

Trabajo Práctico: Pokedéx

Materia: Introducción a la Programación

Docentes: Flavia Bottino y Nora Martinez

Alumnos: Nahuel Rodrigo Cerza Albarracin, Marianela Baez y Sofia Sánchez

Comisión: 12

Índice

Introducción	3
Desarrollo de la página web	3
1. Servicios.py – Listado de las tarjetas de cada Pokémon	3
2. Views.py – Visualización de las tarjetas de Pokémon en la página web	4
3. Home.html – Color del borde de las tarjetas	4
4. Los buscadores	7
4.1 Por el nombre del Pokémon	7
4.2 Por el tipo de Pokémon	9
5. Inicio de sesión	10
6. Favoritos	12
7. Loading spinner	14

Introducción

Este trabajo práctico consistió en completar el desarrollo de una página web que muestra una serie de tarjetas de distintos pokemones, así, simulando un Pokédex. Cada tarjeta brinda información de cada Pokémon correspondiente; su imagen o la foto, su nombre, tipos, peso, altura y el nivel base en que estos existen.

Gran parte de este proyecto ya estaba desarrollado, por lo tanto, nuestro objetivo fue implementar una serie de funciones, junto con las que ya existían, para que la página pueda funcionar adecuadamente y muestre la información de las tarjetas de forma clara.

Desarrollo de la página web

1. Servicios.py – Listado de las tarjetas de cada Pokémon

Una de nuestras tareas principales fue modificar la parte de **servicios.py** del proyecto, la lógica de la aplicación, que trae y transforma los datos.

La función **getAllImages ()**, devuelve un listado con las imágenes de los pokemones ya convertidas en tarjetas.

Para ello, implementamos la variable **json_collection**, que va a pedir el listado de las imágenes con información cruda de la API (la interfaz que provee la información de los pokemones) a través de

```
función que devuelve un listado de cards. Cada card representa una imagen de la API de Pokemon
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.

    json_collection = transport.getAllImages()

    cards = []

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

    return cards
```

transport.getAllImages (). Usamos el archivo **transport** ya que es el encargado de conectarse con la API.

Abrimos la variable **cards** con una lista vacía para ir guardando las imágenes transformadas en tarjetas.

Con un for se recorre la lista de **json_collection** para que por cada **Pokémon** (imagen) allí, se transforme en una tarjeta. Esto se logra mediante la función

translator.fromRequestIntoCard (imagen), usando el **translator**, el encargado de convertir las imágenes crudas de la API en las tarjetas.

Por último, se agregan a la lista **cards** y se pide a la función **getAllImages** que retorne esa lista con las tarjetas de pokemones.

2. Views.py – Visualización de las tarjetas de Pokémon en la página web

```
# esta función obtiene 2 listados: uno de las imágenes de la API y otro de favoritos, ambos en for
def home(request):
    images = services.getAllImages()
    favourite_list = []

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

Después, se debía modificar la sección **views.py**, el controlador entre el servicio y el template (la información que se muestra al usuario).

Para que la página web muestre la tarjeta de cada Pokémon (conseguida por la función **getAllImages** en **servicios.py**), modificamos la función **home (request)** y en la variable **images** (antes con una lista vacía) le agregamos la lista de tarjetas de los pokemones mediante **services.getAllImages**, así, la función se conecta con **services.py** para pedirle los datos que va a mostrar al usuario en la página. Se decidió llamar a la lista de esa forma ya que tomamos de referencia la función anterior.

3. Home.html – Color del borde de las tarjetas

Otra tarea consistía en cambiar el color del borde de las tarjetas según el tipo de Pokémon, las pautas principales eran:

- Si es planta; el color del borde es verde.
- Si es fuego; el color debía ser rojo.
- Si es tipo agua; el borde es de color azul.

En la sección **home.html**, se utilizaron condiciones y a través del **for img in images**, se consultaba por cada imagen el tipo de Pokémon mientras que se le asignaba el color de borde correspondiente. Al principio, uno de los problemas que se nos presentaba era que las tarjetas aparecían en forma vertical (una debajo de la otra), lo que rompía con el formato adecuado del proyecto. Después de observar la estructura del archivo, se solucionó al agregar ciertas clases (**mb-3 y ms-5**) y un estilo de tarjeta (**style="max-width: 540px; "**).


Para el resto de los tipos de Pokémon, decidimos implementar colores adicionales que no estaban incluidos en Bootstrap. Por ello, se recurrió a una fuente externa


de selección de colores html para elegir los tonos y en el archivo **styles.css**, se le asignó un color característico a las tarjetas restantes para que se pueda diferenciar más cada tipo de Pokémon y mejorar el estilo de la página.


Código de **home.html**


