Informe Trabajo Práctico: Aplicación web de búsqueda de Pokémon

INTEGRANTES: Valentina D'Urbano y Martin Esquivel

Introducción

Este trabajo práctico consistió en desarrollar una aplicación web que permite a los usuarios buscar y visualizar información sobre distintos Pokémon. Para lograrlo, nuestra aplicación se conecta a un servicio externo en línea (una API) para obtener los datos necesarios. Estos datos son luego presentados de manera organizada en "tarjetas" o "cards", donde cada una muestra detalles importantes como la imagen del Pokémon, sus tipos (por ejemplo, fuego, agua), su altura, peso y su nivel de experiencia base.

El proyecto ya nos proporcionaba una estructura de arquitectura en capas predefinida, lo cual fue fundamental para organizar nuestro proceso de desarrollo. Nuestra tarea principal fue implementar y completar las funcionalidades necesarias para que la galería de Pokémon se mostrara correctamente y para que los sistemas de búsqueda y filtrado operaran de forma eficiente.

Implementaciones Clave en el Código

A continuación, detallamos las modificaciones y adiciones específicas que realizamos en los archivos del proyecto, explicando el propósito de cada cambio.

Archivo services.py: Lógica de Negocio

Este archivo es el corazón de nuestra aplicación, donde definimos cómo se procesan y manipulan los datos. Nos enfocamos en las funciones de obtención de datos desde la API y en la lógica de filtrado.

Función **getAllImages():** Esta función es la encargada de traer los datos crudos de Pokémon desde la API externa. Procedimos a iterar sobre la información recibida, transformando cada conjunto de datos de Pokémon en un objeto "Card" más manejable. Estas Cards se añaden a una lista final que la función devuelve. Incorporamos un bloque de control de errores para asegurar que, si algún Pokémon individual presentara datos inconsistentes de la API, la aplicación continúe funcionando sin interrupciones, procesando los demás.

```
def getAllImages():
    raw_pokemon_data = transport.getAllImages()
    pokemon_cards = []
    for pokemon_raw_data in raw_pokemon_data:
        try:
            card = translator.fromRequestIntoCard(pokemon_raw_data)
            pokemon_cards.append(card)
```

```
        except Exception as e:
            print(f"[service.py]: Error al traducir Pokémon {pokemon_raw_data.get('name',
'desconocido')}: {e}")
            continue
    return pokemon_cards
```

Función **filterByCharacter(name):** Implementamos esta función para habilitar el filtrado de Pokémon por nombre. Primero, obtenemos la lista completa de todos los Pokémon disponibles (reutilizando **getAllImages()**). Luego, comparamos el nombre buscado por el usuario (convertido a minúsculas) con el nombre de cada Pokémon (también en minúsculas). Esto permite búsquedas parciales y sin distinción entre mayúsculas y minúsculas, agregando a la lista de resultados solo aquellos Pokémon que coinciden con el criterio de búsqueda.

```
 def filterByCharacter(name):
    filtered_cards = []
    all_images = getAllImages()
    for card in all_images:
        if name.lower() in card.name.lower(): # Realiza búsqueda parcial y sin distinguir
mayúsculas/minúsculas.
            filtered_cards.append(card)
    return filtered_cards
```

Función **filterByType(type_filter):** Esta función se encarga del filtrado de Pokémon por tipo. Al igual que con el filtro por nombre, iniciamos obteniendo la lista completa de Pokémon. La lógica de filtrado es crucial aquí, ya que un Pokémon puede tener varios tipos. Utilizamos una verificación para determinar si al menos uno de los tipos del Pokémon coincide con el tipo de filtro ingresado por el usuario (ambos convertidos a minúsculas para una comparación flexible). Los Pokémon que cumplen con este criterio son añadidos a la lista de resultados filtrados.

```
 def filterByType(type_filter):
    filtered_cards = []
    all_images = getAllImages()
    for card in all_images:
        if any(pokemon_type.lower() == type_filter.lower() for pokemon_type in card.types): #
Busca coincidencia en cualquiera de los tipos del Pokémon, sin distinguir
mayúsculas/minúsculas.
            filtered_cards.append(card)
    return filtered_cards
```

Archivo views.py: Interacción con el Usuario

Este archivo gestiona las solicitudes de los usuarios y coordina la presentación de datos en las plantillas HTML. Nuestros cambios aquí se centraron en conectar la interfaz de usuario con la lógica de negocio.

