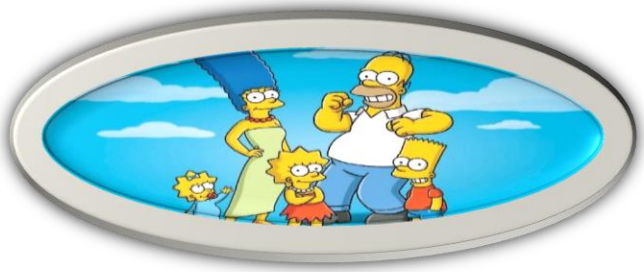
Universidad Nacional de General Sarmiento

Tecnicatura en Informática



Trabajos Práctico
The Simpsons

Introducción a la Programación

(Intensivo de verano-2026)

Resumen:

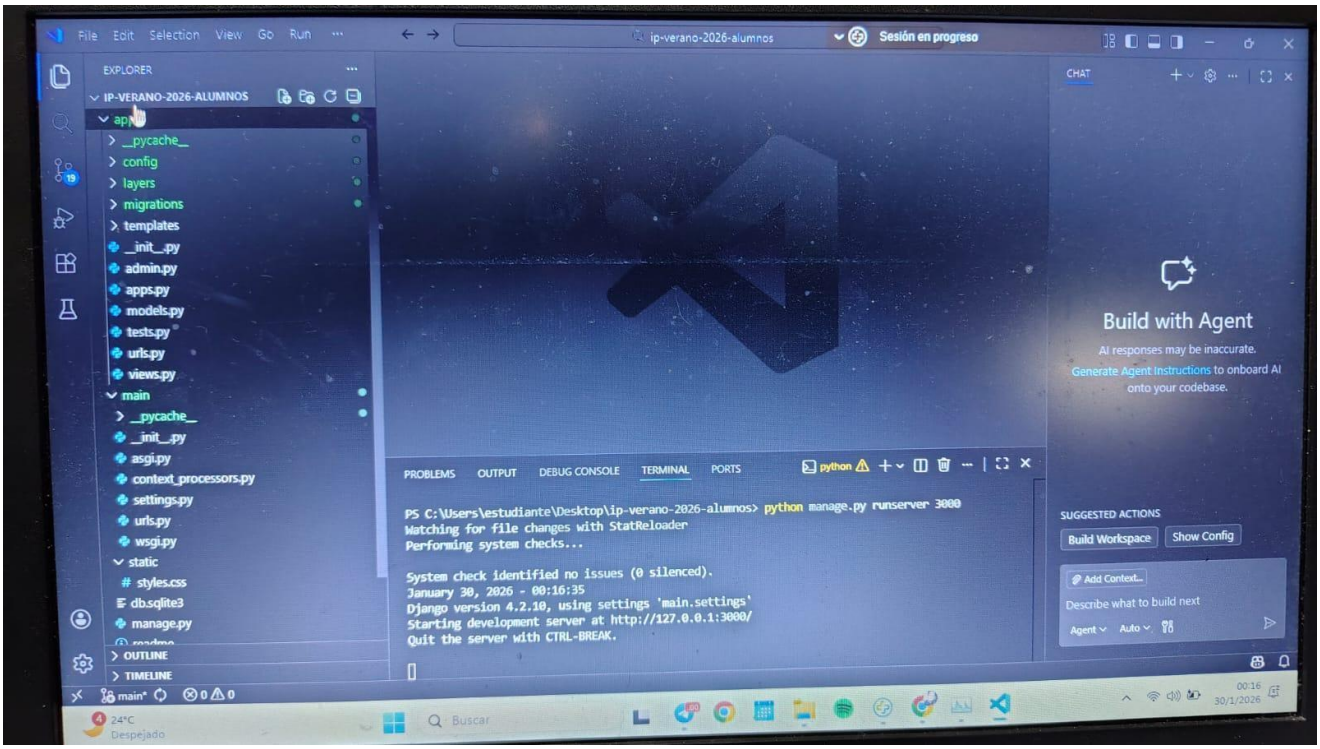
Desarrollo de una aplicación web utilizando el framework Django para visualizar una galería de personajes de "Los Simpsons", obteniendo los datos en tiempo real desde una API externa.

Integrantes:

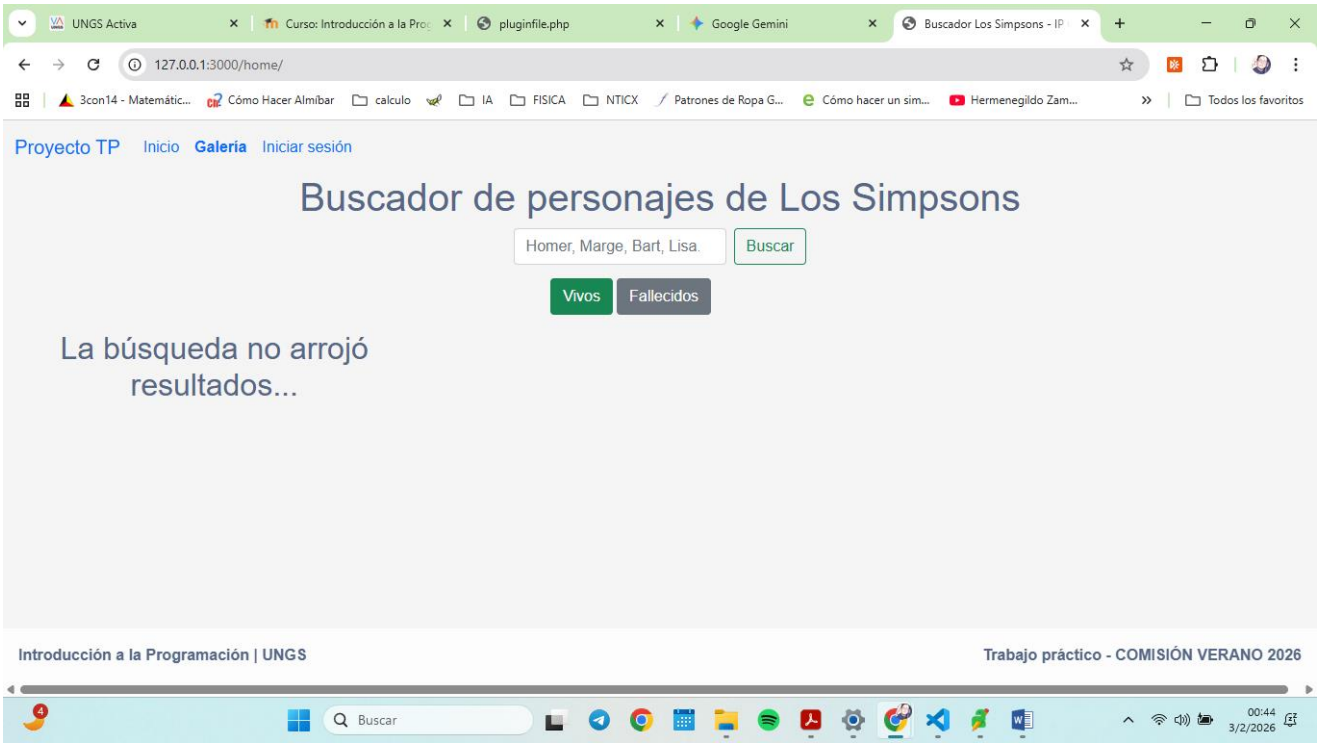
- Moreyra Silvia S. EMail: moreyra1974@gmail.com
- Cruz Daiana EMail:cruzdaiana5@gmail.com
- Agustín Aguero Email: agustin.aguero499@gmail.com

