TRABAJO PRÁCTICO

Aplicación Web – Buscador de Personajes de Los Simpsons

Tecnología: Django (Python)

Integrantes: Antonella Miño - Cristian de Barrios - Daiana Arce Torres

---

1. INTRODUCCIÓN

El presente trabajo práctico consiste en el desarrollo de una aplicación web utilizando el framework Django, cuyo objetivo principal es consumir una API externa de personajes de *Los Simpsons* y mostrar la información obtenida en forma de galería interactiva.

El problema a resolver implicó integrar distintas capas de una aplicación web, permitiendo:

- Obtener información desde una API externa.
- Renderizar imágenes y datos relevantes en una galería.
- Implementar filtros por nombre y estado.
- Gestionar favoritos asociados a usuarios autenticados.
- Aplicar estilos dinámicos según el estado del personaje.

La solución implementada respeta una, favoreciendo la separación de responsabilidades y garantizando mayor mantenibilidad y claridad estructural.

TRABAJO PRÁCTICO

2. DESARROLLO.

Primera Función.

```python
def getAllImages():
    """
    Obtiene todas las imágenes de personajes desde la API y las convierte en objetos Card.

    Esta función debe obtener los datos desde transport, transformarlos en Cards usando
    translator y retornar una lista de objetos Card.
    """
    ##Esta es la primera función implementada, es funadamental para obtener las cards, es la base de todo
    json_collection = transport.getAllImages()
    cards = []

    for character in json_collection:
        card = translator.fromRequestIntoCard(character)
        cards.append(card)

    return cards
```

getAllImages() se encarga de obtener todas las imágenes de personajes desde la API, transformar esos datos crudos en objetos Card, y devolverlos en una lista lista para usar en la aplicación.

json_collection = transport.getAllImages()

En esta línea llamó al módulo transport, que es el encargado de comunicarse con la API. getAllImages() devuelve una colección de datos en formato JSON, es decir, información cruda tal como viene del backend.

Luego generamos una lista vacía cards = [] para ir guardando los objetos ya transformados.

```python
for character in json_collection:
        card = translator.fromRequestIntoCard(character)
        cards.append(card)


    return cards
```

Luego el For va recorriendo cada elemento de la colección JSON. Y character representa el personaje individual que va obteniendo esa variable. Luego usamos translator para poder transformar los datos crudos en cards.
Y por último, una vez obtenida la card, la vamos agregando a la lista vacía, para que luego la retorne completa.

Segunda Función.

TRABAJO PRÁCTICO

```python
def filterByCharacter(name):
    """
    Filtra las cards de personajes según el nombre proporcionado.

    Se debe filtrar los personajes cuyo nombre contenga el parámetro recibido. Retorna una lista de Cards f
    """
    ##2da función implementada, filtro de busqueda por personaje
    all_cards = getAllImages()
    filtered_cards = []

    for card in all_cards:
        if name.lower() in card.name.lower(): #comparo ambas variables con el mismo formato
            filtered_cards.append(card)

    return filtered_cards
```

La función filterByCharacter(name) sirve para que reciba un parámetro , en este caso lo nombramos *name*, que es el texto que quiero usar para filtrar los personajes.

Luego en una variable llamó a la función anterior getAllImages() para poder obtener todas las card ya transformadas desde la api.

Creamos una lista vacía para poder guardar las cards que cumplan con el filtro.

```python
for card in all_cards:
        if name.lower() in card.name.lower(): #Case insensitive
            filtered_cards.append(card)
```

Con el For recorremos la lista de las cards transformadas.
Luego utilizamos un condicional pero le sumamos .lower() para que los textos se conviertan en minúsculas y no haya errores con las letras. Esta condición pregunta si lo que escribió el usuario es similar a lo que hay dentro de la variable card. Esto es para comparar las dos variables en el mismo formato.

TRABAJO PRÁCTICO

Tercera Función

```python
# 3ra función implementada, filtro status
def filterByStatus(status_name):
    """
    Filtra las cards de personajes según su estado (Alive/Deceased).

    Se deben filtrar los personajes que tengan el estado igual al parámetro 'status_name'. Retorna una list
    """
    all_cards = getAllImages()
    filtered_cards = []

    for card in all_cards:
        if card.status.lower() == status_name.lower():
            filtered_cards.append(card)

    return filtered_cards
```

La función filterByStatus(status_name) sirve para filtrar a los personajes según su estado (“Alive” o “Deceased”), devolviendo únicamente los que cumplan con el estado deseado. Primero recibe el parámetro status_name, que representa el estado por el cual se quiere filtrar;

```
all_cards = getAllImages()
filtered_cards = []
```

Luego, se llama a la primera función getAllImages() explicada e implementada anteriormente, para obtener la lista completa de cartas. Después crea una lista vacía llamada filtered_card = [], donde se guardaran las cartas que cumplan con la condición.