Función **home(request):** Esta función fue modificada para poblar la galería de Pokémon con datos reales. Ahora, invoca a **services.getAllImages()** para obtener la lista de Cards procesadas y a **services.getAllFavourites(request)** para la lista de favoritos (si la

funcionalidad estuviera desarrollada, de lo contrario, se espera una lista vacía). Estas listas se pasan a la plantilla **home.html** para su visualización.

```
def home(request):
    images = services.getAllImages()
    favourite_list = services.getAllFavourites(request) # Obtiene los favoritos (si la funcionalidad completa no está desarrollada, devuelve una lista vacía).
    return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
```

Función **search(request):** Esta función activa el buscador por nombre. Cuando el usuario envía un texto de búsqueda, la función delega el filtrado a **services.filterByCharacter(name).** Los resultados filtrados son luego renderizados en la página **home.html.** Si el campo de búsqueda se envía vacío, la página simplemente redirige a la galería completa.

```
def search(request):
    name = request.POST.get('query', '')
    if (name != ''):
        images = services.filterByCharacter(name) # Delega el filtrado a la capa de servicios.
        favourite_list = services.getAllFavourites(request)
        return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
    else:
        return redirect('home')
```

Función **filter_by_type(request):** Esta función habilita el filtro por tipo. Al seleccionar un tipo, la función delega la tarea de filtrado a **services.filterByType(type).** Los Pokémon que coinciden con el tipo seleccionado son entonces presentados en **home.html.** De manera similar, si no se especifica un tipo, se redirige a la galería completa.

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

    if type != '':
        images = services.filterByType(type) # Delega el filtrado a la capa de servicios.
        favourite_list = services.getAllFavourites(request)
        return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
    else:
        return redirect('home')
```

Archivo home.html: Presentación Visual

Este archivo es la plantilla HTML que define la apariencia de la galería. Nos centramos en la visualización de las tarjetas y la interacción con los formularios de búsqueda.

Implementamos una funcionalidad clave para que el **color del borde de cada tarjeta de Pokémon cambiara dinámicamente según su tipo**. Para lograr esto, utilizamos una combinación del **Lenguaje de Plantillas de Django (DTL)** y las **clases CSS predefinidas de Bootstrap**. Bootstrap es un *framework* de desarrollo front-end que nos permite diseñar interfaces de usuario responsivas y estéticas de manera rápida y eficiente sin tener que escribir mucho código de estilo desde cero. En este caso, usamos las clases como border-danger (rojo), border-primary (azul), border-success (verde) y border-warning (naranja) de Bootstrap para los bordes de las tarjetas. La lógica condicional se incorporó directamente en el atributo

class de la etiqueta div que representa cada tarjeta. Esta lógica evalúa los tipos del Pokémon actual, los une en una cadena de texto y los convierte a minúsculas, aplicando la clase de borde de Bootstrap correspondiente.

<div class="card {% if 'fire' in img.types|join:" "|lower %}border-danger{% elif 'water' in img.types|join:" "|lower %}border-primary{% elif 'grass' in img.types|join:" "|lower %}border-success{% else %}border-warning{% endif %} mb-3 ms-5" style="max-width: 540px; ">

También nos aseguramos de que los formularios de búsqueda y los botones de filtro por tipo estuvieran correctamente enlazados a sus respectivas funciones en **views.py.** El formulario de búsqueda por nombre envía el texto ingresado a la vista **search**, mientras que cada botón de filtro por tipo envía su valor específico a la vista **filter_by_type** mediante un campo oculto. Ambos formularios incluyen el token de seguridad necesario.

