Universidad Nacional de General Sarmiento

Licenciatura en Sistemas

Trabajos Práctico
POKEDÉX

Introducción a la Programación
(1 - 2025)

El presente trabajo consistió en el desarrollo de una aplicación web llamada Pokédex, que permite visualizar una galería de Pokémon obtenidos desde una API externa. Los usuarios pueden buscar Pokémon por nombre, filtrarlos por tipo (agua, fuego o planta) y marcarlos como favoritos si están registrados. También se incorporó un sistema de autenticación de usuarios, así como un diseño visual acorde al universo Pokémon. Durante el desarrollo se resolvieron desafíos como el manejo de errores al seleccionar duplicados, el almacenamiento de favoritos por usuario, y la integración fluida entre la interfaz y la lógica interna

-Integrantes: Abréu Kaira 96.005.206

1. Introducción Este trabajo tuvo como objetivo desarrollar una Pokédex interactiva: una aplicación pensada para que cualquier persona pueda explorar y coleccionar Pokémon de manera simple y entretenida. Al ingresar, se presenta una galería de tarjetas con los Pokémon disponibles, con opciones para buscarlos por nombre o filtrarlos por tipo. Si el usuario está registrado, puede marcar sus Pokémon favoritos y consultarlos luego en una sección especial. La idea fue brindar una experiencia accesible e intuitiva, inspirada en el espíritu del mundo Pokémon, donde explorar criaturas y armar una colección es parte del juego.

El objetivo fue crear una experiencia divertida e intuitiva, parecida a la idea del universo Pokémon, donde uno puede explorar criaturas y armar su colección.

2. Desarrollo

2.1 Descripción general: Se implementaron funciones clave en los archivos views.py, services.py, home.html y otros. Cada función fue diseñada para cumplir una tarea específica del flujo general: obtener y mostrar imágenes, permitir búsqueda y filtrado, gestionar favoritos y registrar usuarios. Se reutilizó código y se respetó la separación por capas.

2.2 Funcionalidades principales:

Inicialmente se proporcionó las siguientes funciones:

1) def home (request):

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

Como los listados estan vacios e incompleto el desarrollo no cumple su propósito.

Es por ello que se restructuró la función quedando de la siguiente manera:

```python
def home(request):

    images = services.getAllImages()

    if request.user.is_authenticated:
        favourite_list = services.getAllFavourites(request)

        favourite_names = [fav.name for fav in favourite_list]
    else:
        favourite_list = []
        favourite_names = []

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

-Se obtienen las imágenes al llamar a **services.getAllImages(),** el cual es un método en la capa de servicios que centraliza la lógica de obtención de datos (desde la API).
-Se verifica si el usuario esta logueado, y si esta logueado llama a **services.getAllFavourites(request),** otra función de la capa de servicios.
-Por último, Envía los datos (images, favourite_list, favourite_names) a home.html para que el diseño se encargue de mostrarlos.

2) def getAllImages():

debe ejecutar los siguientes pasos:
1) traer un listado de imágenes crudas desde La API (ver transport.py)

2) convertir cada img. en una card.
3) añadirlas a un nuevo listado que, finalmente, se retornará con todas las card encontradas.

Teniendo presente los requerimientos, se desarrolló la siguiente función:

```
def getAllImages():

    raw_list = transport.getAllImages()
    card_list = []
    for raw in raw_list:
        card = translator.fromRequestIntoCard(raw)
        card_list.append(card)
    return card_list
```

-Toma una lista de imágenes en su forma original (raw_list) obtenidas desde el servicio transport.getAllImages().
-card_list = [] Se crea una lista vacía que almacenará las tarjetas convertidas.
-Se recorre cada elemento de la lista cruda.
-translator.fromRequestIntoCard(raw) transforma cada dato crudo en un objeto Card. Luego, se agrega a la lista card_list.
-Por último, devuelve la lista completa de tarjetas ya convertidas.

(3) home.html:
La card debe cambiar su border color dependiendo de los tipos más icónicos, fuego, agua y planta.
Para la resolución de este ítem se desarrolló el siguiente código en la vista home:

```
{% for img in images %}
    <div class="col">
        <div class="card mb-3 ms-5
            {% if 'fire' in img.types %}
                border border-3 border-danger
            {% elif 'water' in img.types %}
                border border-3 border-primary
            {% elif 'grass' in img.types %}
                border border-3 border-success
            {% else %}
                border border-3 border-warning
            {% endif %}
            pokemon-card
        ">
```

-Se itera sobre la lista de imágenes.
-Para cada img, genera una tarjeta (card) dentro de un contenedor (col).

-Se usa una estructura condicional {% if … elif … else %} para añadir clases CSS (Bootstrap) según el tipo:

Si el Pokémon tiene 'fire' en img.types, usa border-danger.
Si tiene 'water', usa border-primary.
Si tiene 'grass', usa border-success.
En cualquier otro caso, usa border-warning.

Esto aplica un borde de color a la tarjeta, relacionándolo visualmente con el tipo del Pokémon.

2.3 Funcionalidades extras:

Extra: Buscador I (según el nombre del pokémon) y Buscador II (filtro según tipo fuego, agua o planta)

Para el desarrollo de esta funcionalidad se terminó de implementar en services.py las funciones de filtrado por nombre y la de filtro por tipo.

```python
def filterByCharacter(name):

    name_lower = name.strip().lower()
    if not name_lower:

        return getAllImages()
    filtered_cards = []
    for card in getAllImages():
        if name_lower in card.name.lower():
            filtered_cards.append(card)
    return filtered_cards

# función que filtra las cards según su tipo.
def filterByType(type_filter):
    filtered_cards = []
    filter_lower = type_filter.lower()
    for card in getAllImages():

        for t in card.types:
            if t.lower() == filter_lower:
                filtered_cards.append(card)
                break

    return filtered_cards
```

filterByCharacter(): Convierte el nombre ingresado a minúsculas y le quita espacios: esto evita que errores de mayúsculas o espacios afecten la búsqueda.
-Si el usuario no ingresó texto, devuelve todas las imágenes.
-Filtra todas las tarjetas (card) cuyo nombre contenga el texto buscado, sin importar mayúsculas.

filterByType(): Recibe un tipo (por ejemplo, "fire", "grass", etc.).
-Filtra las tarjetas donde alguno de sus tipos (card.types) coincide con el tipo solicitado, sin importar mayúsculas.
-Usa break para no seguir iterando si ya encontró una coincidencia.
También se termino de implementar en views.py

```
def search(request):

  if request.method == 'POST':
    name = request.POST.get('query', '').strip()
    images = services.filterByCharacter(name)

    if request.user.is_authenticated:
        favourite_list = services.getAllFavourites(request)
    else:
        favourite_list = []

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

Esta función verifica si la solicitud es POST (proviene de un formulario).
-Obtiene lo que escribió el usuario en el campo query, lo limpia de espacios y lo convierte a minúsculas.
-Filtra las tarjetas de Pokémon usando filterByCharacter(name); si no se escribió nada, devuelve todas.
-Obtiene la lista de favoritos del usuario si está logueado; si no, devuelve una lista vacía.
-Renderiza la plantilla home.html, enviando las imágenes filtradas y la lista de favoritos.
-Si la solicitud no es POST, redirige a la página principal (home)

fiterByType():recibe la solicitud HTTP (request).
-Obtiene el valor del campo type enviado por POST y le aplica .strip() para eliminar espacios.
-Verifica si se ingresó un tipo (es decir, si type no está vacío).
-Si hay un tipo:
Llama a services.filterByType(type) para obtener solo las tarjetas que tengan ese tipo.
Verifica si el usuario está autenticado.
Si está autenticado, obtiene su lista de favoritos con services.getAllFavourites(request).
Si no está autenticado, usa una lista vacía de favoritos.
-Renderiza la plantilla home.html con:
la lista de imágenes filtradas por tipo.
la lista de favoritos del usuario (si está logueado).
-Si no se ingresó ningún tipo:
Redirige al usuario a la vista home

```python
def filter_by_type(request):
    type = request.POST.get('type', '').strip()

    if type:
        images = services.filterByType(type)
        favourite_list = services.getAllFavourites(request) if request.user.is_authenticated else []

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

Extra Favoritos

En services.py se termino de implementar las siguientes funciones