```
{% for img in images %}
<!-- Tarjeta para mostrar cada Pokémon -->
<div class="col">
  <div class="card mb-3 ms-5
    {% Asigna una clase de borde según el tipo de Pokémon %}
    {% if 'Fire' in img.types %}
      border-danger
    {% elif 'Water' in img.types %}
      border-primary
    {% elif 'Grass' in img.types %}
      border-success
    {% elif 'Bug' in img.types %}
      border-success
    {% elif 'Ground' in img.types %}
      border-ground
    {% elif 'Poison' in img.types %}
      border-poison
    {% elif 'Electric' in img.types %}
      border-electric
    {% elif 'Normal' in img.types %}
      border-secondary
    {% elif 'Fighting' in img.types %}
      border-fighting
    {% elif 'Psychic' in img.types %}
      border-psychic
    {% elif 'Rock' in img.types %}
      border-rock
    {% elif 'Ghost' in img.types %}
      border-ghost
    {% elif 'Dragon' in img.types %}
      border-dragon
    {% elif 'Fairy' in img.types %}
      border-fairy
    {% elif 'Steel' in img.types %}
      border-steel
    {% elif 'Dark' in img.types %}
      border-dark
    {% elif 'Ice' in img.types %}
      border-ice
    {% else %}
      border-info
    {% endif %}"
    style="max-width: 540px;">
```


Código en **styles.css**


```
.border-ground {
  border-color:  #974925 !important;
  border-width: 1px !important;
}


.border-poison {
  border-color:  #A33EA1 !important;
  border-width: 1px !important;
}


.border-electric {
  border-color:  #F7D02C !important;
  border-width: 1px !important;
}

.border-ice {
  border-color:  #96D9D6 !important;
  border-width: 1px !important;
}

.border-fighting {
  border-color:  #771512 !important;
  border-width: 1px !important;
}

.border-psychic {
  border-color:  #F95587 !important;
  border-width: 1px !important;
}

.border-rock {
  border-color:  #B6A136 !important;
  border-width: 1px !important;
}

.border-ghost {
  border-color:  #735797 !important;
  border-width: 1px !important;
}
```

Lista final de los colores del borde de las tarjetas por cada tipo de Pokémon:

- Fuego: rojo
- Agua: azul
- Planta: verde
- Insecto: verde
- Tierra: marrón
- Veneno: violeta rosado
- Eléctrico: amarillo
- Normal: gris
- Hielo: celeste
- Lucha: bordo
- Psíquico: rosa
- Roca: amarillo oscuro
- Fantasma: violeta
- Dragón: violeta azulado
- Hada: rosa claro
- Acero: gris
- Sinistro: marrón claro
- Cualquier otro Pokémon: celeste oscuro

```
.border-dragon {
  border-color:  #6F35FC !important;
  border-width: 1px !important;
}

.border-fairy {
  border-color:  #D685AD !important;
  border-width: 1px !important;
}

.border-steel {
  border-color:  #B7B7CE !important;
  border-width: 1px !important;
}

