UNIVERSIDAD NACIONAL
DE GENERAL SARMIENTO

TRABAJO PRÁCTICO

Introducción a la Programación

DOCENTES

Montiel, Santiago

Godoy, Gonzalo

Rial, Jorgelina

Ruiz,Yair

GRUPO 3

Albarracin, Paula

Pilipiw, Andrea

En la actualidad, el desarrollo de aplicaciones web es fundamental para facilitar el acceso a información y mejorar la interacción del usuario con los datos. En este contexto, trabajaremos en un proyecto que consiste en crear una aplicación web utilizando Django, la cual permitirá buscar y visualizar imágenes de los personajes de la famosa serie *Rick & Morty*. Esta serie, que combina ciencia ficción y comedia, ofrece una rica variedad de personajes que los usuarios podrán explorar a través de la aplicación.

La aplicación se conectará a la API oficial de *Rick & Morty*, lo que nos permitirá acceder a información actualizada sobre cada personaje, incluyendo su imagen, estado, última ubicación y el episodio en el que debutó. Con una interfaz amigable, los usuarios pueden buscar personajes específicos y visualizar sus datos en tarjetas organizadas, mejorando así la experiencia de navegación.

Además, implementaremos un módulo de autenticación que permitirá a los usuarios registrarse y acceder a sus personajes favoritos. Esta funcionalidad no solo enriquecerá la aplicación, sino que también fomentará la personalización y el uso continuo por parte de los usuarios.

En el presente informe, se detallarán las funcionalidades que implementamos y las tareas pendientes que son cruciales para completar la aplicación, como la mejora en la presentación de los datos y la integración de la lógica de búsqueda.

Código de funciones implementadas

Implementados en views.py:

La función *home* obtiene dos listados, uno de imágenes y otro de favoritos. Ambas listas son generadas llamando a funciones definidas en *services.py*

```python
def home(request):
    images = services.getAllImages() # llamamos a la funcion obtener todas las imagenes
    favourite_list = services.getAllFavourites(request) # Llamamos a la funcion de obtener todos los favoritos

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

- [ ] Dificultad: Esta función requería obtener las imágenes desde la API y los favoritos del usuario autenticado, pero la lógica inicial no proporcionaba ninguna interacción con *services.py.* Además, las funciones en services.py estaban incompletas, lo que nos dificultó realizar pruebas en *Visual Studio Code.*

- ☐ Decisión: Integramos las múltiples funciones de services.py y modificamos el archivo de *views.py*, teniendo en cuenta las pistas de lo que hacía cada función en *services.py*.

La función *search* verifica que la búsqueda no esté vacía. Si el campo no está vacío, llama a las funciones correspondientes en *services.py* para obtener los resultados filtrados, que luego se renderizan en el template *home.html*, mostrando tanto las imágenes encontradas como los favoritos.

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

    if (search_msg != ''):
        images = services.getAllImages(search_msg) # llamamos a la funcion de obtener todas las imagenes usando e
        # llamamos a la lista de favoritos para que se muestren correctamente cuando utilizamos la barra de busqu
        favourite_list = services.getAllFavourites(request)
        return render(request, 'home.html', {'images': images, 'favourite_list': favourite_list})
    else:
        return redirect('home')
```

- ☐ Dificultad:En términos de dificultad, fue similar a la función *home*.

Si el usuario inició sesión, podrá acceder a guardar favoritos, verlos y eliminarlos. Y posteriormente salir de la sesión si así lo desea mediante las funciones *getAllFavouritesByUser, saveFavourite, deleteFavouritey exit.*

```python
@login_required
def getAllFavouritesByUser(request): # llamamos a la funcion de obtener todos los favoritos de un usuario para mostrarlos en otra plantilla
    favourite_list = services.getAllFavourites(request)
    return render(request, 'favourites.html', { 'favourite_list': favourite_list })

@login_required
def saveFavourite(request): # llamamos a la funcion agregar favoritos desde servicios
    services.saveFavourite(request)
    return redirect('home') # redireccionamos a la pagina 'home' para seguir viendo las imagenes

@login_required
def deleteFavourite(request): # llamamos a la funcion eliminar favoritos
    services.deleteFavourite(request)
    return redirect('favoritos') # redireccionamos a la pagina de favoritos para seguir viendo el listado de favs

@login_required
def exit(request): # llamamos a la funcion logout para desloguear
    logout(request)
    return redirect('login') # redireccionamos a la pagina de 'login' una vez deslogueado
```

- ☐ Dificultad: Inicialmente, no estaba claro cómo capturar los datos necesarios desde el *request* para guardar un favorito. Fue necesario analizar qué información era relevante (como el ID del personaje o los datos del usuario autenticado) para luego trabajar esas funciones que estaban incompletas en en *services.py.* Por otro lado, también tuvimos problemas a la hora de entender cómo y dónde redireccionar.
- ☐ Decisión: Al notar que en las pistas nos decía donde redireccionar, nos pudimos guiar con esa información.