1. ***Introducción***

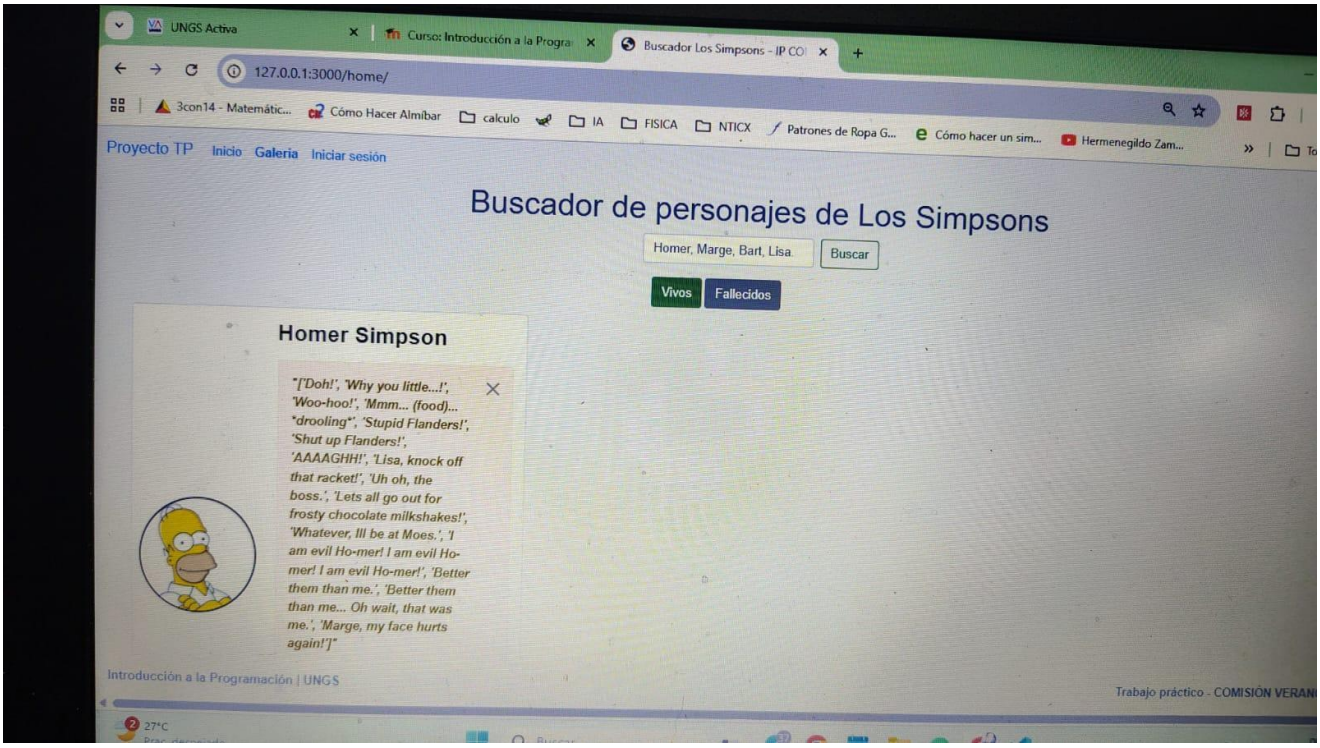
El objetivo de este trabajo es aplicar los conceptos de programación modular y arquitectura por capas. El problema principal consistió en conectar nuestra aplicación con un servidor externo (API) para traer información (nombres, frases e imágenes) y mostrarla de forma organizada en una página web. Se buscó separar las responsabilidades del código para que sea más fácil de mantener y corregir.



Inicio del trabajo



Se generó un error donde no se podía comunicar con la API



Listo¡¡¡¡

2. Desarrollo

Para resolver el problema de la falta de datos, se trabajó siguiendo la arquitectura de **capas** del proyecto. La solución no fue solo escribir código nuevo, sino investigar por qué la conexión estaba fallando y adaptar los componentes existentes.

➢ Principales dificultades encontradas

Durante el proceso, nos enfrentamos a tres problemas críticos:

1. **Error 410 (Gone):** Al ejecutar el servidor, la terminal indicaba que la URL de la API original ya no existía.
2. **Incompatibilidad de datos:** Al cambiar a una nueva API (Glitch), los nombres de los datos que recibíamos (como el nombre del personaje o la imagen) no coincidían con lo que nuestro programa esperaba.
3. **Error de decodificación (JSON):** Hubo momentos donde el programa "explotaba" (pantalla amarilla) porque intentaba leer información de una dirección que devolvía un error en lugar de datos.

➢ Soluciones y decisiones tomadas

Para superar estos obstáculos, tomamos las siguientes decisiones técnicas:

- **Migración de Endpoints:** Decidimos modificar el archivo config.py para apuntar a un servidor de respaldo (thesimpsonsquoteapi.glitch.me). Esto permitió restablecer el flujo de información.
- **Ajuste del Traductor (Mapeo):** Modificamos la función fromRequestIntoCard en la capa de utilidades. Como la nueva API usaba la clave 'character' en lugar de 'name', ajusté el código para que el programa pudiera reconocer los datos correctamente.
- **Manejo de seguridad en la conexión:** Como las direcciones de internet pueden dejar de funcionar de un momento a otro (como nos pasó con el Error 410), configuré una regla de seguridad. Si el programa intenta buscar a los personajes y no los encuentra, el sistema ya sabe que debe avisar que no hay resultados en vez de fallar. Esto hace que la página sea más estable y no se cierre ante problemas externos.

2.1 Descripción general

La solución consistió en crear una aplicación web capaz de conectarse con un servidor externo para obtener información de "Los Simpsons". Para que el código sea ordenado y fácil de corregir, dividimos el trabajo en tres partes o "capas": una que se encarga de viajar a internet a buscar los datos (Transporte), otra que los organiza (Servicios) y una última que traduce esos datos técnicos en fichas con fotos y nombres que el usuario puede ver (Traductor). Gracias a esta organización, cuando el servidor externo falló, pudimos arreglar solo la parte necesaria sin tener que rehacer todo el proyecto.

2.2 Funcionalidades principales:

La funcionalidad principal es la **"Galería Dinámica de Personajes"**. Su objetivo es permitir que el usuario vea una lista de personajes y pueda buscar a uno en particular por su nombre. Se resolvió conectando nuestro código a una API (un servidor de datos). Cuando el usuario escribe un nombre, el sistema hace un pedido a internet, recibe una respuesta con texto y fotos, y la muestra de forma atractiva en la pantalla.

Detalle de funciones implementadas*:*
Función 1: getAllCharacters (Capa de Transporte)

- **Idea general:** Es la función "mensajera". Va hasta lo la dirección de internet de la API y trae la información cruda.
- **Por qué se hizo:** Para separar la conexión de red del resto del programa. Si el servidor de internet cambia, solo tocamos esta función.
- **Código comentado:**

```
def getAllCharacters(input=None):
    # Elegimos la URL: si hay una búsqueda usamos filtros, si no, traemos 50 personajes
    if input:
        url = f"{config.SIMPSONS_API_BASE_URL}/quotes?count=50&character={input}"
    else:
        url = config.SIMPSONS_CHARACTERS_URL

    try:
        # Intentamos pedir los datos al servidor
        response = requests.get(url, timeout=10)
        response.raise_for_status() # Verificamos que la conexión sea exitosa
        return response.json()      # Devolvemos los datos en formato de lista
    except Exception as e:
        # Si algo falla (como el error 410), devolvemos una lista vacía para no romper la web
        print(f"Error de conexión: {e}")
        return []
```

✓ **Parámetros:** Recibe un input (opcional), que es el nombre del personaje que el usuario escribió en el buscador.

✓ **Valores que devuelve:** Devuelve una lista con los datos de los personajes encontrados o una lista vacía si hubo un error.

Función 2: fromRequestIntoCard (Capa de Utilidades / Traductor)

- **Idea general:** Es la función "traductora". Toma los datos técnicos que vienen de internet y los guarda en un formato que nuestro programa entiende (llamado Card).
- **Por qué se hizo:** Porque las APIs suelen usar nombres de etiquetas en inglés o diferentes a los nuestros. Esta función asegura que el nombre del personaje siempre se guarde donde corresponde.
- **Código comentado:**

```
def fromRequestIntoCard(object):
    # Extraemos la dirección de la foto que viene de la API
    image_url = object.get('image')

    # Creamos una 'Card' (ficha) asignando cada dato de la API a nuestro modelo
    card = Card(
        name=object.get('character'),   # Guardamos el nombre del personaje
        phrases=object.get('quote'),     # Guardamos su frase típica
        image=image_url,                # Guardamos la URL de su imagen
        # Los demás campos se completan si están disponibles
        gender=object.get('gender'),
        status=object.get('characterDirection')
    )
    return card
```

✓ **Parámetros:** Recibe un object (un diccionario con los datos crudos de un solo personaje).

✓ **Valores que devuelve:** Devuelve un objeto de tipo Card listo para ser mostrado en la galería.