.border-dark {
  border-color:  #705746 !important;
  border-width: 1px !important;
}
```

4. Los buscadores

4.1 Por el nombre del Pokémon

Para que el usuario pueda encontrar un Pokémon escribiendo su nombre en el buscador, se completó la función **def filterByCharacter (name)**, en **services.py**.

Se crea una lista: **filtered_cards**, que va a guardar la tarjeta del Pokémon encontrado.

Luego, se recorre por cada **card** in **getAllImages ()** y se consulta si el nombre ingresado por el usuario está en alguna tarjeta de dicha lista (pasando a minúsculas el texto del usuario y el nombre del Pokémon en la **card** para evitar errores). Si el nombre se encuentra, se va a agregar dicha tarjeta en la lista de **filtered_cards** y la función la devuelve.

```
# funcion que filtra segun el nombre del pokemon.
def filterByCharacter(name):
    filtered_cards = []

    for card in getAllImages():
        # debe verificar si el name está contenido en el nombre de la card, antes de agregarlo al listado de filtered_cards.
        if name.lower() in card.name.lower():
            filtered_cards.append(card)

    return filtered_cards
```

Al mismo tiempo, se desarrolló la función **def search (request)** alojada en **views.py**. La variable **query** toma el texto ingresado por el usuario, se pasa a minúsculas y quita los espacios para evitar errores.

Se tienen dos listas, una para guardar el Pokémon encontrado (**images**) y otra para los nombres de favoritos (**favourite_list_name**).

Si hay un texto en la búsqueda (if **query**), se va a armar una url de la API: **api_url**, con el nombre del Pokémon.

Se consulta a la API mediante **response= requests.get (api_url)**, si lo encuentra, se obtiene la información del Pokémon: **pokemon_data**. A su vez, se arma una lista con los tipos de Pokémon: **pokemon_type**.

Se extrae la información necesaria para mostrar a través de **pokemon_info**; nombre, id, imagen, altura, peso, nivel base los tipos.

Por último, se agrega el Pokémon encontrado a la lista **images**.

Pero si la búsqueda falla, parte **except** del código, (por ejemplo, si la API devuelve error ya que el nombre estaba mal escrito) se muestra un mensaje: “No se encontró ningún Pokémon con el nombre o ID”.

```
# Función utilizada en el buscador de Pokémon
def search(request):
    query = request.POST.get('query', '').lower().strip() # Obtiene el texto de búsqueda, lo pasa a minúsculas y quita espacios

    images = [] # Lista de resultados de búsqueda
    favourite_list_name = [] # Lista de nombres de favoritos para el contexto del template

    if query: # Si hay texto de búsqueda
        api_url = f"https://pokeapi.co/api/v2/pokemon/{query}/"
        try:
            response = requests.get(api_url) # Hace la petición a la API de PokeAPI
            response.raise_for_status() # Lanza excepción si la respuesta es errónea
            pokemon_data = response.json()

            # Extrae los tipos del Pokémon
            pokemon_types = [t['type']['name'].capitalize() for t in pokemon_data['types']]

            # Extrae los datos necesarios para el template
            pokemon_info = {
                'name': pokemon_data['name'].capitalize(),
                'id': pokemon_data['id'],
                'image': pokemon_data['sprites']['other']['official-artwork']['front_default'],
                'height': pokemon_data['height'],
                'weight': pokemon_data['weight'],
                'base': pokemon_data['base_experience'],
                'types': pokemon_types # Tipos del Pokémon
            }
            images.append(pokemon_info) # Agrega el Pokémon encontrado a la lista

        except requests.exceptions.RequestException as e:
            print(f"Error fetching Pokémon '{query}' from PokeAPI: {e}")
            messages.info(request, f"No se encontró ningún Pokémon con el nombre o ID: '{query}'")
            # Si hay error, la lista 'images' queda vacía y se muestra mensaje en el template

    # Si el usuario está autenticado, obtiene su lista de favoritos
    if request.user.is_authenticated:
        favourite_list = services.getAllFavourites(request)
        for pokemon in favourite_list:
            favourite_list_name.append(pokemon.name)

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

También, si el usuario está autenticado en la página, se va a obtener su lista de favoritos. Finalmente, se mandan los resultados al template **home.html** para que se lo muestre al usuario.

