Universidad Nacional de General Sarmiento

Licenciatura en Sistemas

Trabajo Práctico
Pokedéx

Introducción a la Programación

(1er semestre de 2025)

Resumen: El trabajo consiste en la implementación de una aplicación web desarrollada con Django, que permite buscar imágenes de Pokémon y mostrar su información en una tarjeta o card, a través de una API. Una gran parte de la aplicación ya estaba hecha, solo faltaba implementar algunas funciones importantes.

Integrantes: Agustin Avalos Mildemberger / agustinavalos08@gmail.com
Facundo Pacheco / Facundomartinpacheco26@gmail.com
Santiago Verón / verondsanti@gmail.com
Matias Miguel Serapio / matiasserapio@outlook.com

1. Introducción: El trabajo trata de completar la aplicación web implementado tres importantes funciones faltantes en tres módulos:

Views.py: completar home(request)

```python
# esta función obtiene 2 listados: uno de las imágenes de la API y otro de favoritos, ambos en formato Card, y los dibuja en el template 'home.html'.
def home(request):
    images = []
    favourite_list = []

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

Service.py: completar getAllImages

```python
# función que devuelve un listado de cards. Cada card representa una imagen de la API de HP.
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.
    # ATENCIÓN: contemplar que los nombres alternativos, para cada personaje, deben elegirse al azar. Si no existen nombres alternativos, debe mostrar un mensaje adecuado.
    pass
```

home.html: cambiar el color del borde de la tarjeta/card según el tipo de Pokémon

```html
<div class="col">
    <!-- evaluar si la imagen pertenece al tipo fuego, agua o planta -->
    <div class="card border-success mb-3 ms-5" style="max-width: 540px; ">
```

Además de implementar algunas funcionalidades opcionales como:

- Buscador I (filtro según el nombre del pokémon)
- Buscador II (filtro según tipo fuego, agua o planta)
- ALTA de nuevos usuarios
- Loading Spinner para la carga de imágenes
- La posibilidad de que los usuarios registrados tengan pokemons como favoritos
- Renovación de la interfaz gráfica utilizando Bootstrap

2. Desarrollo: Para resolver estos problemas decidimos dividirnos el trabajo entre los integrantes del grupo, tuvimos varias dificultades como:

- **Comprensión de la consigna**: Inicialmente fue difícil entender si había que modificar o crear funciones desde cero. Se resolvió con ayuda y releyendo la consigna varias veces.
- **GitHub**: No teníamos experiencia previa. Aprendimos a clonar el repositorio, hacer commits y trabajar en ramas.
- **Comparación de versiones**: Comparar el archivo original con la versión editada fue importante para verificar si las modificaciones cumplían con lo requerido y mantenían el resto del código funcional.
- **Entender el código existente**: El proyecto ya tenía varias capas de código preexistente, por lo que fue difícil entender bien cómo se manejaba tanta cantidad de información entre tantas carpetas y archivos .html como .py.

2.1 Descripción general

Se buscó implementar una función llamada getAllImages() en services.py que obtenga imágenes de Pokémon desde una API externa, las convierta en cards utilizando una función de traducción, y las devuelva para su visualización en una aplicación web; home(request) en views.py utiliza getAllImages() para obtener los Pokémon y getAllFavorites(request) para cargar los favoritos del usuario. Ambos listados se pasan al template home.html, donde se renderiza la vista principal de la aplicación. Esto permite separar la lógica de

datos de la lógica de presentación de forma clara; en home.html se usó el condicional IF más documentación de Bootstrap sobre Card para cambiar el color del borde predeterminado por otros colores dependiendo del tipo de pokemon (fuego, agua, planta).

2.2 Funcionalidades principales

◆ **Función: services.py > getAllImages()**

- **Idea general:**
  Obtener un listado de Pokémon desde la API, procesar cada uno en una estructura de tipo card, y devolver la lista final.
- **Realización:**
  Centralizar el acceso a los datos crudos de la API y su conversión en tarjetas. Además, incorporar nombres alternativos seleccionados aleatoriamente, si están disponibles.
- **Código comentado:**

