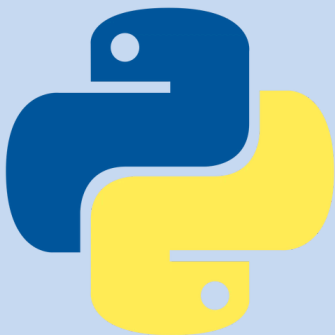


TRABAJO PRACTICO DE PROGRAMACIÓN: DISEÑAMOS UNA POKÉDEX

INTEGRANTES DEL GRUPO:

- Cendon Ignacio
- Joaquín Arenas
- Agustín Levrino

LENGUAJES DE PROGRAMACIÓN UTILIZADOS:



PROYECTO POKÉDEX UNGS 2025

Introducción

Este proyecto consiste en desarrollar una aplicación que simula una Pokédex, el dispositivo utilizado en la reconocida serie Pokémon para obtener información sobre las distintas criaturas. El objetivo principal fue aplicar los contenidos aprendidos durante el cuatrimestre, complementándolos con investigaciones y tutoriales sobre temas adicionales.

Desarrollo del trabajo

Durante el proceso de trabajo, abordamos distintos desafíos que nos permitieron no solo aplicar los conocimientos técnicos, sino también mejorar nuestra organización como grupo. La Pokédex muestra datos como nombre, peso, altura y nivel base de los Pokémon. Para lograrlo, utilizamos diversas herramientas vistas en clase, y otras nuevas que fuimos incorporando.

.gitignore

Antes de comenzar con la codificación principal, implementamos un archivo .gitignore con el objetivo de evitar la subida de archivos no deseados al repositorio, como documentos .docx, o .txt. Esto ayudó a mantener el proyecto limpio y enfocado en el código relevante.

Funciones implementadas:

getAllImages() – services.py

Obtiene la lista completa de Pokémon desde la API y los transforma en 'cards', es decir, estructuras visuales listas para mostrarse. Esta separación facilita un posible cambio de fuente de datos en el futuro.

```
def getAllImages():
    cards = cache.get('poke_cards')
    if cards is not None:
        return cards
    result=transport.getAllImages()
    if result is None:
        return []
    cards=[]
    for p in result:
        card = translator.fromRequestIntoCard(p)
        cards.append(card)
    cache.set('poke_cards', cards, timeout=600)
```

filterByCharacter(name) – services.py

Filtra los Pokémon por nombre, ignorando mayúsculas y minúsculas, para hacer la búsqueda más flexible y tolerante a errores, la decision que tomamos para simplificar la logica fue usar `.lower()` para no tener que validar error de escritura.

```
def filterByCharacter(name):
    filtered_cards = []

    for card in getAllImages():
        if name.lower() in card.name.lower():
            filtered_cards.append(card)

    return filtered_cards
```

filterByType(type_filter) – services.py

Devuelve solo los Pokémon cuyo tipo coincide con el seleccionado por el usuario. Se utiliza una lista auxiliar en minúsculas para asegurar comparaciones correctas.

```
def filterByType(type_filter):
    filtered_cards = []

    for card in getAllImages():
        if type_filter.lower() in [t.lower() for t in card.types]:
            filtered_cards.append(card)

    return filtered_cards
```

home(request) – views.py

Carga todos los Pokémon y los envía al HTML principal para mostrarlos. Si el usuario no está logueado, la lista de favoritos se mantiene vacía, simplificando la lógica.

```
def home(request):
    images = services.getAllImages()
    favourite_list = []

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

search(request) – views.py

Procesa el texto del buscador y filtra los resultados. Si el input está vacío, redirige al inicio para mantener una interfaz clara.

```
def search(request):
    name = request.POST.get('query', '')

    if name != '':
        images=services.filterByCharacter(name)
        favourite_list=[]

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

filter_by_type(request) – views.py

Recibe el tipo seleccionado (fuego, agua, planta) y filtra los Pokémon correspondientes. Requiere método POST para evitar accesos accidentales.

```
def filter_by_type(request):
    if request.method == "POST":
        type = request.POST.get("type")
        images = services.filterByType(type)
        favourite_list = []

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

Savefavourites - views.py

Se desarrollo la función savefavourites, donde se usa el modelo Favourite de django y se crea el objeto con el pokemon que se selecciono con la información del pokemon que proviene del botón agregar.

Se desarrollo la función para eliminar un favorito, al igual que la de savefavourites se hizo con la condición @login_required para que solo sea accesible a un usuario que haya iniciado sesión

En ambos casos, al agregar o eliminar se muestra un mensaje con el nombre del pokemon. Se integro una logica con la nueva variable de identificacion, con una condicion para detectar errores. Esto debido a que se presentaban errores a la hora de agregar el mismo pokemon en 2 usuarios distintos.