```
<form class="d-flex" action="{% url 'buscar' %}" method="POST">
   {% csrf_token %}
   <input class="form-control me-2" type="search" name="query" placeholder="Pikachu, Charizard, Ditto" aria-label="Search">
   <button class="btn btn-outline-success" type="submit">Buscar</button>
</form>
<div class="d-flex gap-1 col-2 mx-auto mb-3">
   <form method="post" action="{% url 'filter_by_type' %}">
      {% csrf_token %}
      <input type="hidden" name="type" value="fire">
      <button type="submit" class="btn btn-danger">FUEGO</button>
   </form>
   <form method="post" action="{% url 'filter_by_type' %}">
      {% csrf_token %}
      <input type="hidden" name="type" value="water">
      <button type="submit" class="btn btn-primary">AGUA</button>
   </form>
   <form method="post" action="{% url 'filter_by_type' %}">
      {% csrf_token %}
      <input type="hidden" name="type" value="grass">
      <button type="submit" class="btn btn-success">PLANTA</button>
   </form>
</div>
```

Mejora de la experiencia de usuario: Implementación del Loading Spinner

Uno de los objetivos que tuvimos fue hacer que la aplicación se sintiera más fluida y agradable de usar. Cuando alguien entra a una página y no pasa nada durante unos segundos, es fácil pensar que se rompió algo. Para evitar esa sensación, decidimos agregar un **Loading Spinner**: una animación circular que aparece mientras se cargan los datos desde la API externa (lo cual puede demorar).

Para que este spinner aparezca en todas las páginas sin tener que repetir código, lo pusimos en header.html, que es una plantilla base. Como todas las páginas del sitio extienden esa plantilla, logramos que el spinner estuviera disponible en todo el sitio sin esfuerzo adicional.

En cuanto al estilo, creamos un bloque de CSS que se encarga de mostrar el spinner centrado en la pantalla, con un fondo semitransparente blanco por detrás. También definimos la animación para que gire constantemente y nos aseguramos de que esté siempre por encima del resto de los elementos de la página con z-index: 9999.

Componentes Técnicos en header.html:

Estilos CSS (<style> en la sección <head>): Se definieron las reglas CSS para el "overlay" (una capa semitransparente que cubre la pantalla) y para el propio "spinner" (un círculo animado). Estas reglas controlan la posición fija del spinner, su centrado en la ventana de visualización (display: flex, justify-content: center, align-items: center), su fondo semitransparente (background-color: rgba(255, 255, 255, 0.8)), y la animación de giro (animation: spin 2s linear infinite;). El z-index: 9999 asegura que el spinner siempre esté por encima de todos los demás elementos de la página.

```
<style>
 #spinner-overlay {
   position: fixed;
   top: 0;
   left: 0;
   width: 100%;
   height: 100%;
   background-color: rgba(255, 255, 255, 0.8);
   display: flex;
   justify-content: center;
   align-items: center;
   z-index: 9999;
 }
 .spinner {
   border: 8px solid #f3f3f3;
   border-top: 8px solid #3498db;
   border-radius: 50%;
   width: 60px;
   height: 60px;
   animation: spin 2s linear infinite;
 }
 @keyframes spin {
   0% { transform: rotate(0deg); }
   100% { transform: rotate(360deg); }
 }
</style>
```

Archivo "login.html": Presentación del formulario de inicio de sesión

Esta plantilla HTML define la interfaz para que los usuarios puedan acceder a sus cuentas.

Inicialmente, el formulario de inicio de sesión se había intentado construir "manualmente", pero presentó algunas desventajas, como la pérdida de la validación automática: Al renderizar los campos manualmente, se pierde la capacidad de Django para inyectar automáticamente mensajes de error de validación (por ejemplo, "datos invalidos" o "este campo es obligatorio") directamente debajo del campo afectado.
Por esta razón, optamos por un enfoque que nos pareció más eficiente y aprovechamos las capacidades del sistema de formularios de Django. La decisión de utilizar la renderización de formularios de Django ("{{ formulario.username }}" y "{{ formulario.password }}") fue conveniente por las siguientes razones:

1. Integración de validación de errores: Django muestra los errores justo debajo de cada campo, lo que ayuda mucho al usuario a saber qué se equivocó.

```html
{% if formulario.non_field_errors %}
    <div class="alert alert-danger">
        {% for error in formulario.non_field_errors %}
            <p>{{ error }}</p>
        {% endfor %}
    </div>
{% endif %}
```

Y para los errores por campo:

```html
{% if formulario.username.errors %}
    {% for error in formulario.username.errors %}
        <p class="text-danger">{{ error }}</p>
    {% endfor %}
{% endif %}
```

2. Estilos predefinidos: En forms.py, ya habíamos asignado las clases CSS para que los inputs se vean bien. Usando el sistema de formularios, se aplican automáticamente.