4.2 Por el tipo de Pokémon

En **services.py**, se completó la función **def filterByType (type_filter)**. **Filtered_cards** va a ser la lista que guarde las tarjetas con el tipo de Pokémon que eligió el usuario.

Por cada card en **GetAllImages ()**, se recorren sus tipos. Si alguno de los tipos coincide con el tipo seleccionado por el usuario, esta tarjeta se va a agregar a la lista **filtered_cards** y, al terminar, la función la devuelve.

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

    for card in getAllImages():
        for type in card.types:
            if type_filter in type.lower():
                filtered_cards.append(card)

    return filtered_cards
```

La función **filter_by_type (request)** (en **views.py**), va obtener el tipo de Pokémon que ingreso el usuario. Como en el buscador anterior, se tiene dos listas: **images** y **favourite_list_name**. Si el tipo es un texto (no está vacío), se llama a la función **filterByType** y le manda el tipo para que busque la tarjeta. Después, si el usuario está autenticado, se recupera su lista de favoritos y se guarda en **favourite_list_name**. Finalmente, se envían los resultados (de **images** y de la lista de favoritos) a **home.html** para que se muestren las tarjetas.

```
# Función utilizada para filtrar por el tipo del Pokémon
def filter_by_type(request):
    type = request.POST.get('type', '') # Obtiene el tipo seleccionado

    images = []
    favourite_list_name = [] # Lista de nombres de favoritos para el contexto

    if type != '':
        images = services.filterByType(type) # Trae un listado filtrado de imágenes según el tipo

    # Si el usuario está autenticado, obtiene su lista de favoritos
    if request.user.is_authenticated:
        favourite_list = services.getAllFavourites(request)
        for pokemon in favourite_list:
            favourite_list_name.append(pokemon.name)

    return render(request, 'home.html', { 'images': images, 'favourite_list_name': favourite_list_name }) # Devuelve el template con los datos
```

5. Inicio de sesión

Para permitir que los usuarios inicien sesión y se registren en la página web de Pokédex, se implementó un sistema de autenticación básico utilizando las herramientas que ofrece Django.

El archivo **views.py** contiene la función **subscribe**, la cual se encarga de registrar un nuevo usuario y enviarle un correo con sus credenciales:

```
81 # Esta función envía un mail al usuario al registrarse
82 def subscribe(request):
83     form = SubscribeForm()
84     if request.method == 'POST':
85         form = SubscribeForm(request.POST)
86         if form.is_valid():
87             usuario = form.cleaned_data
88             errores = register_user(form.cleaned_data) # Intenta registrar el usuario
89
90             if errores:
91                 for error in errores:
92                     messages.error(request,error) # Muestra errores si los hay
93             else:
94                 # Si el registro fue exitoso, envía un mail con las credenciales
95                 username = usuario['username']
96                 email = usuario['email']
97                 password = usuario['password']
98                 subject = 'Registro exitoso'
99                 message = f'¡Gracias por registrarte {username}!\nEstas son tus credenciales de inicio de sesión:\nUsuario:{username}\nContraseña:{password}'
100                 send_mail(subject, message, settings.EMAIL_HOST_USER, [email], fail_silently=False)
101         return render(request, 'registration/register.html', {'form': form})
102
```

Este código verifica si el formulario es válido. Intenta registrar al usuario usando **register_user**. Si tiene éxito, envía un mail con los datos del usuario (nombre, email y contraseña).

register.html: Este archivo renderiza un formulario para que el usuario se registre.