```python
def saveFavourite(request):
    fav = translator.fromTemplateIntoCard(request)
    fav.user = get_user(request)


    existing = repositories.find_favourite_by_name_and_user(fav.name, fav.user)
    if existing:
        raise Exception("Ya está en favoritos")

    return repositories.save_favourite(fav)
```

saveFavourites(): Convierte los datos del request en una tarjeta (Card) usando translator.fromTemplateIntoCard.
-Asigna al favorito el usuario actual con get_user(request).
-Verifica si ya existe ese favorito para ese usuario llamando a repositories.find_favourite_by_name_and_user(...).
-Si ya existe, lanza una excepción con el mensaje "Ya está en favoritos".
-Si no existe, lo guarda con repositories.save_favourite(fav) y lo retorna.

```python
def getAllFavourites(request):

    if not request.user.is_authenticated:
        return []
    else:
        user = get_user(request)

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

        for fav_dict in favourite_list:
            card = translator.fromRepositoryIntoCard(fav_dict)
            mapped_favourites.append(card)

        return mapped_favourites
```

getAllFavourites(): Si el usuario no está autenticado, devuelve una lista vacía.
-Si está autenticado:

Obtiene al usuario con get_user(request).
Recupera sus favoritos como diccionarios desde repositories.get_all_favourites(user).
Convierte cada diccionario en una tarjeta (Card) con translator.fromRepositoryIntoCard.
Los agrega a una lista llamada mapped_favourites.
Devuelve esa lista de tarjetas.

```python
def deleteFavourite(request):
    favId = request.POST.get('id')
    return repositories.delete_favourite(favId)
```

deleteFavourite(): Obtiene el id del favorito desde request.POST.get('id').
-Llama a repositories.delete_favourite(favId) para eliminarlo.

En views.py se desarrolló:

```python
def agregar_favorito(request):
    try:
        services.saveFavourite(request)

    except Exception as e:
        messages.warning(request, str(e))
    return redirect('home/')
```

agregar_favorito(): Intenta ejecutar services.saveFavourite(request) para guardar el Pokémon como favorito.

-Si no hay errores, continúa (podrías mostrar un mensaje de éxito, pero está comentado).
-Si ocurre una excepción (por ejemplo, si el Pokémon ya está en favoritos):
-Muestra un mensaje de advertencia con el texto del error usando messages.warning.
-Finalmente, redirecciona al usuario a la página principal ('home/').

ALTA de nuevos usuarios

```python
def register(request):
    first_name = request.POST.get('first_name')
    last_name = request.POST.get('last_name')
    username = request.POST.get('username')
    email = request.POST.get('email')
    password = request.POST.get('password')
    if not all([first_name, last_name, username, email, password]):
        return render(request, 'register.html', {'error': 'Todos los campos son obligatorios.'})

    if User.objects.filter(username=username).exists():
        return render(request, 'register.html', {'error': 'Ese nombre de usuario ya existe.'})

    user = User.objects.create_user(
        username=username,
        email=email,
        password=password,
        first_name=first_name,
        last_name=last_name
    )
    user.save()
    subject = 'Registro exitoso en la App Pokémon'
    message = f"Hola {first_name},\n\nTu cuenta fue creada exitosamente.\n\nUsuario: {username}\nContraseña: {password}\n\n¡Bienvenido!"
    from_email = settings.DEFAULT_FROM_EMAIL
    recipient_list = [email]
    try:
        send_mail(subject, message, from_email, recipient_list)
    except Exception as e:
        print(f"Error al enviar el correo: {e}")
    return redirect('login')