```
import random

def getAllImages():
    raw_pokemons = transport.getAllImages() # Obtiene los datos crudos desde la API
    cards = []

    for poke_data in raw_pokemons:
        alternative_names = poke_data.get('alternative_names', [])  # Lista de nombres alternativos

        if alternative_names:
            chosen_name = random.choice(alternative_names)  # Se elige uno al azar
            print(f"Se eligió el nombre alternativo '{chosen_name}' para {poke_data.get('name', 'pokemon')}")
            poke_data['name'] = chosen_name  # Se reemplaza el nombre original

        card = translator.fromRequestIntoCard(poke_data)  # Se convierte en una Card
        cards.append(card)

    return cards  # Se devuelve el listado de Cards
```

- **Valores devueltos:**
  Devuelve una lista de objetos Card, cada uno representando un Pokémon.

◆ **Función: views.py > home(request)**

- **Idea general:**
  Carga las imágenes de los Pokémon y los favoritos del usuario.
- **Realización:**
  Permite mostrar al usuario las cards de los pokemons y cuales de estos están entre sus favoritos.
- **Código comentado:**

```
def home(request):
    images = services.getAllImages()
    favourite_list = services.getAllFavorites(request)
    return render(request, 'home.html', {
        'images': images,
        'favourite_list': favourite_list
    })
```

- **Parámetros:**
- request: Contiene toda la información sobre la solicitud hecha por el usuario.
- **Valores devueltos:**
  Renderiza una página html con la plantilla “home.html” y un diccionario de contexto con ‘images’ (valor que retorna services.getAllImages(), siendo una lista de cards de todos los pokemons que entrega la API externa) y ‘favourite_list’ (valor que retorna services.getAllFavorites(request), siendo una lista con las cards de todos los pokemons guardados en favoritos).

◆ **Función: Color al borde de la tarjeta/card:**

- **Idea general:**
  Agregarle/cambiar el color al borde de la tarjeta según el tipo de Pokémon
- **Realización:**
  Uso de “documentación de Bootstrap sobre Cards” (como cambiar/agregar color) y “cómo generar condicionales en Django” (el uso del if y el else).
- **Código comentado:**

```
<div class="card {% if 'fire' in img.types %}border-danger{% elif 'water' in img.types %}border-primary{% elif 'grass'

in img.types %}border-success {% else %} border-warning {% endif %} mb-3 ms-5" style="max-width: 540px; ">
```

- **Parámetros:**
  Img.Types: lista que contiene los tipos de pokemon

- **Valores devueltos:**
  El color del borde de la tarjeta según su tipo

Funcionalidades opcionales

◆ **Funcionalidad: Buscador I (filtro según nombre de Pokémon)**

- **Idea general:**
  Filtrar las tarjetas según el nombre del Pokémon.
- **Realización:**
  Permitir búsquedas parciales o completas de nombres en la interfaz de usuario.
- **Código comentado:**

```
def search(request):
    name = request.POST.get(‘query’, ‘’)
    # si el usuario ingresó algo en el buscador, se deben filtrar las imágenes por dicho ingreso.
    if(name.strip() != ‘’):
        images = services.filterByCharacter(name)
        favourite_list = services.filterFavoritesByCharacter(request, name)t

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

- **Parámetros:**
- request: Contiene toda la información sobre la solicitud hecha por el usuario.
- name: Variable extraída de request. Si bien no es un parámetro de search, se pasa como argumento a las funciones filterByCharacter y filterFavoritesByCharacter.

- **Valores devueltos:**

- Si se envía un name válido, renderiza una página html con la plantilla “home.html” y un diccionario de contexto con ‘images’ (valor que retorna services.filterByCharacter(type), siendo una lista de cards filtradas de los pokemons que entrega la API externa) y ‘favourite_list’ (valor que retorna services.filterFavoritesByCharacter(request, name), siendo una lista con las cards filtradas de los pokemons guardados en favoritos). Esta última función no estaba en el repositorio original, sino que fue añadida ya que no había una función con esta capacidad.
- Si type está vacío, redirige al usuario a la URL asociada con el nombre ‘home’ (definido en urls.py con name = ‘home’).

◆ Funcionalidad: Buscador II (filtro según tipo de Pokémon)

- **Idea general:**
  Filtrar tarjetas según el tipo de Pokémon (agua, fuego, etc.).
- **Realización:**
  Agregar un sistema de categorización visual útil para la navegación del usuario.
- **Código comentado:**

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

    if type != '':

        images = services.filterByType(type)

        favourite_list = services.filterFavoritesByType(request, type)

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

- **Parámetros:**

- request: Contiene toda la información sobre la solicitud hecha por el usuario.
- type: Variable extraída de request. Si bien no es un parámetro de filter_by_type, se pasa como argumento a las funciones filterByType y filterFavoritesByType.