3. Conclusiones:

En conclusión, en el trabajo práctico **The Simpson** se logró desarrollar una aplicación web funcional utilizando el framework **Django**, aplicando correctamente el patrón **MVC** mediante la separación en *views, services y repositories*.

Se implementaron funcionalidades principales como el inicio de sesión, la visualización de personajes y la gestión de favoritos (agregar, listar y eliminar), utilizando el ORM de Django para la conexión con la base de datos.

Además, se aplicaron validaciones de seguridad como el uso de @login_required y el control de permisos para asegurar que cada usuario solo pueda administrar sus propios favoritos.

También, el sistema demuestra un correcto funcionamiento y permite una experiencia ordenada e intuitiva para el usuario, cumpliendo con los objetivos planteados en el trabajo práctico.

Finalmente, este proyecto nos permitió entender la importancia de la arquitectura por capas. Aprendimos que, aunque una API externa falle, si el código está bien organizado, se puede solucionar el problema tocando solo una parte del sistema sin afectar el resto.
Con ayuda del profesor nos mostró a ver cómo ir revisando las capas dentro del code, poder ver lo escrito, lo borrado y lo sobre escrito a tal punto de volver al principio.
Además, cuando la página no cargaba, dentro de ese error poder leer e interpretar lo que nos estaba

ANEXO 1

- **Personalización dinámica de los bordes**

Para mejorar la visualización de los personajes, implementamos una estructura condicional en el template utilizando las etiquetas de Django ({% if %}, {% elif %}, {% else %}).

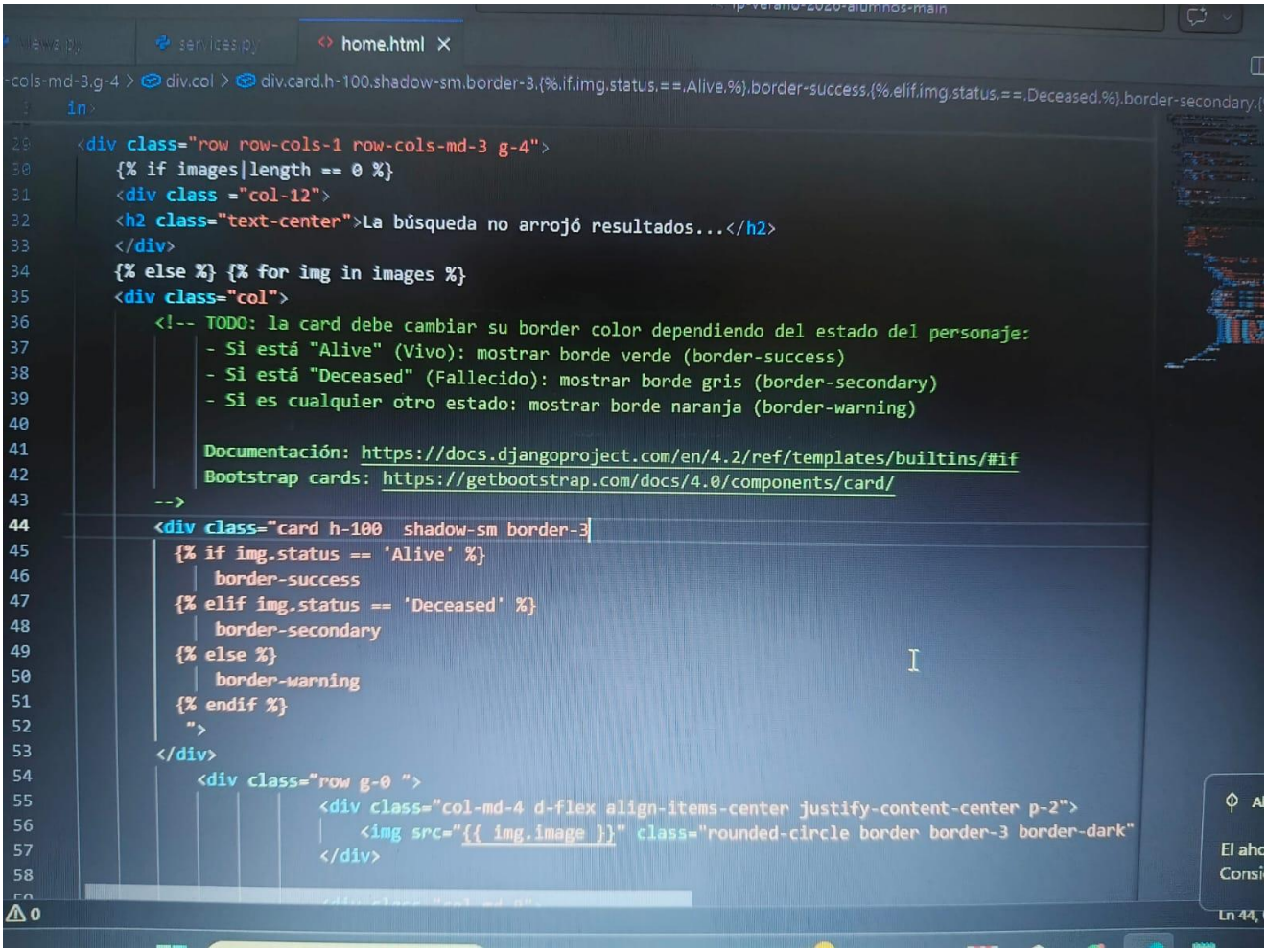
El objetivo fue que el color del borde de cada card cambie automáticamente según el estado del personaje recibido desde la API.

Si el estado es "Alive", se aplica la clase border-success, mostrando el borde en verde.

Si el estado es "Deceased", se aplica border-secondary, mostrando el borde en gris.

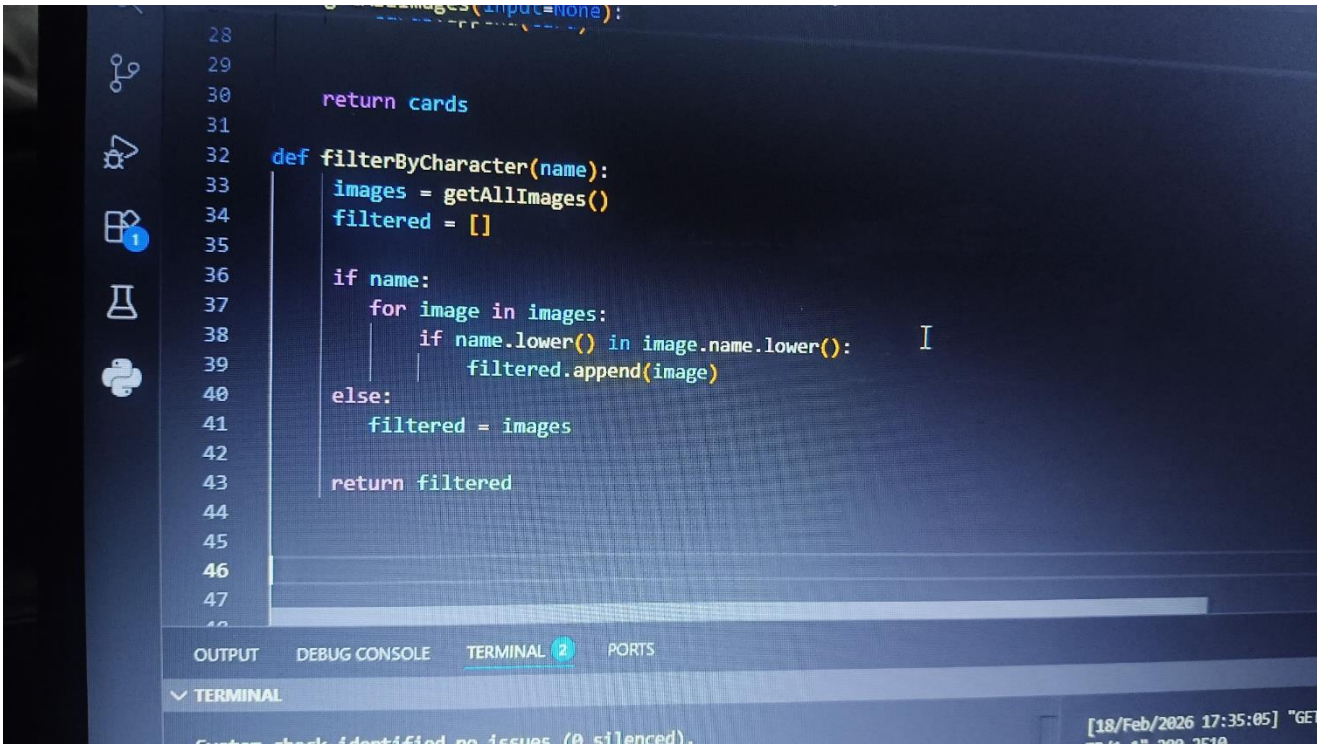
En cualquier otro caso, se utiliza border-warning, mostrando el borde en naranja

Esto lo realizamos combinando lógica de Django en el template con clases de Bootstrap, logrando que el diseño sea dinámico y que la información visual refuerce el estado del personaje. Usamos divs para organizar la estructura y cerrarlos correctamente para que no se rompa el diseño. Gracias a eso y a los Bootstrap que es un framework de CSS que utilizamos para mejorar el diseño de la aplicación. Nos permitió usar clases ya definidas para organizar las cards en filas y aplicar colores a los bordes .