```
def saveFavourite(request):
    if request.method == 'POST':
        poke_id_str = request.POST.get('id')
        try:
            poke_id = int(poke_id_str)
        except (TypeError, ValueError):
            messages.error(request, "ID de Pokémon inválido.")
            return redirect('buscar')

        exists = Favourite.objects.filter(user=request.user, poke_id=poke_id).exists()
        if not exists:
            Favourite.objects.create(
                poke_id=poke_id,
                name=request.POST.get('name'),
                height=request.POST.get('height'),
                weight=request.POST.get('weight'),
                types=request.POST.get('types'),
                image=request.POST.get('image'),
                user=request.user
            )
            nombre = request.POST.get('name')
            messages.success(request, f"✓ Se agregó el pokemon {nombre} a tus favoritos.")
        else:
            messages.info(request, f"El pokemon ya está en tus favoritos.")
    return redirect('buscar')
```

Getallfavouritesbyuser - views.py

Se modificó la función `getallfavouritesbyuser` para que obtenga la lista de favoritos de el usuario, esto es útil para el botón dinámico y la página de favoritos.

```
def getAllFavouritesByUser(request):
    if request.method == 'GET':
        favourite_list = Favourite.objects.filter(user=request.user)
        return render(request, 'favourites.html', {'favourite_list': favourite_list})
    else:
        return redirect('home')
```

Se agregaron estas líneas en `home.html` y `favourites.html` para poder mostrar mensajes.

```
2  {% block content %}
3  {% if messages %}
4  {% for message in messages %}
5      <div class="alert alert-{{ message.tags }}">{{ message }}</div>
6  {% endfor %}
7  {% endif %}
```

home(request) - views.py

Se retoco la función home para que defina una lista de favoritos convertida a string, para que luego en home.html el botón dinámico responda correctamente y se muestre como agregar o ya agregado. Luego la función search se modifico de la misma forma para el mismo fin.

```
16 def home(request):
17     images = services.getAllImages()
18     # Definir favoritos del usuario en una lista convertida en id para poder mostrar el boton de que el favorito ya
19     favourite_list = Favourite.objects.filter(user=request.user)
20     favourite_ids = [str(fav.id) for fav in favourite_list]
21     return render(request, 'home.html', {
22         'images': images,
23         'favourite_list': favourite_list,
24         'favourite_ids': favourite_ids,
25     })
26 # función utilizada en el buscador.
27 def search(request):
28     name = request.POST.get('query', '')
29
30     if name != '':
31         images = services.filterByCharacter(name)
32         return render(request, 'home.html', {
33             'images': images,
34             'favourite_list': favourite_list,
35             'favourite_ids': favourite_ids,
36         })
37     return redirect('home')
```

Se importo el recurso favourite para armar la función de favoritos. Se importo el recurso messages para mostrar mensaje de se ha agregado un favorito.

```
8 from .models import Favourite
9 from django.contrib import messages #SE importa para enviar el mensaje de ya esta en favoritos
```

home.html

Se revisaron los parámetros de el botón dinámico que muestra agregar o favorito en home.html ya que no funcionaba, se aplico otra lógica para que quede funcional.

```

{% end_for %}
<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="base_experience" value="{{ img.base_experience }}">
<input type="hidden" name="types" value="{{ img.types|join:' ' }}">
<input type="hidden" name="image" value="{{ img.image }}">
{% if img.id|stringformat:"s" in favourite_ids %}
    <button class="btn btn-sm btn-primary" disabled>✔ Favorito</button>
{% else %}
    <button class="btn btn-sm btn-primary">❤ Agregar</button>
{% endif %}

```

class favourite - models.py

Se cambio el modelo Favourite, se incluyo una nueva variable de informacion de los pokemon, ya que la variable ID no permitia que el mismo pokemon fuese agregado por dos usuarios distintos. Se tuvo que hacer migraciones en la base de datos.