- **Valores devueltos:**

- Si se envía un type válido, renderiza una página html con la plantilla “home.html” y un diccionario de contexto con ‘images’ (valor que retorna services.filterByType(type), siendo una lista de cards filtradas de los pokemons que entrega la API externa) y ‘favourite_list’ (valor que retorna services.filterFavoritesByType(request, type), siendo una lista con las cards filtradas de los pokemons guardados en favoritos). Esta última función no estaba en el repositorio original, sino que fue añadida ya que no había una función con esta capacidad.
- Si type está vacío, redirige al usuario a la URL asociada con el nombre ‘home’ (definido en urls.py con name = ‘home’).

◆ **Funcionalidad: Alta de Nuevos Usuarios:**

- **Idea general:**
  El usuario tiene la posibilidad de registrarse en la página mediante un formulario con su Nombre, Apellido, Username, Email y Contraseña. El username es único. Una vez creado el usuario, se envía un correo con las credenciales de acceso.
- **¿Por qué se hizo?:**
  Falta de un sistema de registro en la página.
- **Código Comentado:**

```
def register(request):
    form = RegisterForm(request.POST or None)
    if request.method == 'POST':
        if form.is_valid():
            firstname = form.cleaned_data['firstname']
            lastname = form.cleaned_data['lastname']
            username = form.cleaned_data['username']
            password = form.cleaned_data['password']
            email = form.cleaned_data['email']

            user = User.objects.create_user(
                first_name=firstname,
                last_name=lastname,
                username=username,
                password=password,
                email=email,
            )
            subject = 'Registro - Proyecto IP Pokedex'
            message = f'''
                Hola {firstname} {lastname},
                Gracias por registrarte en nuestra página.
                Credenciales de acceso:
                Usuario: {username}
                Contraseña: {password}
                Saludos!
                '''
            recipient = email
            send_mail(subject, message, settings.EMAIL_HOST_USER, [recipient], fail_silently=False)
            messages.success(request, 'Usuario registrado con éxito!')
            return redirect('register')
        else:
            if form.errors.get('username'):
                for e in form.errors['username']:
                    messages.error(request, e)
    return render(request, 'registration/register.html', {'form': form})
```

- **Parámetros:**

- request: Contiene toda la información sobre la solicitud hecha por el usuario.
- (firstname, lastname, username, email, password): Obtenidos de request. Parámetros del constructor del objeto User.
- (subject, message, settings.EMAIL_HOST_USER, recipient): Parámetros pasados a la función send_mail para el envío del mail. Se trata del motivo del mail, su cuerpo, correo emisor y correo receptor respectivamente.

- **Valores devueltos:**

- Si el formulario es válido, se crea un objeto con los datos, se envía un mail con las credenciales y redirecciona a la vista de registro, además de enviar un mensaje de

éxito que luego se representa en la página.
- Si el formulario o el método no es válido, renderiza la página ‘register.html’ con los valores del formulario ya escritos. Además, si el username ya existe, envía un mensaje de error.

◆ Función: services.py > getAllFavorites(request)

- **Función:**
  Obtiene los Pokémon favoritos del usuario desde la base de datos.
- **Código comentado:**

```
def getAllFavorites(request):
    if not request.user.is_authenticated:
        return []
    else:
        user = get_user(request)
        favourite_list = repositories.get_all_favourites(user)
        mapped_favourites = []

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

        return mapped_favourites
```

◆ Función: services.py > saveFavourite(request)

- **Función:**
  Convierte los datos del formulario en una Card y los guarda.
- **Código comentado:**

```
def saveFavourite(request):
    fav = translator.fromTemplateIntoCard(request)
    fav.user = get_user(request)
    return repositories.save_favourite(fav)
```

Función: Modificación en home.html

Agregué inputs ocultos en el formulario para enviar correctamente los datos, incluyendo el id, que antes faltaba y causaba errores. También implementé el botón ❤ para añadir a favoritos, y su versión deshabilitada ✔ si ya fue añadido.