```
{% extends 'header.html' %} {% block content %}

<div class="register-form" style="text-align: center; display: flex; flex-direction: column; justify-content: center; align-items: center; height: 100%;">
  <div style="display: flex; flex-direction: column; align-items: center; width: 400px; margin-bottom: 20px;">
    {% if messages %}
      {% for message in messages %}
        <div class="alert alert-danger alert-dismissible fade show" role="alert">
          {{ message }}
          <button type="button" class="btn-close" data-bs-dismiss="alert" aria-label="Close"></button>
        </div>
      {% endfor %}
    {% endif %}
  </div>

  <form action="" method="POST" style="display: inline-block;">

    {% csrf_token %}
    {% for field in form %}
      <div class="form-group" style="margin-bottom: 5%; max-width: 400px">
        <label style="display: block; text-align: left; margin-bottom: 5px">{{ field.label_tag }}</label>
        {{ field }}
      </div>
    {% endfor %}

    <input type="submit" value="Submit">
  </form>
</div>

{% endblock %}
```

Se mejoró el diseño del formulario de registro centrándolo en la pantalla con flexbox y se agregó un bloque para mostrar mensajes de error con alertas de Bootstrap. Los campos se renderizan dinámicamente con un bucle **{% for field in form %}**, lo que simplifica el código. También se aplicaron estilos para mejorar la visual y usabilidad del formulario.

Login html:

Se implementó un formulario sencillo para el inicio de sesión. Permite al usuario ingresar su nombre de usuario y contraseña, con

protección **{% csrf_token %}** incluida. El diseño está centrado y usa clases de Bootstrap para mejorar la presentación visual.

```
{% extends 'header.html' %} {% block content %}
<div class="login-form" style="text-align: center;">
  <form action="" method="POST" style="display: inline-block;">
    {% csrf_token %}
    <h2 class="text-center">Inicio de sesión</h2>
    <div class="form-group" style="margin-bottom: 5%;">
      <input type="text" name="username" id="username" class="form-control" placeholder="Usuario" required="required">
    </div>
    <div class="form-group" style="margin-bottom: 5%;">
      <input type="password" name="password" id="password" class="form-control" placeholder="Contraseña" required="required">
    </div>
    <div class="form-group">
      <button type="submit" class="btn btn-primary btn-block">Ingresar</button>
    </div>
  </form>
</div>
{% endblock %}
```

Cierre de sesión

Para permitir que un usuario pueda cerrar sesión, se implementó la siguiente función en el archivo **views.py**:

Esta función utiliza **logout()** para finalizar la sesión activa del usuario y luego lo redirige automáticamente a la página principal (home). El decorador **@login_required** asegura que esta vista solo sea accesible si el usuario está autenticado.

```
# Cierra la sesión del usuario
@login_required
def exit(request):
    logout(request)
    return redirect('home')
```

6. Favoritos

```
# añadir favoritos (usado desde el template 'home.html')
def saveFavourite(request):
    fav = translator.fromTemplateIntoCard(request) # transformamos un request en una Card (ver translator.py)

    fav.user = request.user # 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_all_favourites(user) # buscamos desde el repositories.py TODOS Los favoritos del usuario (variable 'user').

        mapped_favourites = []

        for pokemon in favourite_list:
            card = translator.fromRepositoryIntoCard(pokemon)
            mapped_favourites.append(card)
        return mapped_favourites
```

Para que la sección de favoritos pueda funcionar adecuadamente, se completó la función **def saveFavourite (request)** y **def getAllFavourites (request)** de **services.py**.

Con respecto a la primera, se activa cuando el usuario interactúa con la página (home.html). En la variable **fav** va a guardar la tarjeta (es decir, la información que el usuario mandó desde el template) transformada gracias a

translator.fromTemplateIntoCard (request). Luego, esta se le asigna al usuario (autenticado) correspondiente mediante **fav.user = request.user**, es decir, se asocia la tarjeta al usuario, para que cuando se guarde el favorito, se sepa a quién le pertenece. Por último, llama a la función **save_favourite** del archivo **repositories** para que guarde el favorito en la base de datos.

En la función **def getAllFavourites**, recibe los datos de sesión del usuario y se verifica que el usuario no está autenticado, si es así, se devuelve una lista vacía.

Si está autenticado, la variable **user** guarda la información del usuario.

Favourite_list va a tener la lista de todos los favoritos del usuario a través de **get_all_favourites (user)** del archivo **repositories**. Se abre una lista **mapped_favourites**, por cada **Pokémon** en **favourite_list**, se convierte en una **card** mediante **translator.fromRepositoryIntoCard** y después se agrega la tarjeta a la lista de **mapped_favourites**. Al terminar, la función la devuelve.

En **views.py**, hay una serie de funciones que se van a usar solo si el usuario está logueado: **@login_required**. En **def getAllFavouritesByUser (request)** se le agrega a la lista: **favourite_list**, las tarjetas de los favoritos del usuario obtenidas de la función **GetAllFavourites**, así se muestran al usuario.

