleteFavourite(request):
if request.method == 'POST':
services.deleteFavourite(request)
return redirect('home')
```

def exit

Esta función es necesaria para que el usuario pueda cerrar su sesión de forma segura.

Recibe el parámetro request y usa la función logout() para cerrar la sesión del usuario autenticado.

Luego, redirige a la página principal (home) para que el usuario salga completamente del sistema.

Es una parte fundamental para manejar correctamente las sesiones de usuario en la aplicación.

```
@login_required
def exit(request):
logout(request)
return redirect('home')
```

Services.py

```python
def getAllImages():

    cards = []
    rawimages = transport.getAllImages()

    for r_image in rawimages:       You, 2 hours ago • probando

        name = r_image.get('name', 'unknown')
        image_url = r_image.get('sprites', {}).get('other', {}).get('official-artwork', {}).get('front_default', '')

        id_ = r_image.get('id', 0)
        height = r_image.get('height', 0)
        weight = r_image.get('weight', 0)
        base_exp = r_image.get('base_experience', 0)
        types = r_image.get('types', [])

        type_names = []

        if types:
            for t in types:
                type_names.append(t['type']['name'])
```

Esta función es la base de la galería. Es necesaria porque permite cargar y mostrar todos los Pokémon cuando el usuario entra por primera vez a la aplicación. No recibe parámetros. Llama internamente a → **transport.getAllImages()** para obtener los datos desde la API, y luego recorre cada Pokémon para armar una tarjeta con su nombre, tipo, imagen y el ícono del tipo. En esta variable obtenemos una lista de 29 datos en formato Json por cada Pokémon basándose en su id para identificarlos dentro de la API.

Continúa con:

```python
        type_names = []

        if types:
            for t in types:
                type_names.append(t['type']['name'])

        type_id = type_names[0] if type_names else 'unknown'

        type_icon_url = transport.get_type_icon_url_by_id(type_id)

        card = {
            'name': name,
            'image': image_url,
            'id': id_,
            'base': base_exp,
            'weight': int(weight / 10),  # Convierte hectogramos a kilogramos y redondea al entero
            'height': height,
            'types': type_names,
            'type': type_id,
            'type_icon_url': type_icon_url
        }

        cards.append(card)

    return cards
```

Debemos ingresar datos como nombre, imagen del Pokémon, tipo del mismo e icono del tipo, etc. Si alguno de los datos que pedimos está vacío ponemos una opción tanto vacía como de unknown así no genera errores.

Empleamos la función **get_type_icon_url_by_id** para obtener el ícono según el type de Pokémon. Posteriormente se genera la card con la recolección de los datos obtenidos.

El return es una lista de cards armadas por cada Pokémon.

```
41  # función que filtra según el nombre del pokemon.
42  def filterByCharacter(name):
43      filtered_cards = []
44
45      for card in getAllImages():
46          if name.lower() in card['name'].lower():
47              filtered_cards.append(card)
48
49      return filtered_cards
```

La función **filterByCharacter** recibe como parámetro el nombre del pokémon tal como lo ingresa el usuario en la barra de búsqueda. Es decir, recibe el texto de manera explícita. Para facilitar la búsqueda ".lower". Esta función es necesaria para implementar la barra de búsqueda por nombre. Recibe como parámetro una cadena de texto (**name**) ingresada por el usuario y compara ese texto con los nombres de los Pokémon disponibles.

A continuación se recorre con un ciclo for las cards y se evalúa si el nombre ingresado corresponde al nombre de la card. De ejecutarse la rama verdadera se agrega a la lista y se devuelve con el return.

```
51  # función que filtra las cards según su tipo.
52  def filterByType(type_filter):
53      filtered_cards = []
54
55      for card in transport.getAllImages():
56          if type_filter.lower() in card['type'].lower():
57              filtered_cards.append(card)
58
59      return filtered_cards
```

Se repite el comportamiento pero cambia el parámetro que recibe la función. Ya que se filtra por el tipo de pokémon, y se emplea un string para clasificarlos.

Al igual que en la función que filtra por nombre, se recorre con un for la card y evalúa con if si el tipo del pokémon ingresado está dentro de la misma.

```python
# añadir favoritos (usado desde el template 'home.html')
def saveFavourite(request):
    fav = translator.fromRequestIntoCard(request.POST)
    fav.user = get_user(request) # le asignamos el usuario correspondiente.

    return repositories.save_favourite(fav) # lo guardamos en la BD.