```

5 class Favourite(models.Model):
6     # Detalles del pokemon.
7     id = models.AutoField(primary_key=True)
8     poke_id = models.IntegerField() #usar id como primary key generaba problemas a la hora de anadir el mismo pokemon en distintos usuarios y se cambio la logica
9     name = models.CharField(max_length=200) # Nombre del personaje
10    height = models.CharField(max_length=200) # Altura
11    weight = models.CharField(max_length=200) # Peso
12    base_experience = models.IntegerField(null=True, blank=True) # Experiencia base
13    types = models.JSONField(default=list) # Lista de tipos (ej: ["grass", "poison"])
14    image = models.URLField() # URL de la imagen

```

Registrar_usuario - [views.py](#) , registro.html y login.html

Se creo la función para registrar usuario y la pagina registro que es lo que el usuario verá para registrarse, con dos campos obligatorios, usuario y contraseña. Se integro el mensaje de usuario creado, usuario existente y falta de uno de los campos, todos con su respectiva logica. Tambien se añadio la logica para mostrar mensajes en la pagina login.html.

```

9  def registrar_usuario(request):
10     if request.method == 'POST':
11         username = request.POST.get('username', '').strip()
12         password = request.POST.get('password', '').strip()
13         if not username or not password:
14             messages.error(request, 'Se requiere de usuario y contraseña.')
15         elif User.objects.filter(username=username).exists():
16             messages.error(request, 'Ese nombre de usuario ya existe.')
17         else:
18             User.objects.create_user(username=username, password=password)
19             messages.success(request, 'Usuario registrado exitosamente.')
20             return redirect('login')
21     return render(request, 'registro.html')

```

Explicación de la interfaz HTML

Estructura general


- **home.html:** es la página principal, con:
 - Buscador arriba con botón “Buscar”.
 - Botones de tipo con emojis (🔥 fuego, 💧 agua, 🌿 planta).
 - Tarjetas de Pokémon con imagen, nombre, tipo, altura, peso y experiencia.
 - Botón de “Agregar a favoritos” si estás registrado.
- **favourites.html:**
 - Muestra los Pokémon que marcaste como favoritos.
 - Se usa una tabla con columnas: imagen, nombre, altura, peso.

- Tiene botón para eliminar favoritos.
- **header.html:**
 - Encabezado que aparece en todas las páginas.
 - Tiene enlaces a “Inicio”, “Favoritos” y “Cerrar sesión”, “Ingresar” o “registrarse”
- **footer.html:**
 - Pie de página con info de la materia y versión del trabajo.
- **index.html:**
 - Es la portada del sitio, muestra un mensaje de bienvenida.
 - Tiene botón para entrar al buscador.

Dificultades generales:

La mayor dificultad que tuvimos fue encontrarnos en el trabajo con el manejo de html y css, que eran idiomas de programación con los que todavía no nos habíamos familiarizado, por lo tanto, tuvimos que buscar información y ver videos para poder avanzar con el trabajo y el correcto funcionamiento de la Pokedex.

- Problemas de conexión con la API al inicio, solucionados mediante manejo de errores (try/except).
- Errores de indentación en HTML que requerían reorganización de bloques for, if y endblock.
 - Al conectar con la API externa, algunas funciones fallaban cuando un Pokémon no se encontraba. Se mejoró con manejo de errores y try/except.
- Para evitar llamados a la API, se uso cache para guardar resultados durante 10 minutos.

- El botón de favoritos fue una decisión clave: se bloquea si ya está agregado, y muestra un  si ya fue marcado.
- Se optó por usar Bootstrap para mejorar la estética y mantener consistencia en toda la aplicación.

Conclusión

Este trabajo práctico nos permitió desarrollar un sistema web funcional y visualmente atractivo que simula el comportamiento de una Pokédex. A lo largo del desarrollo se integraron diversas funcionalidades claves como la búsqueda de Pokémon por nombre, filtrado por tipo, y gestión de favoritos, utilizando datos externos en tiempo real.

La experiencia implicó tanto desafíos técnicos como decisiones de diseño, lo cual fomentó el aprendizaje de buenas prácticas de organización del código, manejo de errores, trabajo en capas y diseño de interfaz orientado al usuario.

Además, fue fundamental aplicar Bootstrap para lograr una interfaz coherente y accesible, mejorando así la experiencia general del usuario. El control de acceso mediante login, y la interacción con una API real consolidaron una base sólida de conocimientos prácticos aplicables a futuros proyectos.