ANEXO 2

❖ Buscador de Los Simpsons 1



- **Función filterBy**

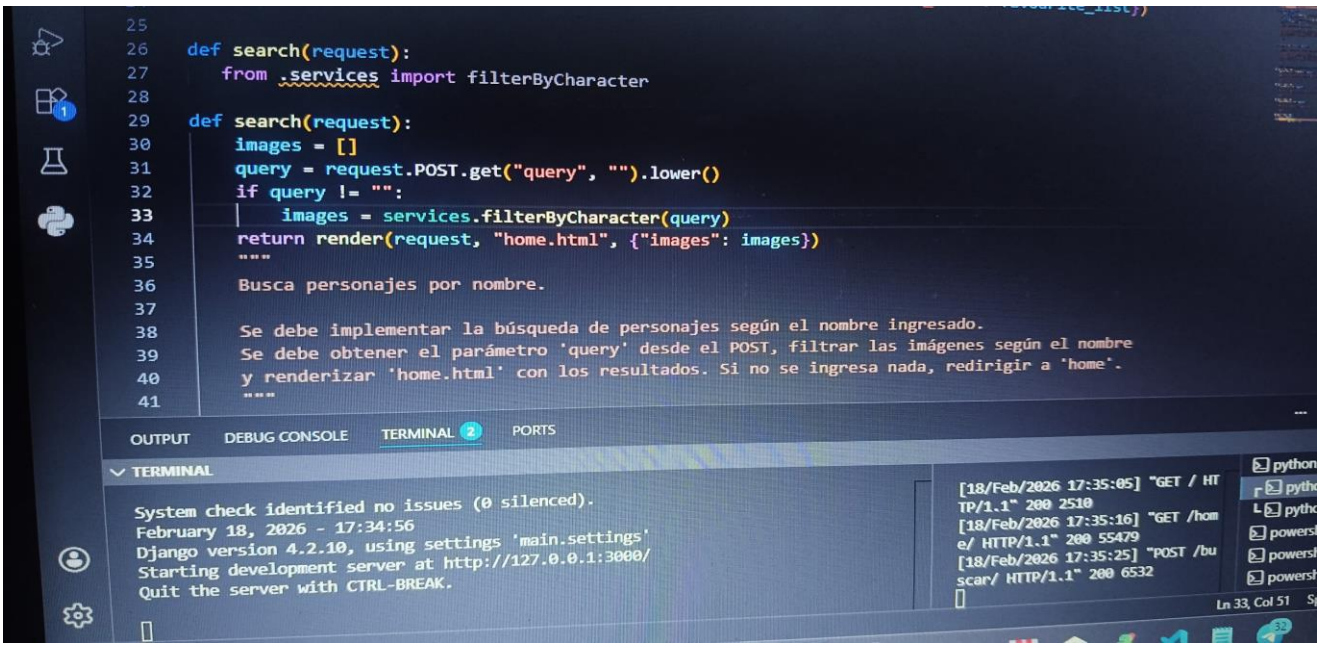
filterByCharacter(name)

La función filterByCharacter fue creada para implementar la lógica del buscador dentro de la aplicación. Su objetivo es filtrar las imágenes disponibles según el nombre del personaje que el usuario ingrese.
En primer lugar, la función recibe como parámetro name, que representa el texto escrito por el usuario en el campo de búsqueda. Luego, se llama a getAllImages() para obtener la lista completa de imágenes disponibles en la aplicación. Esto permite trabajar con el conjunto total de datos antes de aplicar cualquier criterio de filtrado.
A continuación, se crea una lista vacía llamada filtered, que será utilizada para almacenar únicamente las imágenes que coincidan con la búsqueda realizada.
Posteriormente, se verifica si el usuario ingresó algún texto. Si el valor de name contiene información, la función recorre todas las imágenes mediante un bucle for. En cada iteración, compara el texto ingresado con el nombre de la imagen (image.name). Para evitar problemas derivados de diferencias entre mayúsculas y minúsculas, se utiliza el método. lower () en ambos valores antes de realizar la comparación. De esta manera, la búsqueda se vuelve más flexible y no depende del formato en que el usuario escriba el nombre.
Si el texto ingresado se encuentra dentro del nombre de la imagen, dicha imagen se agrega a la lista filtered. En cambio, si el usuario no escribió nada en el buscador, la función devuelve directamente la lista completa de imágenes, evitando aplicar el filtro innecesariamente.
Finalmente, la función retorna la lista filtered, que contiene únicamente las imágenes que cumplen con la condición de búsqueda.

- **Decisión de diseño**

Esta función fue ubicada en el archivo services.py para respetar el principio de separación de responsabilidades. La vista (views.py) se encarga exclusivamente de recibir la solicitud del usuario y renderizar la respuesta, mientras que la lógica de filtrado se delega a la capa de servicios. Esta organización permite que el código sea más claro, reutilizable y fácil de mantener a medida que el proyecto evoluciona.

Una vez implementada la lógica de filtrado en la capa de servicios, fue necesario crear una vista que permitiera recibir el texto ingresado por el usuario desde el formulario y utilizar dicha función para mostrar los resultados en pantalla. Para ello, se desarrolló la función search(request), que actúa como intermediaria entre la interfaz y la lógica del sistema.



La función search(request) es una vista de Django cuya finalidad es procesar la búsqueda de personajes realizada por el usuario desde el formulario del sitio. En una aplicación web, la vista cumple el rol de intermediaria entre la petición que realiza el usuario y la respuesta que devuelve el sistema. En este caso, la función recibe el texto ingresado en el buscador, lo procesa y devuelve únicamente las imágenes que coinciden con ese criterio.
La función comienza recibiendo como parámetro el objeto request, que contiene toda la información relacionada con la petición HTTP realizada por el usuario, incluyendo los datos enviados desde el formulario. Luego, se inicializa una lista vacía llamada images, que será utilizada para almacenar los resultados de la búsqueda.
A continuación, se obtiene el valor ingresado por el usuario mediante la instrucción request.POST.get("query", ""). Esto permite acceder al parámetro llamado "query" enviado a través del método POST. En caso de que el parámetro no exista, se devuelve un string vacío para evitar errores. Posteriormente, se aplica el método. lower () al texto ingresado con el objetivo de convertirlo a minúsculas. Esto garantiza que la búsqueda no sea sensible a diferencias entre mayúsculas y minúsculas, permitiendo que, por ejemplo, “Mario” y “mario”

sean interpretados de la misma manera.
¡Después, se verifica que el usuario haya ingresado efectivamente algún texto mediante la condición if query! = "". Esta validación es importante porque evita realizar un filtrado innecesario cuando el campo está vacío y permite controlar mejor el flujo del programa.
Si el usuario ingresó un valor, la vista delega la lógica de filtrado a la función filterByCharacter, ubicada en el archivo services.py. Esta decisión responde al principio de separación de responsabilidades. La vista se encarga de manejar la solicitud y devolver la respuesta, mientras que la lógica del negocio —en este caso, el filtrado de imágenes— se encuentra en la capa de servicios. Esta organización mejora la claridad del código, facilita su mantenimiento y permite reutilizar la lógica en otras partes del proyecto si fuera necesario.
Finalmente, la función utiliza render para devolver la respuesta al usuario. Se renderiza la plantilla home.html y se le envía como contexto la variable images, que contiene las imágenes filtradas. El template es el encargado de recorrer esa lista y mostrarlas en pantalla.
En resumen, esta vista actúa como un puente entre el formulario de búsqueda y la lógica del sistema: recibe la petición, obtiene el dato ingresado, valida la información, delega el filtrado al servicio correspondiente y devuelve los resultados para que sean visualizados en la interfaz.