return render(request, "register.html")
```

-Verifica que la solicitud sea un POST (se envió el formulario de registro).
-Obtiene del formulario estos campos: first_name, last_name, username, email, password.
-Valida que todos los campos estén completos; si alguno falta, renderiza de nuevo register.html con un error.
-Comprueba que el nombre de usuario no exista usando User.objects.filter(username=username).exists().
-Si ya existe, muestra un error en register.html.
-Crea el usuario con User.objects.create_user(...)
-Genera un correo de bienvenida con asunto, mensaje (incluye usuario y contraseña), remitente (DEFAULT_FROM_EMAIL) y destinatario (email).
-Envía el correo usando send_mail(...); si falla, lo informa por consola
-Redirige al usuario a la página de login (redirect('login')).

-Si se accede con GET (no envían datos), renderiza el formulario vacío de registro (register.html).

En el frontend se creo el archivo register.html que presenta el formulario al usuario para su registro.

Función	Parámetros que recibe	Devuelve / Modifica
`home(request)`	`request` : objeto `HttpRequest`	Renderiza `home.html` con: `images` , `favourite_list` , `favourite_names`
`getAllImages()`	No recibe	Lista de objetos `Card` construidos desde la API
`filterByCharacter(name)`	`name` : texto ingresado	Lista de `Card` que contienen el texto en su nombre
`filterByType(type)`	`type` : tipo de Pokémon (fire, water, etc.)	Lista de `Card` que contengan ese tipo
`getAllFavourites(request)`	`request` : usado para identificar al usuario	Lista de favoritos (`Card`) o lista vacía si no está logueado
`saveFavourite(request)`	`request` : contiene datos del Pokémon y del usuario	Devuelve el favorito guardado. **Modifica:** guarda en la base de datos
`deleteFavourite(request)`	`request` : contiene el ID del favorito a eliminar	No devuelve nada. **Modifica:** elimina el favorito en la base de datos
`agregar_favorito(request)`	`request` : con datos del Pokémon	Redirige a `home/` . **Modifica:** guarda el favorito si no existe
`register(request)`	`request` : con datos del formulario de registro	Redirige al login o muestra errores. **Modifica:** crea un nuevo usuario y envía email

Renovar interfaz gráfica: Se propuso una mejor interfaz gráfica relacionada al mundo pokemón, utilizando los colores característicos del mismo, para ello se usó código CSS y la librería Bootstrap.

3. Conclusiones: Durante el desarrollo de este proyecto enfrenté varios desafíos que me ayudaron a aprender tanto del lenguaje como de la lógica de desarrollo. Al principio, me costó comprender el flujo general del proyecto, ya que abarcaba varias capas (vistas, servicios, plantillas) y no tenía del todo claro cómo interactúan entre sí. Otro inconveniente importante fue que me quedé sin grupo de trabajo, por lo que tuve que enfrentar el desarrollo de manera individual. Esto implicó tomar decisiones por mi cuenta y resolver problemas sin tener a alguien con quien dividir tareas o debatir ideas.

Además, varios temas involucrados (como el manejo de vistas en Django, el trabajo con formularios o el uso de servicios externos como APIs) requerían conocimientos más profundos de los que tenía, por lo que debí investigar bastante por mi cuenta. Consulté documentación oficial, foros y tutoriales para poder implementar funcionalidades como el registro de usuarios, los filtros de búsqueda y el sistema de favoritos.

A pesar de estos obstáculos, se logró un sistema funcional, agradable y con buena interacción, cumpliendo el objetivo de poder explorar y coleccionar Pokémon desde una Pokédex personalizada. Considero que el proyecto me permitió afianzar conceptos clave de desarrollo web, aprender a resolver errores de forma autónoma y organizar mejor el código respetando la separación por capas.

Anexo:

https://medium.com/%40kkarann07/session-authentication-in-django-35536af29187
https://realpython.com/django-templates-tags-filters/?utm_sourc
https://stackoverflow.com/questions/14412752/django-applying-a-style-class-based-on-a-conditional?utm_source
https://docs.djangoproject.com/en/5.2/topics/forms/
https://docs.djangoproject.com/en/5.2/topics/auth/default/?utm_source
https://chatgpt.com/