```html
<div class="form-group" style="margin-bottom: 5%;">
    {{ formulario.username.label_tag }} {# Opcional, para la etiqueta del campo #}
    {{ formulario.username }} {# Renderiza el input con sus atributos #}
```

```
</div>
```

3. Reducción de código y mantenimiento: Al delegar la renderización a Django, el código HTML se vuelve más conciso y fácil de leer. Más importante aún, cualquier cambio futuro en la estructura o apariencia de los campos (ej. añadir un atributo `required`, cambiar el tipo de `input`) solo necesitará ser modificado en `forms.py`, sin tocar el template. Esto mejora significativamente la mantenibilidad del código.

Archivo "register.html": Página de registro de usuarios

Esta página no estaba entre las plantillas originales, así que tuvimos libertad para diseñarla desde cero. La idea fue crear una vista simple y clara para que nuevos usuarios puedan registrarse fácilmente.

En vez de escribir todos los <input> a mano, usamos el método {{ formulario.as_p }} de Django, que nos ahorra muchísimo trabajo: dibuja todos los campos necesarios (usuario, contraseña, confirmación) y también muestra los errores automáticamente si algo está mal.

Además, incluimos un mensaje de error general en caso de que haya algún problema con el formulario (como que el usuario ya existe o las contraseñas no coinciden) y un enlace para que quien ya tiene cuenta pueda iniciar sesión directamente.

```
{% extends 'header.html' %}
{% block content %}
<div class="register-form" style="text-align: center;">
   <form method="POST" style="display: inline-block; text-align: left; max-width: 400px; margin: auto;">
      {% csrf_token %}
      <h2 class="text-center">Registrarse</h2>

      {% if formulario.errors %}
         <div class="alert alert-danger">
            <ul>
               {% for field in formulario %}
                  {% for error in field.errors %}
                     <li>{{ error }}</li>
                  {% endfor %}
               {% endfor %}
               {% for error in formulario.non_field_errors %}
                  <li>{{ error }}</li>
               {% endfor %}
            </ul>
         </div>
      {% endif %}

      {{ formulario.as_p }}
```

```
      <div class="form-group" style="text-align: center;">
        <button type="submit" class="btn btn-success">Crear cuenta</button>
      </div>

      <div class="mt-3 text-center">
        <small>¿Ya tenés cuenta? <a href="{% url 'login' %}">Iniciá sesión acá</a></small>
      </div>
    </form>
</div>
{% endblock %}
```

Conexión con Django y "forms.py" (fundamento del registro):

La clave de este template radica en su integración con el sistema de formularios de Django y, específicamente, con nuestra clase "FormularioCreacionUsuarioPersonalizado" definida en "forms.py". En lugar de construir el formulario con etiquetas "<input>" HTML individuales, el template renderiza el formulario de Django de manera eficiente con "{{ formulario.as_p }}". Esta decisión es fundamental porque:

1) Automatiza la creación de campos: Para el formulario de registro decidimos reutilizar una clase que ya viene con Django, que nos permite crear usuarios de forma segura. Esto nos ahorró tiempo y nos evitó tener que programar validaciones desde cero. También usamos {{ formulario.as_p }} para mostrar los campos de forma automática y ordenada en la interfaz.

2) Manejo robusto de errores: El bloque "{% if formulario.errors %}" es crucial. Así logramos que los errores comunes, como contraseñas que no coinciden o usuarios ya registrados, se muestren automáticamente al usuario, mejorando mucho la experiencia de uso. Este manejo de errores detallado y campo por campo (iterando sobre "formulario.errors" y "formulario.non_field_errors") mejora significativamente la usabilidad y la retroalimentación al usuario.

3) Integración de estilos: Gracias a la lógica en "forms.py" (explicada a continuación), todos los campos se renderizan automáticamente con la clase "form-control" de Bootstrap, asegurando una apariencia uniforme y estética sin necesidad de añadirla manualmente en el HTML.

4) Redirección y usabilidad: Incluye un enlace explícito ("{% url 'login' %}") para que los usuarios que ya tienen una cuenta puedan navegar fácilmente a la página de inicio de sesión, mejorando el flujo de usuario.

Archivo "forms.py": Definición de formularios de autenticación personalizados

Este archivo fue creado en equipo con el propósito de centralizar y personalizar los formularios necesarios para las operaciones de autenticación de usuarios (inicio de sesión y registro), aprovechando las clases base proporcionadas por Django. Este enfoque nos permitió desacoplar la lógica del formulario del resto del código y aplicar estilos de manera consistente.

Conexión con el sistema de autenticación de Django: "forms.py" es el puente entre los datos que el usuario ingresa en el "frontend" y el sistema de autenticación de Django en el "backend". Define la estructura y las reglas de validación de los datos antes de que sean procesados.

"FormularioInicioSesion(AuthenticationForm)":