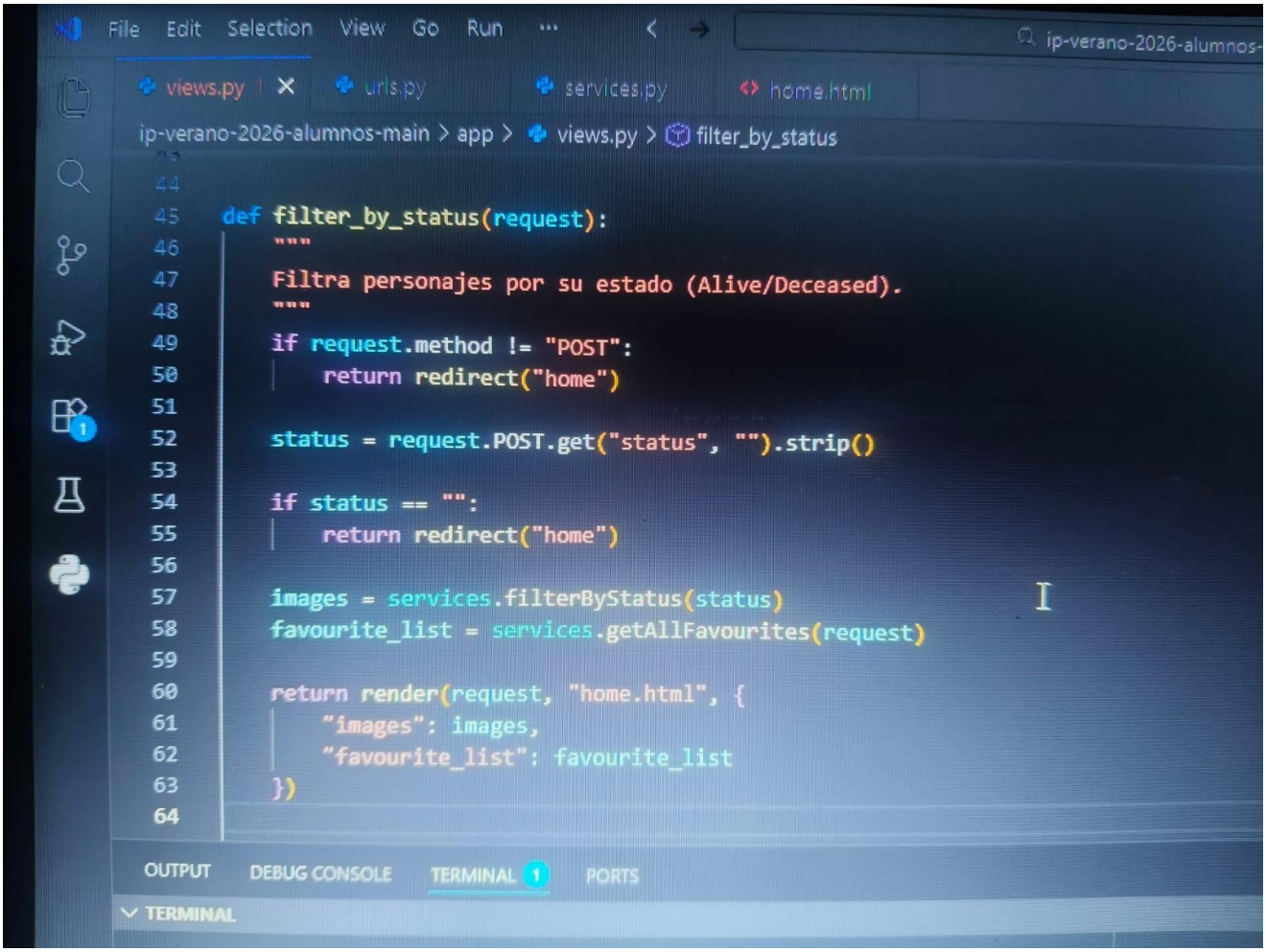
- **Dificultades Encontradas**

Durante el desarrollo del buscador surgieron errores que impedían que el código funcionara correctamente. En un primer momento, el filtrado fue implementado directamente dentro de la vista search, pero esta decisión generaba fallos que resultaban difíciles de detectar. A pesar de que la lógica parecía correcta, la forma en que estaba organizada dentro de la vista provocaba que la aplicación no respondiera como se esperaba.

Luego de revisar la estructura del proyecto, se identificó que ya existía la función filterByCharacter en services.py, diseñada específicamente para realizar el filtrado. Al reutilizar esa función y delegar allí la lógica correspondiente, el código comenzó a funcionar correctamente. Este cambio no solo solucionó los errores, sino que también permitió comprender en la práctica la importancia de respetar la separación de responsabilidades y mantener una organización coherente del proyecto.

Este proceso representó un aprendizaje significativo, ya que evidenció que una buena estructura del código no solo mejora la claridad, sino que también impacta directamente en su correcto funcionamiento.

❖ Buscador de Los Simpsons 2



Explicación de la función filter_by_status(request)
La función filter_by_status permite filtrar los personajes según su estado (Alive o Deceased).
Primero, la función recibe el objeto request, que contiene la información enviada desde el formulario cuando el usuario presiona uno de los botones de filtrado.
Se verifica que la solicitud sea de tipo POST. Si no lo es, el usuario es redirigido a la página principal. Esto asegura que la vista solo se utilice desde el formulario correspondiente.
Luego, se obtiene el valor del estado enviado desde el formulario mediante request.POST.get("status", ""). Si el valor está vacío, también se redirige al inicio para evitar aplicar un filtro sin criterio.
Si el estado tiene un valor válido, la vista llama a la función filterByStatus(status) ubicada en services.py. Esta función es la encargada de realizar el filtrado de las imágenes según el estado seleccionado.
Finalmente, se renderiza el template home.html enviando la lista de imágenes filtradas. También se envía la lista de favoritos para que la página pueda mostrarlos correctamente, aunque el filtrado no depende de ellos.
En resumen, esta función recibe el estado seleccionado por el usuario, llama al servicio que realiza el filtrado y devuelve los resultados a la página.

ANEXO 3

➢ Funcionalidad de Favorito

En el trabajo práctico se implementó la funcionalidad de guardar y eliminar personajes favoritos dentro de una aplicación web desarrollada en Django. Para esto se utilizó una arquitectura organizada por capas, separando la lógica en views, services y repositories, lo cual mejora el orden del código y facilita el mantenimiento.

1. Vista: Obtener todos los favoritos del usuario (getAllFavouritesByUser)

- En el archivo views.py se implementó la función:

```
@login_required
def getAllFavouritesByUser(request):
```

Esta función permite que el usuario autenticado pueda visualizar la lista de personajes que guardó como favoritos.

- Se utiliza el decorador @login_required para asegurar que **solo un usuario logueado** pueda acceder.
- Se llama al servicio:

```
favourite_list = services.getAllFavourites(request)
```

Esto devuelve la lista de favoritos del usuario.

Finalmente, se renderiza el template favourites.html, enviando la lista como contexto:

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

El objetivo es mostrar en pantalla todos los favoritos guardados por el usuario autenticado.

➢ Guardar un personaje como favorito

También en views.py se implementó:

```
@login_required

def saveFavourite(request):
```

Esta función permite guardar un personaje como favorito.

- Primero se valida que el método HTTP sea POST, ya que se está enviando información desde un formulario o botón:

```
if request.method == "POST":
```

- Luego,se lo llama al servivcio

```
services.saveFavourite(request)
```

- al finalizar se redirige al usuario nuevamente a la página principal

```
return redirect("home")
```

El objetivo es guardar un personaje en favorito y volver a home sin mostrar errores en pantalla.