- **Código comentado:**

```
<form method="post" action="{% url 'agregar-favorito' %}">
    {% csrf_token %}
    <input type="hidden" name="id" value="{{ img.id }}">
    <input type="hidden" name="name" value="{{ img.name }}">
    <input type="hidden" name="height" value="{{ img.height }}">
    <input type="hidden" name="weight" value="{{ img.weight }}">
    <input type="hidden" name="types" value="{{ img.types }}">
    <input type="hidden" name="image" value="{{ img.image }}">
    {% if img.name in favorite_names %}
        <button type="submit" class="btn btn-primary btn-sm float-left" disabled>✔ Favoritos</button>
    {% else %}
        <button type="submit" class="btn btn-primary btn-sm float-left">❤ Favoritos</button>
    {% endif %}
</form>
```

◆ Funcionalidad: Loading Spinner para la carga de imágenes

- **Descripción general**

Para mejorar la experiencia del usuario durante la carga de imágenes, se implementó un loading spinner animado en forma de espiral de círculos. Esta funcionalidad fue agregada al archivo home.html y complementada con un nuevo archivo CSS llamado spinner.css.
El objetivo fue que el spinner se muestre mientras se cargan las imágenes de los Pokémon, y desaparezca automáticamente cuando todas las imágenes hayan terminado de cargarse. Además, también se configuró para que el spinner aparezca al momento de enviar formularios de búsqueda o filtrado por tipo.

- **Funcionalidades principales**
- **Loading Spinner al iniciar la carga de imágenes (static/spinner.css)**
- **Idea general:**

  La función del spinner es mostrar una animación visual al usuario mientras se están cargando las imágenes de los Pokémon, sobre todo cuando la carga se demora por motivos de red o cantidad de datos y para que no piense que la app se congeló o está fallando. Este tipo de feedback visual mejora la usabilidad, especialmente si se están cargando muchos datos desde una API externa.
- **Realización:**

  Se hizo porque al consultar múltiples imágenes desde la API, dependiendo de la conexión del usuario, la espera podía ser incómoda o confusa. El spinner indica visualmente que el sistema está funcionando y solo necesita tiempo para cargar los datos.

  Se creó un archivo spinner.css con los estilos de la animación en espiral.

- **Código implementado:**

  Idea: Crear una animación circular de 12 puntos que simulan un espiral animado.

```
/* Fondo semitransparente */
.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;
}


/* Contenedor del spinner */
.spinner {
  width: 64px;
  height: 64px;
  position: relative;
}


/* 12 puntos que giran */
.spinner div {
  position: absolute;
  width: 5px;
  height: 5px;
  background: #333;
  border-radius: 50%;
```

```
  animation: spinnerAnim 1.2s linear infinite;
}

/* Ubicación y delay de cada punto /
.spinner div:nth-child(1) { top: 29px; left: 53px; animation-delay: 0s; }
/*... hasta nth-child(12) con distintos delays ... */

@keyframes spinnerAnim {
    0% {
        transform: scale(1);
        opacity: 1;
    }
    100% {
         transform: scale(0.3);
         opacity: 0.3;
    }
}
```

Archivo: home.html (en templates/app/)

1. Fragmento agregado en home.html

- Se incluyo el spinner al cargar la vista, antes del contenido principal:

```
<!-- Sección agregada al principio de home.html -->
{% load static %}
<link rel="stylesheet" href="{% static 'spinner.css' %}">
<div id="spinnerOverlay" class="spinner-overlay">
 <div class="spinner">
   <div></div><div></div>...<div></div> <!-- 12 divs en total -->
 </div>
</div>
```

2. Script de control de carga de imágenes

- Detectar cuando todas las imágenes hayan sido cargadas, y esconder el spinner:

```
<script>
 window.addEventListener('load', function () {
   const spinner = document.getElementById('spinnerOverlay');
   const images = document.querySelectorAll('img');
   let loadedCount = 0;
   const totalImages = images.length;

   if (totalImages === 0) {
     spinner.style.display = 'none';
   }




   images.forEach(function(img) {
     if (img.complete) {
       loadedCount++;
     } else {
       img.addEventListener('load', function () {
         loadedCount++;
         if (loadedCount === totalImages) {
           spinner.style.display = 'none';
         }
       });
     }
   });

   if (loadedCount === totalImages) {
     spinner.style.display = 'none';
```

```
  }
 });
</script>
```

3. Script para mostrar el spinner al enviar formularios

- Mostrar el spinner mientras se espera la respuesta del servidor:

```
<script>
 document.addEventListener('DOMContentLoaded', function () {
  const spinner = document.getElementById('spinnerOverlay');
  const forms = document.querySelectorAll('form');

  forms.forEach(function (form) {
   form.addEventListener('submit', function () {
    spinner.style.display = 'flex';
   });
  });
 });
</script>
```