```
class FormularioInicioSesion(AuthenticationForm):
    def __init__(self, *args,**kwargs):
        super().__init__(*args, **kwargs)
        self.fields['username'].widget.attrs.update({'class': 'form-control', 'placeholder': 'Nombre de usuario'})
        self.fields['password'].widget.attrs.update({'class': 'form-control', 'placeholder': 'Contraseña'})
```

Propósito: Gestionar la entrada de credenciales para el inicio de sesión.

Herencia y personalización: Hereda directamente de "django.contrib.auth.forms.AuthenticationForm". Esta herencia es fundamental porque nos proporciona de inmediato toda la lógica de Django para validar credenciales (verificar usuario y contraseña contra la base de datos).

Método "__init__" para estilización: La personalización se realiza dentro del método "__init__". Aca accedemos a los campos "username" y "password" que "AuthenticationForm" ya define internamente. Para que todos los formularios tuvieran una apariencia uniforme y siguieran el estilo de Bootstrap, aplicamos las clases CSS necesarias desde el archivo forms.py. Esto nos permitió mantener el HTML más limpio y no repetir código en cada template.

"FormularioCreacionUsuarioPersonalizado(UserCreationForm)":

```
class FormularioCreacionUsuarioPersonalizado(UserCreationForm):
  def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        for nombre_campo, campo in self.fields.items():
              campo.widget.attrs['class'] = 'form-control'
```

Propósito: Gestionar la creación de nuevas cuentas de usuario.
Herencia robusta: Hereda de "django.contrib.auth.forms.UserCreationForm". Usamos esta clase porque ya incorpora buenas prácticas para la creación de usuarios, como el manejo

seguro de contraseñas. Así pudimos centrarnos en el diseño y en que el formulario se vea y funcione bien. Esto nos ahorra una gran cantidad de código y nos garantiza una implementación segura del registro.

Estilización dinámica: Similar al formulario de inicio de sesión, el método "__init__" se utiliza para iterar sobre todos los campos generados por "UserCreationForm" (nombre de usuario, contraseña1, contraseña2) y aplicarles la clase "form-control" de Bootstrap. Esto asegura que todos los elementos del formulario de registro se muestren con el estilo deseado, manteniendo la consistencia visual de la aplicación sin esfuerzo manual en el HTML.

En conclusión, la creación de `forms.py` y `register.html` por nuestro equipo fue una decisión estratégica para implementar un sistema de autenticación robusto y amigable con el usuario. Al heredar y extender las potentes herramientas de formularios y autenticación de Django, pudimos desarrollar esta funcionalidad de manera eficiente, segura y altamente mantenible, garantizando que tanto el inicio de sesión como el registro se integraran sin problemas.

Eliminación de la funcionalidad "Favoritos": Detalle y justificación

La funcionalidad de "Favoritos" fue inicialmente considerada y contenía un esqueleto de implementación. Sin embargo, para cumplir con el alcance definido para el trabajo final y simplificar el proyecto a sus requisitos principales (visualización, búsqueda y filtrado de Pokémon), tomamos la decisión de eliminar completamente esta funcionalidad. Esto implicó eliminar todos los componentes de código y estructura de base de datos relacionados.

A continuación, se detalla qué archivos y elementos fueron eliminados, y el porqué:

1. **Archivo: "app/models.py" - Modelo "Favourite"**
   Elemento eliminado: La definición completa de la clase "Favourite(models.Model)".
   Por qué: Este modelo representaba la estructura de los datos de un Pokémon favorito a ser guardado en la base de datos y su relación con un usuario. Al decidir no implementar la funcionalidad de "Favoritos", esta tabla y su definición en el ORM de Django se volvieron redundantes y su existencia habría causado migraciones innecesarias o conflictos en la base de datos.