# usados desde el template 'favourites.html'
def getAllFavourites(request):
    if not request.user.is_authenticated:
        return []
    else:
        user = get_user(request)

        favourite_list = repositories.get_favourites_by_user(user)
        mapped_favourites = []

        for favourite in favourite_list:
            card = translator.fromTemplateIntoCard(favourite)
            mapped_favourites.append(card)

        return mapped_favourites

def deleteFavourite(request):
    favId = request.POST.get('id')
    return repositories.delete_favourite(favId) # borramos un favorito por su ID
```

En la función **saveFavourite** se le da el parámetro de request, que luego se traduce a una card por medio del **translator**. Se emplea POST como una forma de enviar datos al servidor de la app. El return de la función es utilizado cuando un usuario agregue favoritos desde el sector home.

En el caso de la función **getAllImages(request)**, si el usuario no se encuentra logueado devuelve una lista vacía. Mientras que **get_favourites_by_user(user)** toma como parámetro el usuario y busca sus favoritos para guardarlos en una lista.

fromTemplateIntoCard(favourite) convierte cada favorito en una card y devuelve una lista de cards de los pokémons favoritos del usuario.

Finalmente la función **deleteFavourite (request)** toma el id del favorito guardado desde el formulario enviado al servidor (POST) y lo elimina.

home.html

```
<div class="col">
    <!-- evaluar si la imagen pertenece al tipo fuego, agua o planta -->
    <div class="card mb-3 ms-5
        {% if img.types and 'fire' in img.types %}
            border-danger
        {% elif img.types and 'water' in img.types %}
            border-primary
        {% elif img.types and 'grass' in img.types %}
            border-success
        {% else %}
            border-orange
        {% endif %}"

        style="width:800px; height: 350px;">
```

En este bloque de plantilla era necesario agregar una condición para evaluar el tipo de Pokemon. También definimos en style="width:800px; height: 350px;"> el alto y el ancho de la tarjeta.

- Si es de tipo **fuego**, aplica **border-danger** (borde rojo).
- Si es de tipo **agua**, aplica **border-primary** (borde azul).
- Si es de tipo **planta**, **aplica border-success** (borde verde).
- Si no es ninguno de esos, aplica un borde **naranja personalizado** (**border-orange**)

{% else %}

border-orange → aplicar borde naranja a todos los pokemons que no sean agua, fuego ni planta.

Como Bootstrap solo ofrece clases como border-danger (rojo), border-primary (azul), border-success (verde),etc, fue necesario crear manualmente un una clase personalizada para el color naranja para cumplir con la consigna. Para ello, se agregó el siguiente código:

```
143
144  .border-orange {
145      border: 1px solid orange !important;
146  }
```

Esto fue agregado en styles.css ya que no puede ser creado directamente en el html porque no funciona con {% if %} de Django, ya que este que espera class="..."

Border

La propiedad border permite definir el **tamaño**, el **estilo** y el **color** del borde. Se utiliza 1px porque es el grosor que utiliza Bootstrap utiliza 1px para sus bordes y es necesario que sean iguales todos y mantener la coherencia visual.

!important

Utilizamos !important para asegurar que esta regla tenga prioridad sobre otras posibles reglas de Bootstrap y en caso de conflicto tenga prioridad.

Como se puede ver a continuación fue necesario hacer ligeros ajustes para que la imagen estuviese a la izquierda y los datos a la derecha. Asimismo, cambiamos algunos valores para que se vean bien visualmente los datos dentro del contenedor (la card).

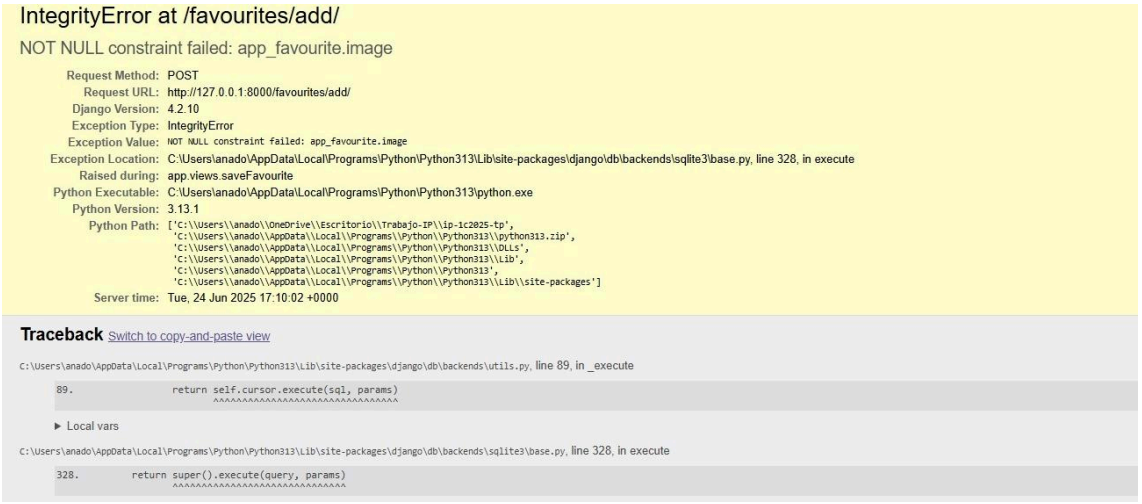
```
v class="row g-0 h-100 align-items-center"> <!--dividir la tarjeta en 2 columnas y h-100 para que ocupe el 100%  de la altu
 <div class="col-md-4 d-flex align-items-center justify-content-center">
     <img src="{{ img.image }}" class="card-img-top"alt="imagen"style="max-width: 100%; height: auto; object-fit: contain;"
 </div>