Dentro del for se recorre cada una de las cartas y se compara el status con el parámetro recibido, la comparación es insensible a mayúsculas y minúsculas. Si la carta cumple con la condición, se agrega a la lista filtered_card = [].

Una vez terminado el recorrido, la función retoma la lista con los personajes filtrados.

TRABAJO PRÁCTICO

Cuarta Función

```python
def home(request):
    #Esta función...#
    images = services.getAllImages()
    favourite_list = services.getAllFavourites(request)

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

Esta función es la que arma la pantalla principal. Pide los datos al servicio y los envia al template home.html para que se muestre en pantalla.

images = services.getAllImages() ---> Esta variable trae una lista de personajes

favourite_list = services.getAllFavourites(request)

Con esta variable obtengo la lista de favoritos del usuario actual usando el *request*, que me permite saber quién es el usuario.

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

Esto retorna varias cosas juntas:
*Usa el template home.html donde le pasa los datos. y luego devuelve la página al navegador. (En síntesis, Cuando el usuario entra a la página, esta función va a buscar los personajes y los favoritos, y se los pasa al HTML para que los muestre.)

TRABAJO PRÁCTICO

Quinta Función

```python
#5ta función invoco services.py para realizar la busqueda de personajes
def search(request):
    """
    Busca personajes por nombre.

    Se debe implementar la búsqueda de personajes según el nombre ingresado.
    Se debe obtener el parámetro 'query' desde el POST, filtrar las imágenes según el nombre
    y renderizar 'home.html' con los resultados. Si no se ingresa nada, redirigir a 'home'.
    """
    if request.method == "POST":
        query = request.POST.get("query")

        if not query:
            return redirect('home')

        images = services.filterByCharacter(query)
        favourite_list = services.getAllFavourites(request)

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

    return redirect('home')
```

La función search(request) se encarga de gestionar la búsqueda de personajes desde la acción que hace el usuario en pantalla.

```
if request.method == "POST":
        query = request.POST.get("query")
```

Primero, la función recibe como parámetro request, que contiene la información de la solicitud HTTP hecha por el usuario, se verifica si el método de la solicitud es “POST”, obteniendo el valor ingresado en el campo “query” del formulario.
Si el campo está vacío, se redirige al usuario a la página principal para evitar realizar una búsqueda sin datos. En caso contrario, la función no realiza el filtrado directamente, sino que delega la lógica a la capa de servicios. Además de obtenerse la lista de favoritos del usuario con services.getAllFavourites(request)

Finalmente, se envían al template los resultados filtrados y la lista para que sean mostrados en la interfaz. Si la solicitud no es de tipo "POST", la función redirige nuevamente a la vista principal.

Sexta Función

TRABAJO PRÁCTICO

```python
#6ta función invoco services.py y verificar su status
def filter_by_status(request):
    """
    Filtra personajes por su estado (Alive/Deceased).

    Se debe implementar el filtrado de personajes según su estado.
    Se debe obtener el parámetro 'status' desde el POST, filtrar las imágenes según ese estado
    y renderizar 'home.html' con los resultados. Si no hay estado, redirigir a 'home'.
    """
    if request.method == "POST":
        status = request.POST.get("status")

        if not status:
            return redirect('home')

        images = services.filterByStatus(status)
        favourite_list = services.getAllFavourites(request)

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

    return redirect('home')
```

La función solicita al service que traiga solo los personajes que tengan ese estado (Alive o Deceased)

Todo lo que devuelve filterByStatus(status) se guarda en la variable images. Básicamente esto me trae solo los personajes que coincidan con el estado elegido.

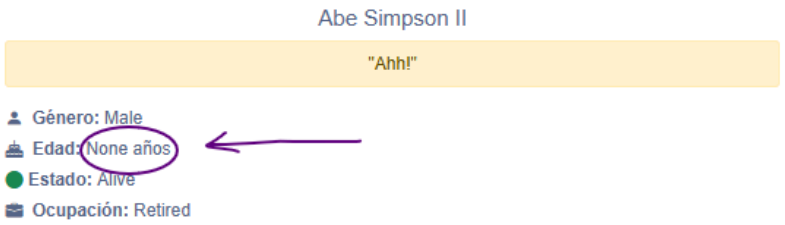
Adicional: Aquí encontramos un problema que rompe el guardado del favorito con age. Cuando la edad no es numérica el post es = none.

TRABAJO PRÁCTICO