2. **Archivo: "app/layers/persistence/repositories.py"**
   Elemento eliminado: El archivo completo y todas sus funciones ("save_favourite", "get_all_favourites", "delete_favourite").
   Por qué: Este archivo constituía la capa de persistencia (DAO) para la funcionalidad de "Favoritos", encargada de interactuar directamente con la base de datos (SQLite) para el alta, baja y consulta de los objetos `Favourite`. Dado que era su único propósito y el modelo "Favourite" fue eliminado, este archivo ya no tenía ninguna función en el proyecto.

3. **Archivo: "app/layers/services/services.py" - Funciones de favoritos**

Elemento eliminado: Las funciones "saveFavourite", "getAllFavourites", "deleteFavourite" y la importación a "repositories".

Por qué: Estas funciones formaban parte de la capa de servicios y actuaban como la lógica de negocio para la gestión de favoritos, invocando a la capa de persistencia. Al eliminar la capa de persistencia y el modelo, estas funciones se volvieron inoperantes y se eliminaron para mantener el código limpio y libre de referencias a funcionalidades inexistentes.

4. Archivo: "app/views.py" - Funciones de favoritos y referencias

Elemento eliminado: Las funciones "getAllFavouritesByUser", "saveFavourite", "deleteFavourite" y cualquier llamada a "services.getAllFavourites()" en otras vistas como "home()", "search()", y "filter_by_type()".

Por qué: Estas vistas y llamadas eran los puntos de entrada desde la interfaz de usuario para interactuar con la lógica de favoritos. Su eliminación garantiza que no haya rutas ni intentos de obtener o manipular datos de favoritos desde el frontend.

5. Archivo: "app/urls.py" - Rutas de favoritos

Elemento eliminado: Las definiciones de rutas ("path") para "favoritos" ("/favourites/", "/favourites/add/", "/favourites/delete/").

Por qué: Estas URLs ya no apuntaban a vistas existentes ni a una funcionalidad activa.

6. Archivo: "header.html" - Enlace "Favoritos"

Elemento eliminado: La línea HTML ("<li>") que creaba el enlace "Favoritos" en la barra de navegación.

Por qué: Era el punto de acceso visual para los usuarios a la sección de favoritos. Al no existir la funcionalidad, el enlace no tenía razón de ser y fue eliminado para limpiar la interfaz de usuario y evitar confusiones.

7. Archivo: "home.html" - Botón "Agregar a Favoritos"

Elemento Eliminado: El bloque de código HTML y de plantilla de Django que generaba el botón "Agregar a Favoritos" dentro de cada tarjeta de Pokémon.

Por qué: Este botón era la interacción directa del usuario para añadir Pokémon a su lista de favoritos. Al igual que el enlace del "header", su remoción es vital para reflejar la eliminación de la funcionalidad en la interfaz de usuario.

8. Archivo: "favourites.html" - Plantilla completa

Elemento eliminado: El archivo "favourites.html" completo.

Por qué: Esta plantilla estaba diseñada para mostrar el panel o listado de Pokémon favoritos. Dado que toda la lógica y los modelos de favoritos fueron eliminados, esta plantilla se volvió completamente redundante y fue removida para mantener el proyecto ordenado.

9. Carpeta: "app/migrations/" - Archivos de migración de "app" (excepto "__init__.py")

Elemento eliminado: Todos los archivos ".py" dentro de la carpeta "app/migrations/" (ej. "0001_initial.py", "0002_favourite_...py", "0003_usuario.py", etc.).

Por qué: Estos archivos eran el historial de cambios de la base de datos de la aplicación "app", incluyendo la creación de la tabla `Favourite`. Su eliminación (junto con "db.sqlite3") permitió a Django "resetear" el estado de la base de datos para la aplicación "app" y generar un nuevo historial de migraciones que refleje la ausencia de modelos personalizados, garantizando una base de datos limpia y consistente con el código final.

10. Archivo: "db.sqlite3"

Elemento eliminado: El archivo de la base de datos SQLite.