<div class="col-md-8 d-flex flex-column h-100">
    <div class="card-body" style="overflow-y: auto; flex-grow: 1;">
```

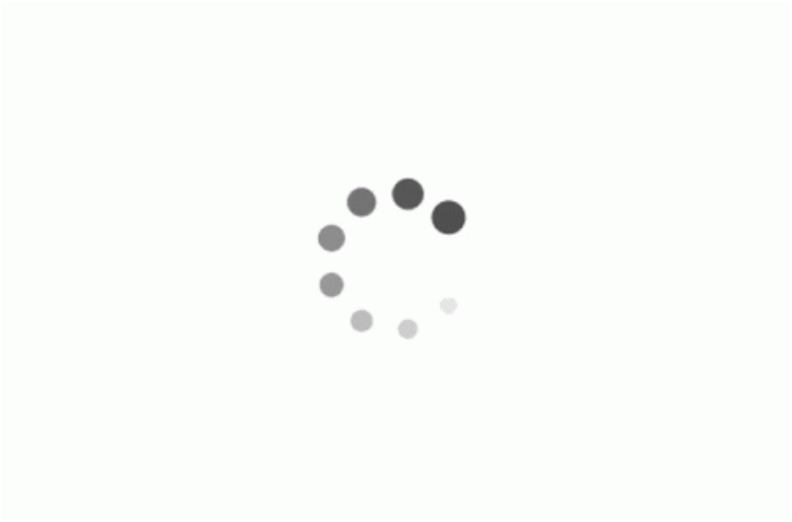
- **row**: crea una fila de Bootstrap para dividir en columnas.
- **g-0**: elimina el espacio entre las columnas.
- **h-100**: hace que la fila ocupe el 100% de la altura de su contenedor padre.
- **align-items-center**: centra verticalmente el contenido dentro de la fila.
- **src="{{ img.image }}"**: carga la imagen.
- **max-width: 100% y height: auto**: ajusta el tamaño de la imagen sin deformarla.
- **object-fit: contain**: asegura que la imagen se muestre completa, sin recortes, dentro de su contenedor.
- **card-body**: parte estándar de una tarjeta de Bootstrap para contener el contenido textual.
- **flex-grow: 1**: hace que este bloque se expanda para ocupar el espacio disponible dentro de la columna.

Observaciones:

- Encontramos dificultad con el botón de favoritos, no pudimos detectar a qué se debía el error, por lo cual no logramos darle funcionalidad. Ya que al interactuar nos lanzaba el siguiente error:



- Por otra parte, no alcanzamos a agregar el Loading Spinner para la carga de imágenes que se encontraba dentro de los opcionales del trabajo. Lo cual nos hubiera gustado sumar a la app.



- Como no sabiamos como crear la parte de registrarse encontramos la solución de crear un superusuario con el siguiente comando: python manage.py createsuperuser

  usuario= IP

  contraseña = 417

Conclusión:

El trabajo se presenta interesante para la práctica de la implementación de diferentes utilidades dentro de los lenguajes vistos. Pero a la vez, se ve un cierto nivel de dificultad en cuanto a lo que involucra html, json, etc. Ya que no son elementos vistos en

profundidad en la cursada. Cabe aclarar que esto no fue un impedimento, sino más bien una primera instancia de familiarización con el trabajo que demandó más atención.