➢ Eliminar un favorito de un usuario (deletefavourite)

Para permitir eliminarun personaje guardado se implementó:

```
@login_required
def deleteFavourite(request):
```

dentro de esta funcionse aplican validaciones importantes:

a) Validación del método POST

Primeros se verifica que la petición sea POST

```
if request.method != "POST":
    return redirect("home")
```

esto evita que se eliminen datos mediante una URL escrita manualmente

b) Obtención de ID del favorito

se obtiene el id del favorito desde el formulario:

```
fav_id = request.POST.get("id")
```

sino se recibe un id valido se redirige al home

```
if not fav_id:
    return redirect("home")
```

c) Validación de Seguridad

Se verifica que el favorito pertenezca realmente al usuario autenticado

```
if not Favourite.objects.filter(id=fav_id, user=request.user).exists():
    return redirect("home")
```

esto es fundamental porque evita que un usuario intente borrar favoritos de otra persona.

d) Eliminación mediante service

Si todo esta correcto se llama al servicio correspondiente

```
services.deleteFavourite(request)
```

y luego se re dirige nuevamente

```
return redirect("home")
```

En este caso el objetivo es eliminar un favorito garantizando que solo pueda hacerlo el dueño.

➢ Capa de Repository: Persistencia de datos (repositorie.py)

En el archivo repositories.py se encuentra la capa encargada de interactuar con la base de datos.

I. Guardar favorito en la base de datos:

Se implementó la función:
def saveFavourite(fav):

esta duncion utiliza:

```
Favourite.objects.get_or_create(...)
```

Esto significa que:

- Si el favorito ya existe, no lo vuelve a crear.
- Si no existe, lo guarda.

Se guardan atributos

- Name
- Gender
- Status
- Occupation
- Phrases
- Age
- image
- user

la ventaja es que se evita los duplicados y mantiene consistenciaen la base de datos

II. obtener todos los favoritos del usuario

se implementó:

```
def getAllFavourites(user):
```

esta función devuelve todos los favoritos del usuario usando:

```
Favourite.objects.filter(user=user).values(...)
```

Con values() se devuelve solo la información necesaria, como:

- id

- name
- gender
- status
- occupation
- phrases
- age
- image
- user_id

En este caso, el objetivo es listar solo los favoritos correspondientes al usuario autenticado.

- Evidencia den funconamiento (terminal)

En las imágenes del servidos se observael resgistro de peticiones HTTP, por ejemplo:

Con values() se devuelve solo la información necesaria, como:

- id
- name
- gender
- status
- occupation
- phrases
- age
- image
- user_id

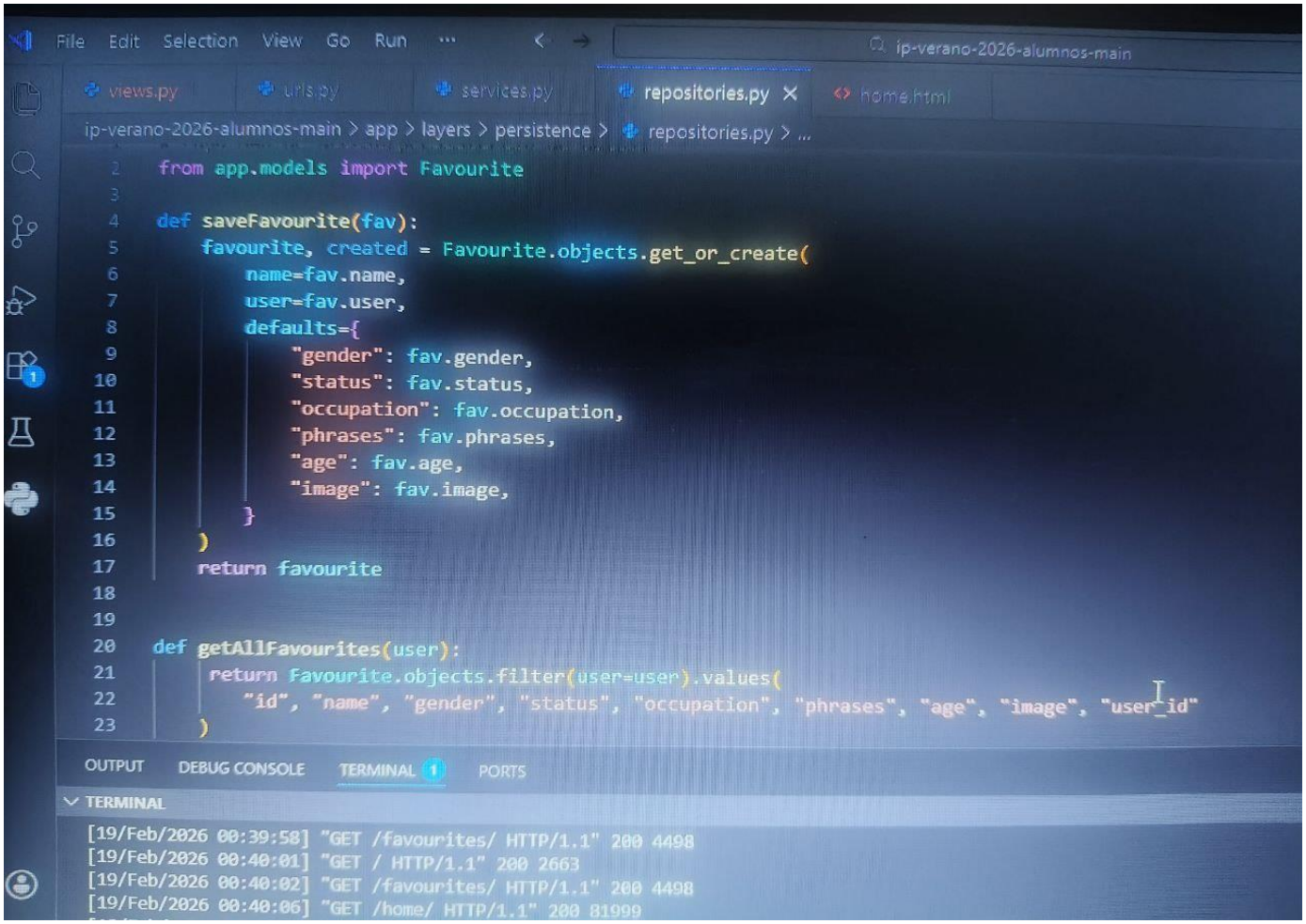
Finalmente, podemos decir que la funcionalidad de favoritos fue desarrollada utilizando Django siguiendo una estructura por capas:

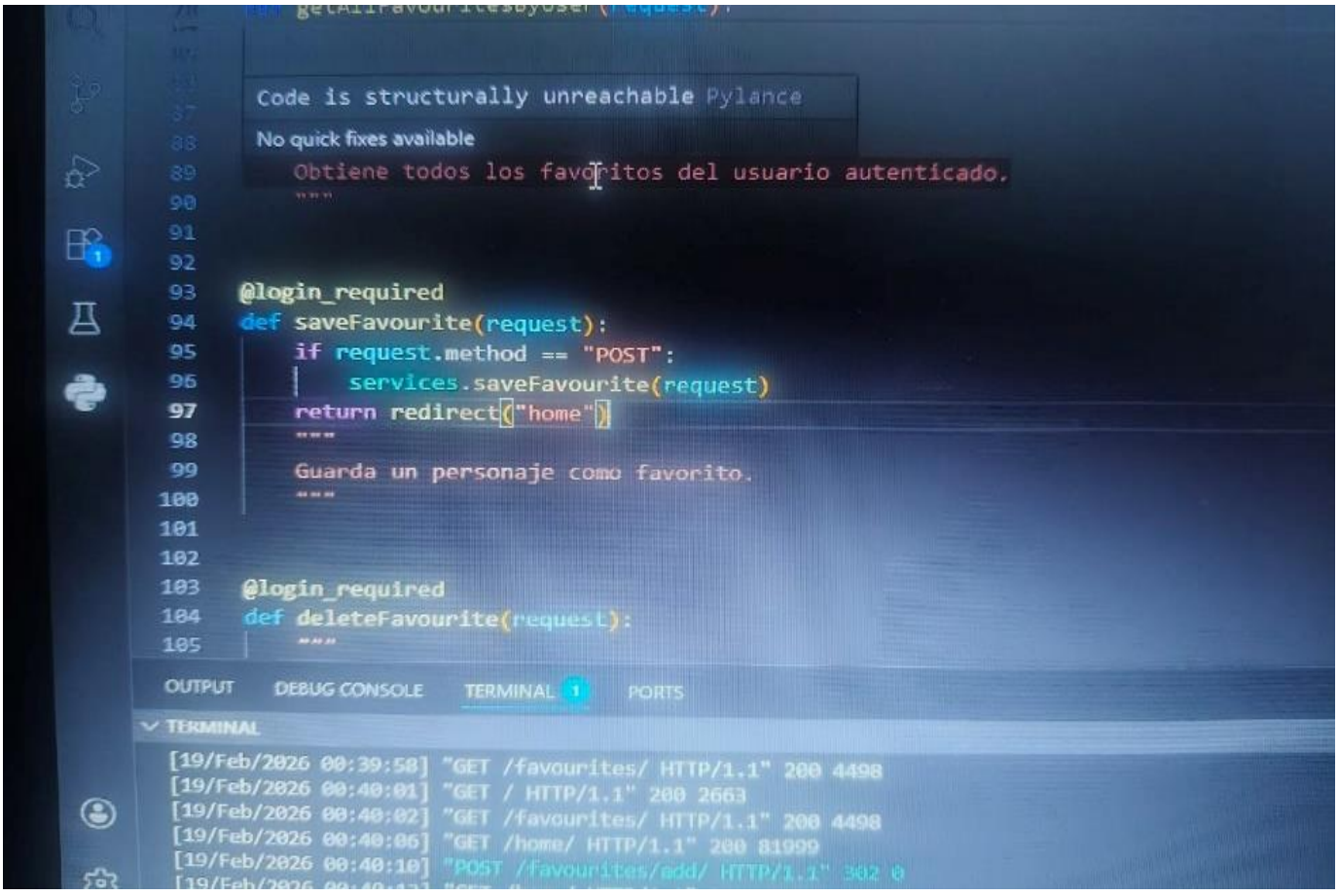
- **Views:** controlan la interacción del usuario y las rutas.
- **Services:** contienen la lógica de negocio.
- **Repositories:** se encargan del acceso a la base de datos.