Por qué: Contenía la base de datos actual del proyecto, incluyendo la tabla de "Favoritos" y sus datos, así como el historial de migraciones aplicado. Su eliminación fue necesaria para garantizar una base de datos completamente fresca y libre de cualquier vestigio de la funcionalidad eliminada, recreada con las migraciones actualizadas.

Esta eliminación exhaustiva y justificada asegura que el proyecto final se centre únicamente en las funcionalidades requeridas, minimizando el código redundante y la complejidad.

Proceso de migración y creación de "superusuario"

Después de las modificaciones en el código y la eliminación de archivos, se ejecutaron los siguientes comandos en la terminal (mcd) para actualizar la base de datos y preparar la aplicación:

1. **"python manage.py makemigrations app":** Generó un nuevo archivo de migración (o confirmó que no había cambios de modelos en “app” que refleja la ausencia del modelo “Favourite”.

2. **"python manage.py migrate"**: Aplicó las migraciones, recreando las tablas de la base de datos de Django (incluyendo las de autenticación) en un nuevo “db.sqlite3”.

3. **"python manage.py createsuperuser":** Permitió la creación de un nuevo usuario administrador para acceder al panel de Django y probar la autenticación.

4. **"python manage.py runserver":** Inició el servidor de desarrollo para verificar el funcionamiento de la aplicación en el navegador.

Dificultades de implementación y decisiones tomadas

Durante el desarrollo de este trabajo práctico nos enfrentamos a varios desafíos, principalmente debido a que gran parte de las herramientas y conceptos utilizados eran nuevos para nosotros. Tuvimos que investigar, pedir ayuda y, sobre todo, dedicar tiempo a entender cómo encajaban las distintas partes del proyecto.

Una de las principales dificultades fue comprender el flujo de datos entre las distintas capas del sistema: desde la obtención de información de la API hasta su visualización en el navegador. Esto nos obligó a analizar cuidadosamente el rol de cada componente (servicios, vistas, plantillas) para asegurarnos de que los datos se manipularan y presentaran correctamente.

Otro punto complejo fue implementar la lógica condicional para cambiar el color de los bordes de las tarjetas según el tipo de Pokémon. Esto implicó trabajar con listas de tipos y decidir cómo comparar valores dentro de las plantillas de Django, algo que al principio nos resultaba poco intuitivo.

También fue un desafío diseñar filtros de búsqueda útiles y funcionales. Queríamos permitir tanto búsquedas parciales por nombre como filtrado por tipo, considerando además que un Pokémon puede tener más de un tipo. Resolver esto nos llevó a tomar decisiones sobre cómo estructurar las funciones y reutilizar código ya existente.

Entre las decisiones más importantes que tomamos están: reutilizar la función de obtención de todos los Pokémon (getAllImages()) en los filtros para evitar duplicar código; mantener separadas las responsabilidades de cada capa (por ejemplo, dejando la lógica de negocio en los servicios y evitando sobrecargar las vistas); y usar clases CSS de Bootstrap para manejar el estilo visual, lo que nos permitió mantener una estética coherente y agilizar el desarrollo.

Muchas de las decisiones que tomamos a lo largo del proyecto surgieron de prueba y error, investigación y colaboración. No sabíamos todo desde el principio, pero fuimos aprendiendo sobre la marcha qué herramientas de Django nos podían facilitar las cosas. Implementar formularios reutilizables, un spinner global o una arquitectura organizada fueron aprendizajes valiosos que surgieron de enfrentar obstáculos reales y buscar la mejor forma de resolverlos.

En definitiva, más allá de los obstáculos técnicos, este trabajo nos ayudó a comprender mejor cómo se estructura y desarrolla una aplicación web. El proceso de tener que aprender a medida que implementábamos fue clave para afianzar los contenidos de la materia y ganar experiencia práctica en desarrollo web.