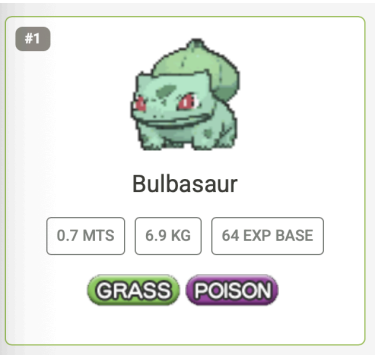
◆ Funcionalidad: Rediseño de la Interfaz Gráfica

- **Idea general:**
  Implementación de un rediseño visual en la página, añadiendo un estilo global de bootstrap y cambiando el código de varios templates.
- **¿Por qué se hizo?:**
  Si bien no es una funcionalidad esencial para el funcionamiento de la aplicación, hace a la página más atractiva para el usuario.
- **Código comentado:**

```
# Código para las cards en home.html

<div class="col">
        <div class="card position-relative mb-0 ms-2
         {% if 'fire' in img.types %}border-danger
         {% elif 'water' in img.types %}border-primary
         {% elif 'grass' in img.types %}border-success
         {% else %}border-warning
         {% endif %}" style="max-width: 19rem;">
         <span class="position-absolute top-0 start-0 m-2 badge bg-secondary">#{{ img.id }}</span>
         <div class="card-body d-flex flex-column align-items-center">
          <img src="{{ img.image }}" class="card-img-top mb-3" style="width:96px;height:96px;" alt="{{
img.name }}">
          <h5 class="card-title text-center mb-3">{{ img.name }}</h5>
          <div class="d-flex justify-content-center mb-3">
           <div class="d-flex flex-column align-items-center mx-1">
            <button type="button" class="btn btn-outline-dark btn-sm" disabled>{{ img.height }}</button>
           </div>
           <div class="d-flex flex-column align-items-center mx-1">
            <button type="button" class="btn btn-outline-dark btn-sm" disabled>{{ img.weight }}</button>
           </div>
           <div class="d-flex flex-column align-items-center mx-1">
            <button type="button" class="btn btn-outline-dark btn-sm" disabled>{{ img.base }} EXP
```

```
Base</button>
                </div>
              </div>

              <div class="w-100">
                <div class="d-flex justify-content-center mb-3">
                  {% for icon_url in img.type_icons %}
                    <img src="{{ icon_url }}" alt="type" class="me-1">
                  {% endfor %}
                </div>

              </div>
            </div>
            {% if request.user.is_authenticated %}
            <div class="card-footer text-center">
              <form method="post" action="{% url 'agregar-favorito' %}">
                {% csrf_token %}
                <input type="hidden" name="name" value="{{ img.name }}">
                <input type="hidden" name="height" value="{{ img.height }}">
                <input type="hidden" name="weight" value="{{ img.weight }}">
                <input type="hidden" name="types" value="{{ img.types }}">
                <input type="hidden" name="image" value="{{ img.image }}">
                {% if img.name in favorite_names %}
                <button type="submit" class="btn btn-primary btn-sm" style="color:white" disabled>✔️
Favoritos</button>
                {% else %}
                <button type="submit" class="btn btn-primary btn-sm" style="color:white">❤️
Favoritos</button>
                {% endif %}
              </form>
            </div>
            {% endif %}
          </div>
        </div>
```

Comparación entre versiones:



3. Conclusiones

Este trabajo permitió desarrollar una funcionalidad completa, reutilizable y clara, que cumple con los objetivos de la consigna. Las funciones implementadas están correctamente estructuradas y pensadas para integrarse fácilmente con otros módulos del proyecto.
Se superaron varias dificultades técnicas (Git, manejo de múltiples archivos, comprensión de consignas poco claras) y se logró un resultado que, además de funcional, es fácil de mantener. La solución es flexible y se adapta a futuras extensiones del sistema.

La implementación del spinner fue efectiva para mejorar la experiencia del usuario, especialmente cuando hay demoras en la carga de datos. Se utilizaron herramientas como HTML, CSS y JavaScript, que fueron integradas a la estructura existente de Django.
Una de las principales dificultades fue lograr que el spinner se escondiera solo cuando todas las imágenes estuvieran completamente cargadas, lo cual se resolvió con JavaScript y viendo los eventos load de cada imagen.