```python
#def fromTemplateIntoCard(templ):
#   card = Card(
#       name=templ.POST.get("name"),
#       gender=templ.POST.get("gender"),
#       status=templ.POST.get("status"),
#       phrases=templ.POST.get("phrases"),
#       occupation=templ.POST.get("occupation"),
#       image=templ.POST.get("image"),
#       age=templ.POST.get("age")
#   )
#    return card

def fromTemplateIntoCard(templ):
    raw_age = templ.POST.get("age")

    # Limpieza del dato
    try:
        age = int(raw_age) if raw_age not in (None, "", "None") else None
    except ValueError:
        age = None
```



ADICIONAL: Sumamos un if para que el icono del estado cambie de color según este vivio o fallecido.

```
+                <p class="card-text mb-1"> <!--el icono del estado cambia de color según vivo o fallecido-->
+                    <strong>
+                        {% if img.status == 'Alive' %}
+                            <i class="bi bi-circle-fill text-success"></i>
+                        {% elif img.status == 'Deceased' %}
+                            <i class="bi bi-circle-fill text-secondary"></i>
+                        {% else %}
+                            <i class="bi bi-circle-fill text-warning"></i>
+                        {% endif %}
+                        Estado:
+                    </strong>
+                    {{ img.status }}
```

TRABAJO PRÁCTICO

3. CONCLUSIÓN

Desde el comienzo y durante toda la etapa del desarrollo nos encontramos con muchos desafíos relacionados a:

- Comprensión de la arquitectura en capas (views, services, persistence, transport y utilities) y cómo se comunican entre sí.

- Aprender sobre el manejo del flujo de datos desde la API hasta el template, entendiendo cómo transformar la información en objetos (Card) antes de renderizarlos.

- Errores de template en Django, como bloques mal cerrados (if, for) y condicionales incorrectos.

Integración con GitHub, especialmente problemas de autenticación (uso de token en lugar de contraseña y permisos 403).

Uso correcto del grid de Bootstrap, para lograr una visualización ordenada de las cards en filas de tres columnas.

Durante la etapa del desarrollo del grid surgieron problemas relacionados con la estructura del HTML y el uso correcto del sistema de grillas de Bootstrap.
Uno de los inconvenientes fue la mala anidación de etiquetas (div), lo que generó:

- Cards mal alineadas
- Bordes que no cerraban correctamente
- Diseño que no respetaba las 3 columnas esperadas
- Comportamiento visual diferente al previsto

También se detectó duplicación innecesaria de componentes (card dentro de card), lo que afectaba la estructura del grid.

Para resolverlo, se reutilizó el Código inviado por el profesor, y para resolver y mejorar las vistas, iconos, etc, se importó en home.html, la librería: <link rel="stylesheet"

TRABAJO PRÁCTICO

href="https://cdn.jsdelivr.net/npm/bootstrap-icons@1.11.1/font/bootstrap-icons.css">

Por otro lado, en la implementación de la funcionalidad de favoritos, nos encontramos con distintos errores relacionados principalmente con rutas, vistas, templates y manejo de formularios.

Tambien surgieron inconvenientes en la configuración de URLs y en la correcta conexión entre vistas y templates, lo que generó errores como TemplateDoesNotExist. Esto nos obligó a revisar la estructura del proyecto y comprender la importancia de mantener coherencia entre archivos y nombres definidos.

Además enfrentamos dificultades al implementar la lógica para agregar y eliminar favoritos, especialmente en el envío de datos mediante formularios POST y en la correcta identificación del usuario logueado. Revisando los mensajes de error y el traceback que provee Django, más ayuda de IA, pudimos detectar rápidamente dónde estaba el problema y corregirlo.

Como aprendizaje principal, entendimos que interpretar correctamente los errores, focalizarse y ejecutar buenas prácticas como mantener una estructura ordenada, validar cada cambio de manera incremental y probar cada funcionalidad antes de continuar, es fundamental para no perder el hilo del desarrollo.

En conclusión, el proceso no solo mejoró la aplicación, sino que también fortaleció nuestra capacidad de análisis y resolución de problemas dentro del framework Django.

Fuente: https://github.com/ungs-ip/ip-verano-2026-alumnos