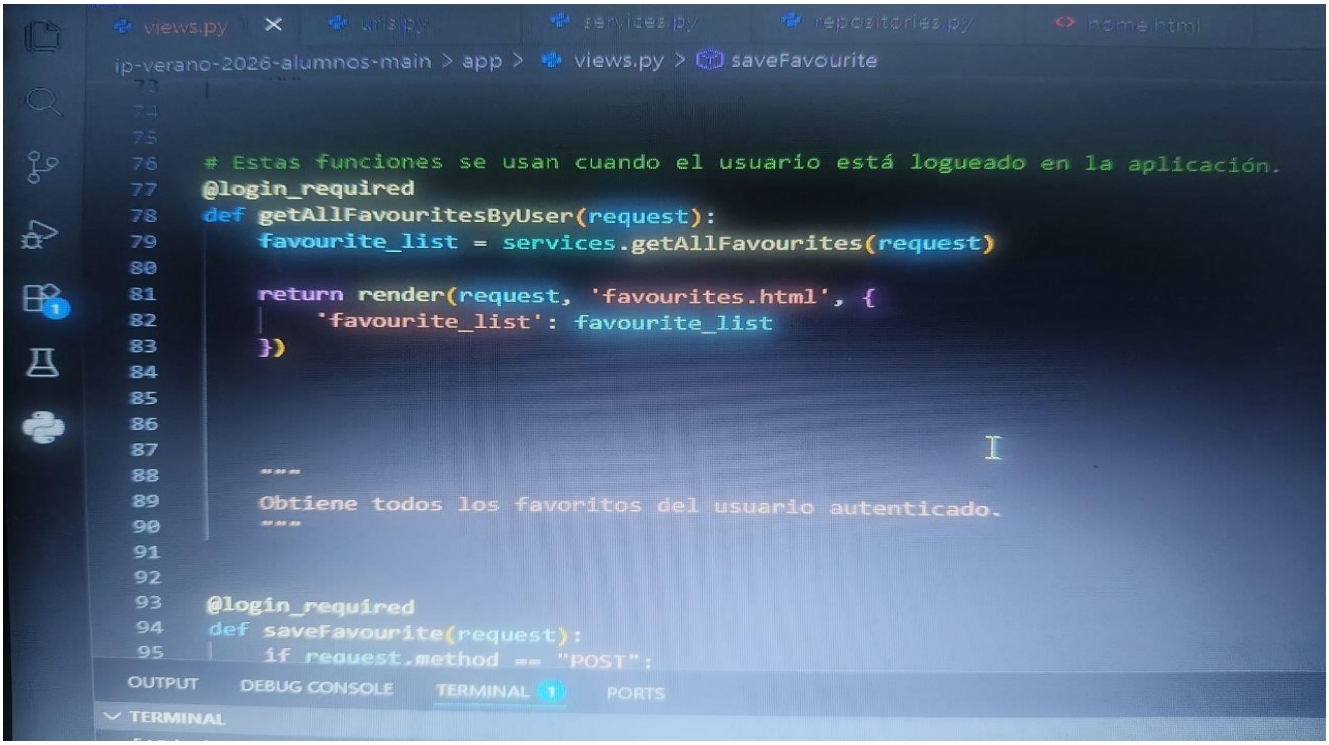
Además, se aplicaron validaciones de seguridad como:

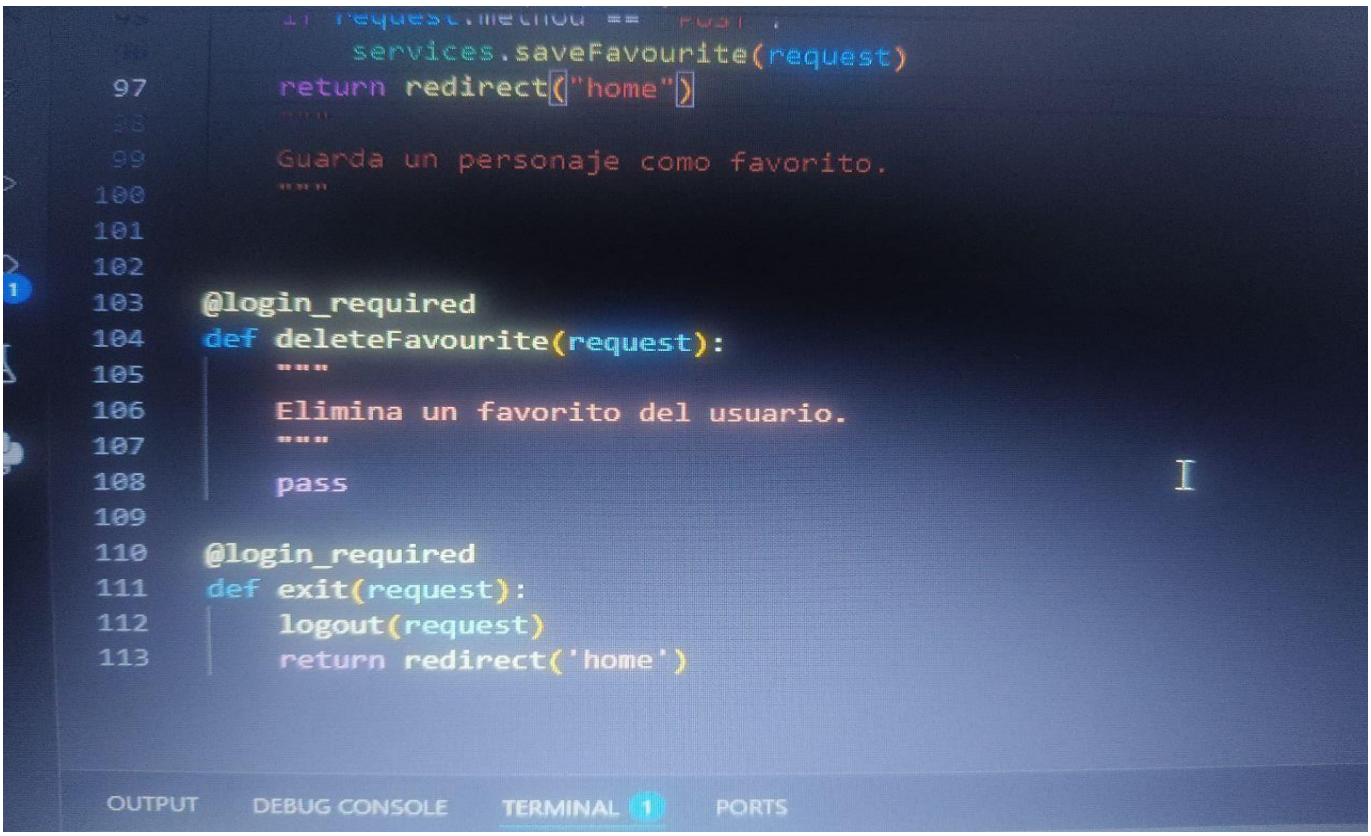
- uso de @login_required
- control del método POST
- verificación de pertenencia del favorito al usuario autenticado

