Universidad Nacional de General Sarmiento

Licenciatura en Sistemas

POKEDÉX

Introducción a la Programación

(1er semestre, año 2025)

Resumen: El trabajo consiste en construir una aplicación web interactiva, **organizada por capas de trabajo**, usando la herramienta Django para consultar una API e interactuar con el usuario.

Integrantes: Balverdi, Enzo Nicolás (*exnxb021@gmail.com)*

Monteros, Tobías Agustín (*tobimonteros@gmail.com*)

1. Introducción:

Al interactuar con una aplicación web —como por ejemplo al enviar formularios HTML, o al hacer clic en botones o enlaces— estamos enviando una **solicitud HTTP** desde el navegador hacia el servidor web. Estas solicitudes se procesan utilizando un framework para desarrollo web. **Django** es uno de ellos, y no es más que una herramienta encargada de **procesar las solicitudes del usuario, y devolver páginas web en respuesta**. Django trabaja dividiendo las tareas en **capas**, y otorgándole a cada una, una función en específico.

2. Desarrollo

En este trabajo, gran parte de la estructura de la página ya se encontraba desarrollada y dividida. Por eso, como grupo, nuestra primera tarea fue comprender el funcionamiento general del sistema. Analizamos los archivos existentes dentro del proyecto y observamos cómo Django utilizaba cada capa.

Nos propusimos ser lo más descriptivos posible para entender y explicar la tarea de cada función en las capas. Luego, avanzamos hacia la implementación de las funciones que aún no estaban completas (o directamente faltaban), como en el caso de los archivos **views.py** y **services.py**.

Completamos las funciones parcialmente desarrolladas, tratamos de improvisarlas cuando fue posible, y reutilizamos código ya existente para mantener la coherencia del proyecto.

2.1 Descripción general

En general el trabajo se centró en ciertos archivos que permiten que se puedan mostrar las cards, y filtrar los Pokémon en pantalla por su nombre y tipo (agua, fuego, planta).
Se trabajó principalmente en el archivo views.py, en los servicios definidos en services.py, y en la plantilla home.html. En algunos casos se reutilizaron estructuras ya existentes, respetando la arquitectura en capas. También se modificó la cantidad de pokémones que se mostraban en la home

2.2 Funcionalidades principales (de los archivos)

- **views.py**: las vistas es como el "cerebro" del sistema, responde a las solicitudes del navegador del usuario en la aplicación. Maneja lo que ocurre cuando el usuario ingresa una búsqueda o selecciona un filtro, por ejemplo.
- **transport.py**: se encarga de conectarse a la PokéAPI y traer todos los datos (JSON).
- **translator.py**: transforma datos sin procesar en estructuras simples que se puedan mostrar (tarjetas).
- **services.py**: éste archivo procesa pedidos desde **views.py**, y se conecta con el archivo **transport.py** para solicitarle información en crudo. Luego llama a **translator.py** para procesar dicha información en forma de tarjetas.

- **urls.py**: Es el archivo de rutas. Funciona como intermediario entre las direcciones del sitio y las funciones en **views.py**. Le indica a Django qué función debe ejecutarse cuando el usuario accede a una determinada URL.
- **home.html**: es la plantilla HTML. Muestra en pantalla la galería y el buscador interactivo. Recibe datos ya procesados desde **views.py** (como imágenes, filtros, etc.) y se encarga de mostrar toda esa información en la pantalla del usuario.

2.2 Funcionalidades principales (funciones)

Mostar todos los Pókemon:

```python
def getAllImages():
    json_pokemons = transport.get_AllImages() or []
    print(f"Cantidad de pokémon en json_pokemons: {len(json_pokemons)}")
    lista_de_tarjetas = []
    for dato in json_pokemons:
        tarjeta = translator.fromRequestIntoCard(dato)
        lista_de_tarjetas.append(tarjeta)
    return lista_de_tarjetas
```

¿Qué hace?
Obtiene los datos de todos los pokémon usando get_AllImages() y los convierte en tarjetas visuales con fromRequestIntoCard(). Devuelve una lista lista para mostrar en la galería. El or [] evita errores si no llegan datos.

Decisión tomada:
Se decidió crear esta función para unificar el proceso de carga de pokémon y evitar repetir ese código en los filtros por tipo y nombre. Así, cada vez que se necesite la lista completa, se usa esta única función.