```
# Estas funciones se usan cuando el usuario está logueado en la aplicación.

# Muestra la lista de favoritos del usuario
@login_required
def getAllFavouritesByUser(request):
    favourite_list = []
    images = [] # Puede usarse para mostrar detalles de los favoritos
    favourite_list = services.getAllFavourites(request)

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

# Guarda un Pokémon como favorito
@login_required
def saveFavourite(request):
    services.saveFavourite(request)
    return redirect('favoritos')

# Elimina un Pokémon de favoritos
@login_required
def deleteFavourite(request):
    services.deleteFavourite(request)
    return redirect('favoritos')
```

Luego, en **def SaveFavourite (request)** se llama a la función **saveFavourite** de **servicios** que guarda el favorito en la base de datos y después dirige al usuario a la sección de favoritos para que vea que fue guardado.

Finalmente, en **def deleteFavourite (request)**, se llama a la función **deleteFavourite** que está en `servicios.py` para que elimine el favorito de la sección. Luego, redirige al usuario a la sección de favoritos con la tabla actualizada.

7. Loading Spinner

```
<div id="loading_home" style="display: none; position: fixed; top: 0; left: 0; width: 100vw; height: 100vh; background-color: ■rgba(255,255,255,0.8); z-index: 9999;
  <span class="spinner-border text-primary" role="status"></span> text-align: center; padding-top: 200px;">
  <p>Cargando...</p>
</div>

<script>
  window.addEventListener("beforeunload", () => {
    const spinner = document.getElementById("loading_home");
    if (spinner) {
      spinner.style.display = "block";
    }
  });
</script>
```

Para implementar el **loading spinner**, se implementó un `div` dentro del archivo **header.html** ya que este encabezado se incluye automáticamente en todas las páginas del sitio. De esta manera, el símbolo “Cargando” se va a mostrar en todas las vistas de la página cuando se recargue alguna de ellas y, al mismo tiempo, se evita repetir el código en cada archivo.

En la primera línea: **div id="loading_home"**, le da un identificador al **spinner** para poder encontrarlo con JavaScript. **Style="display: none;"**, sirve para ocultarlo cuando no se esté cargando la página, **position: fixed; top: 0; left: 0;** hace que ocupe toda la pantalla y **width: 100vw; height: 100vh** lo estira a todo el ancho y alto de la ventana. Luego, **background-color rgba (255, 255, 255, 0.8)** le pone un fondo blanco semitransparente, **z-index: 9999;** hace que se vea por encima de todo y **text-align: center; padding-top: 200px;** centra el texto y lo baja un poco desde arriba.

En la parte ****, se trata del ícono de la carga (rueda girando), **spinner border** es una clase de Bootstrap que dibuja la animación, **text-primary** le da un

color azul y **role="status"** es un atributo para que si una persona está usando un lector de pantalla, este lea lo que se muestra.

<p> Cargando...</p> muestra ese texto debajo del ícono.

En el **script: window.addEventListener("beforeunload"...**) es donde se detecta cuando el usuario hace algo que lleve a recargar la página.

const spinner = documents.getElementById ("loading_home"); busca el div con el id "loading_home" y una vez que lo encuentra lo muestra: **display= "block"** y así, aparece la animación.