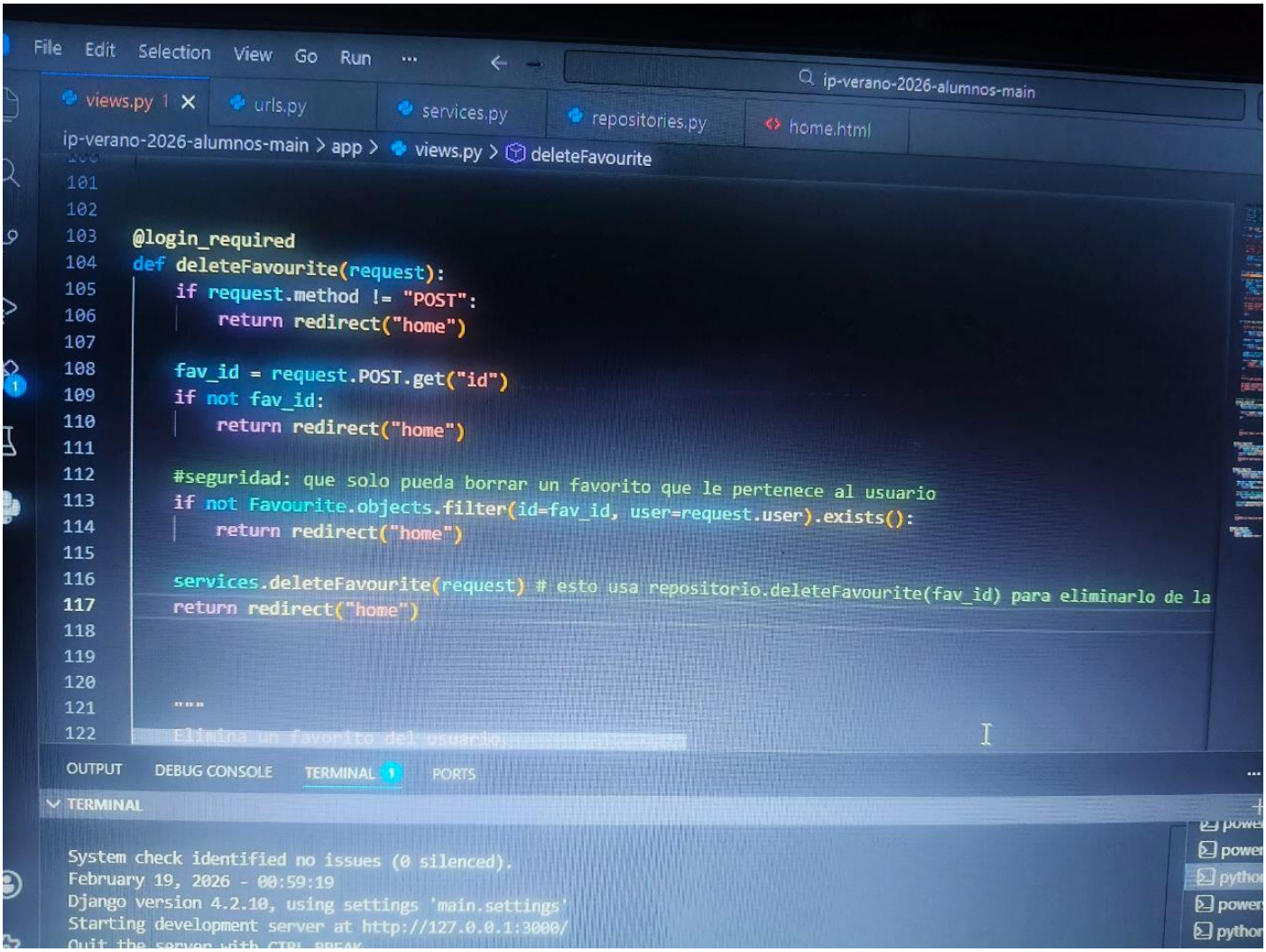
De esta forma, el sistema garantiza que cada usuario pueda **guardar, ver y eliminar únicamente sus propios favoritos**.


File Edit Selection View Go Run ···
ip-verano-2026-alumnos-main
views.py
urls.py
services.py
repositories.py
home.html
ip-verano-2026-alumnos-main > app > layers > persistence > repositories.py > ...
from app.models import Favourite
def saveFavourite(fav):
favourite, created = Favourite.objects.get_or_create(
name=fav.name,
user=fav.user,
defaults={
"gender": fav.gender,
"status": fav.status,
"occupation": fav.occupation,
"phrases": fav.phrases,
"age": fav.age,
"image": fav.image,
}
)
return favourite
def getAllFavourites(user):
return Favourite.objects.filter(user=user).values(
"id", "name", "gender", "status", "occupation", "phrases", "age", "image", "user_id"
)
OUTPUT DEBUG CONSOLE TERMINAL PORTS
TERMINAL
[19/Feb/2026 00:39:58] "GET /favourites/ HTTP/1.1" 200 4498
[19/Feb/2026 00:40:01] "GET / HTTP/1.1" 200 2663
[19/Feb/2026 00:40:02] "GET /favourites/ HTTP/1.1" 200 4498
[19/Feb/2026 00:40:06] "GET /home/ HTTP/1.1" 200 81999


Code is structurally unreachable Pylance
No quick fixes available
Obtiene todos los favoritos del usuario autenticado.
"""
@login_required
def saveFavourite(request):
if request.method == "POST":
services.saveFavourite(request)
return redirect("home")
"""
Guarda un personaje como favorito.
"""
@login_required
def deleteFavourite(request):
"""
OUTPUT DEBUG CONSOLE TERMINAL PORTS
TERMINAL
[19/Feb/2026 00:39:58] "GET /favourites/ HTTP/1.1" 200 4498
[19/Feb/2026 00:40:01] "GET / HTTP/1.1" 200 2663
[19/Feb/2026 00:40:02] "GET /favourites/ HTTP/1.1" 200 4498
[19/Feb/2026 00:40:06] "GET /home/ HTTP/1.1" 200 81999
[19/Feb/2026 00:40:10] "POST /favourites/add/ HTTP/1.1" 302 0


views.py
services.py
repositories.py
home.html
ip-verano-2026-alumnos-main > app > views.py > saveFavourite
Estas funciones se usan cuando el usuario está logueado en la aplicación.
@login_required
def getAllFavouritesByUser(request):
favourite_list = services.getAllFavourites(request)
return render(request, 'favourites.html', {
'favourite_list': favourite_list
})
"""
Obtiene todos los favoritos del usuario autenticado.
"""
@login_required
def saveFavourite(request):
if request.method == "POST":
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
TERMINAL


services.saveFavourite(request)
return redirect("home")
Guarda un personaje como favorito.
"""
@login_required
def deleteFavourite(request):
"""
Elimina un favorito del usuario.
"""
pass
@login_required
def exit(request):
logout(request)
return redirect('home')
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS


File Edit Selection View Go Run ...
ip-verano-2026-alumnos-main
views.py 1
urls.py
services.py
repositories.py
home.html
ip-verano-2026-alumnos-main > app > views.py > deleteFavourite
@login_required
def deleteFavourite(request):
if request.method != "POST":
return redirect("home")
fav_id = request.POST.get("id")
if not fav_id:
return redirect("home")
#seguridad: que solo pueda borrar un favorito que le pertenece al usuario
if not Favourite.objects.filter(id=fav_id, user=request.user).exists():
return redirect("home")
services.deleteFavourite(request) # esto usa repositorio.deleteFavourite(fav_id) para eliminarlo de la
return redirect("home")
"""
Elimina un favorito del usuario
OUTPUT
DEBUG CONSOLE
TERMINAL
PORTS
TERMINAL
System check identified no issues (0 silenced).
February 19, 2026 - 00:59:19
Django version 4.2.10, using settings 'main.settings'
Starting development server at http://127.0.0.1:3000/
Quit the server with CTRL-BREAK.