Implementados en services.py:

getAllImages nos permitió obtener los datos de una API determinada, en este caso la de Rick & Morty. Llama a la función desde transport.py, recorre cada dato y los convierte en Card, devolviendo un listado de imágenes.

```
def getAllImages(input=None):

    json_collection = transport.getAllImages(input)


    images = []

    for object in json_collection:
        images.append(translator.fromRequestIntoCard(object))
    return images
```

- ☐ Dificultad: Al principio, la función no tenía ninguna implementación para interactuar con la API. La dificultad principal fue entender cómo hacer la solicitud a la API y qué tipo de datos esperar. La API de Rick & Morty devuelve una estructura JSON (también tuvimos que comprender que significaba) con múltiples niveles de información, y no estaba claro cómo extraer solo lo necesario (imagen, estado, ubicación, episodio).
- ☐ Decisión: Nos guiamos con las pistas y fuimos a revisar *transport.py*. Entendimos que es una capa que se comunica con la API, preparando los datos para ser utilizados. Además, nos encontramos que contenía la función *getAllImages* que la pista decía que debíamos usar. Por otro lado, la pista nos pedía que convirtamos los datos en una Card y agregarlo a images. Para eso, nos dimos cuenta de visar las importaciones, y notamos que se importó *translator.py.* Investigamos las funciones que contenía y la función *fromRequestIntoCard* coincidía con lo que la pista nos sugirió. Por el mismo motivo, decidimos llamarla y usar *images.append* para ir agregando a images.

La función *saveFavorite* es la encargada de guardar un elemento como favorito a un usuario determinado, guardándolo en la base de datos.

```
def saveFavourite(request):
    fav = translator.fromTemplateIntoCard(request) # transformamos un request del template en
    fav.user = get_user(request) # le asignamos el usuario correspondiente.

    return repositories.saveFavourite(fav) # lo guardamos en la base.
```

- ☐ Dificultad:No sabíamos bien cómo convertir los datos del template en un objeto Card y asignarle un usuario.

- ☐ Decisión: Ya habíamos revisado *translator.py*, por lo cual usamos la función que nos cumplía con lo que la pista solicitaba. Para determinar que asignarle a fav.user, nos ayudamos con la función siguiente que tenía la asignación de usuarios de forma correcta.

La función *getAllFavourites* verifica si el usuario inició sesión. Si no lo hizo, devuelve una lista vacía o procede a recuperar los favoritos del usuario, convertirlos en card y devolver el listado.

```python
def getAllFavourites(request):
    if not request.user.is_authenticated:
        return []
    else:
        user = get_user(request)

        favourite_list = repositories.getAllFavourites(user) # buscamos desde el repositories.py
        mapped_favourites = []

        for favourite in favourite_list:
            card = translator.fromRepositoryIntoCard(favourite) # transformamos cada favorito en
            mapped_favourites.append(card)

        return mapped_favourites
```

- ☐ Decisión:Para completar esta parte del código, investigamos el código que nos sugirieron *repositories.py*. Usamos la función *getAllFavourites* que cumplía con lo pedido.

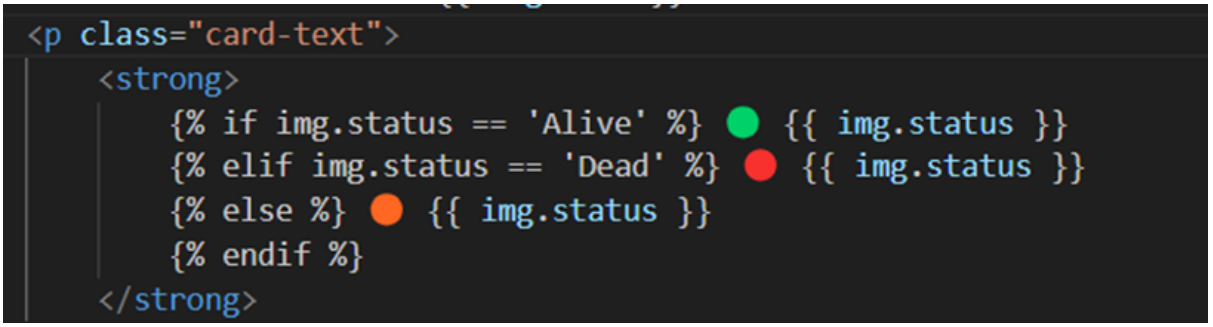
Implementados en HTML:

Esta clase fue implementada con Bootstrap para cambiar el color del borde de las cards según su estado para responder a la consigna pedida.

```
<div class="card mb-3 ms-5
    {% if img.status == 'Alive' %} border-success
    {% elif img.status == 'Dead' %} border-danger
    {% else %} border-warning
    {% endif %}"
    style="max-width: 540px;">
```

- ☐ Dificultad: Enlazar el borde de la card con el estado de los personajes.
- ☐ Decisión:Se decidió copiar el código ya implementado en *class=”card-text”* ya que evaluaba el estado de los personajes, y se agregó a través de bootstrap border.

Se evalúa el estado de img.status y genera un circulo de color dependiendo ese estado (vivo, muerto o desconocido).



- ☐ Dificultad: La condición para mostrar los íconos de estado en el texto usaba una comparación incorrecta (true == 'Alive'). Esto no reflejaba correctamente los estados de los personajes.
- ☐ Decisión: Corregimos la lógica en el template para comparar correctamente el valor de img.status con las cadenas 'Alive', 'Dead' y otros estados posibles. Esta modificación garantizó que los íconos correctos se mostrará junto al nombre del personaje, asegurando que los usuarios pudieran ver rápidamente su estado (vivo, muerto, o desconocido).

DESAFÍOS Y ERRORES GENERALES

1. Manejo de Nombres Establecidos (listas y variables)

Uno de los primeros desafíos fue lidiar con los nombres establecidos tanto para las listas como para las variables. A medida que comenzamos a trabajar con las funciones que ya estaban definidas en el proyecto, nos dimos cuenta de que algunos nombres de variables y listas eran claves para la correcta ejecución del sistema. Sin embargo, al principio cometimos el error de modificar estos nombres sin entender completamente cómo afectaban al flujo general de la aplicación. Esto nos causó confusión y varios errores, ya que modificábamos las variables en las vistas o servicios sin considerar las interacciones entre ellas.

Este problema se resolvió cuando decidimos centrarnos en entender cómo las funciones y variables ya existentes interactuaban y cómo podíamos utilizarlas adecuadamente sin crear conflictos en el código.

2. Confusión al Modificar Funciones sin Usar las Funciones Ya Dadas

Al principio, tratamos de crear nuevas funciones y modificar las existentes sin aprovechar las que ya estaban implementadas en `views.py` y `services.py`. Esto generaba muchos errores, ya que no sabíamos si debíamos modificar ambas al mismo tiempo o si una debía depender de la otra. Esta falta de claridad sobre cómo

debía estructurarse la lógica nos causó frustración, ya que no estábamos seguros de cuál era la forma correcta de dividir el trabajo entre las vistas y los servicios.

Por ejemplo, cuando intentábamos guardar los favoritos, no sabíamos si deberíamos hacerlo directamente en la vista o si debíamos pasar esos datos a los servicios y luego llamarlos desde las vistas. Esto generó algunos errores de lógica y problemas de comunicación entre las diferentes partes del código.

3. Dificultades con GitHub

Uno de los mayores desafíos que enfrentamos fue la integración de GitHub con Visual Studio Code para poder trabajar de manera colaborativa. Tuvimos varios problemas al intentar enlazar nuestros repositorios, lo que nos llevó a perder mucho tiempo. No pudimos sincronizar correctamente nuestros cambios, lo que provocó que no pudiéramos compartir de manera eficiente el código. Intentamos seguir tutoriales y guías, pero no logramos solucionar los errores de configuración en *GitHub* y *Visual Studio Code*.

Esto generó varios inconvenientes: tuvimos que enviar nuestros avances manualmente, lo que hacía el proceso más lento y, en ocasiones, cometíamos errores al tratar de compartir los archivos. También hubo momentos en los que, al no estar sincronizadas las versiones del código, perdimos algunos cambios importantes.