Obtener datos de API externa:

```python
def get_AllImages():
    json_collection = []
    for id in range(1, 152):
        response = requests.get(config.STUDENTS_REST_API_URL + str(id))


        if not response.ok:
            print(f"[transport.py]: error al obtener datos para el id {id}")
            continue

        raw_data = response.json()

        if 'detail' in raw_data and raw_data['detail'] == 'Not found.':
            print(f"[transport.py]: Pokémon con id {id} no encontrado.")
            continue

        json_collection.append(raw_data)

    return json_collection
```

¿Qué hace?
Solicita uno por uno los pokémon desde la fuente externa, utilizando su número de ID del 1 al 151. Si alguno da error o no existe, lo omite automáticamente.

Decisión tomada:
En vez de traer todos los pokémon en una sola llamada “con ?limit=151”, se optó por esta versión que asegura que cada pokémon tenga su información completa. Esto también evita cargar personajes inexistentes o incompletos, y permite manejar errores con más precisión.

Transformar datos crudos en tarjetas:

```python
def fromRequestIntoCard(poke_data):
    card = Card(
        id=poke_data.get('id'),
        name=poke_data.get('name'),
        height=poke_data.get('height'),
        weight=poke_data.get('weight'),
        base=poke_data.get('base_experience'),
        image=safe_get(poke_data, 'sprites', 'other', 'official-artwork', 'front_default'),
        types=getTypes(poke_data)
    )
    return card
```

¿Qué hace?
Transforma los datos de un pokémon en una tarjeta bien estructurada, con toda la información necesaria para mostrar en pantalla: nombre, imagen oficial, altura, peso, experiencia base y tipos.

Decisiones tomadas:

Se usó safe_get para evitar errores con imágenes faltantes, y getTypes() para extraer los tipos de forma ordenada. También se accedió a cada valor con .get() para prevenir fallos si algún campo no viene cargado

Filtrar por tipo

```
# función que filtra las cards según su tipo.
def filterByType(type_filter):
    filtered_cards = []

    for card in getAllImages():
        # Compara en minúsculas para evitar errores de mayúsculas
        if type_filter.lower() in [t.lower() for t in card.types]:
            filtered_cards.append(card)

    return filtered_cards
```

¿Qué hace?
Devuelve una lista de tarjetas con los Pokémon cuyo tipo coincide con el seleccionado. La comparación se hace en minúsculas para evitar errores por diferencias de formato.

Decisión tomada:
Se convirtió todo a minúsculas para que el filtro funcione sin importar cómo venga escrito el tipo. Se usó una lista auxiliar para mantener la lista original sin modificar.

Buscar por nombre:

```
def filterByCharacter(name):
    name = name.lower()
    tarjetas_filtradas = []
    tarjetas = getAllImages()
    if not name:
        return tarjetas
    for tarjeta in tarjetas:
        nombre_de_tarjeta = tarjeta.name.lower()
        if name in nombre_de_tarjeta:
            tarjetas_filtradas.append(tarjeta)
    return tarjetas_filtradas
```

¿Qué hace?
Recibe una palabra o parte del nombre de un pokémon, busca coincidencias dentro de todos los nombres disponibles (ignorando mayúsculas) y devuelve una lista con los que coinciden.

Decisiones tomadas:
Se usó minúsculas para evitar errores de formato, se devuelve todo si el campo está vacío y se permite coincidencia parcial sin requerir el nombre exacto.

3. Conclusiones

A modo de conclusión, podemos decir que el trabajo nos permitió un acercamiento a lo que es la estructura y funcionamiento de una aplicación web desarrollada con Django. Aprendimos a ver las diferentes capas del proyecto, a comprender el rol de cada archivo del proyecto, como se conectan entre sí, y como se genera la respuesta en pantalla.

En resumen, y con lo visto hasta ahora, podemos afirmar que el flujo de trabajo funciona de la siguiente manera:

1) **Se pide la información** (views.py pide los datos a services.py, que a la vez los obtiene desde la API con transport.py).
2) **Se transforma** (los JSON se procesan en translator.py, convirtiéndolos en tarjetas)
3) **Se filtra según lo que el usuario ingrese** (views.py filtra las tarjetas basándose en la acción del usuario).
4) **Se muestra** (home.html recibe los datos ya trabajados y los muestra al usuario).

A continuación, se incluye el enunciado original del trabajo práctico solicitado.

El trabajo consiste en implementar una aplicación web usando **Django** que permita buscar imágenes de **POKéMON**. La información será proporcionada mediante una API y luego renderizada por el framework en distintas cards que mostrarán -como mínimo- la imagen del Pokémon, los tipos del mismo, altura, peso y el nivel base en el que estos existen.