TRABAJO PRÁCTICO – INTRO A LA PROGRAMACIÓN

INTEGRANTES:

- Poblet Tobias
- Villalba Nehuén
- Segovia Sebastian

El trabajo trata de hacer funcionar una página web, completando funciones ya dadas y añadiendo otras para mejorar la página.

En la clase del trabajo práctico, en un principio estuvimos un poco perdidos acerca de cómo podíamos empezar a hacer lo obligatorio, completar las primeras tres funciones, views.py, services.py y home.html, pero le pedimos ayuda al profe Lucas, nos dio una mano y pudimos completarlas sin mucho problema.

BUSCADOR:

SERVICES.PY:

```
def getAllImages(input=None):

    # obtiene un listado de datos "crudos" desde la API, usando a transport.py.

    if input:

        json_collection = transport.getAllImagesFiltered(input)  #Si hay input, usa
la funcion que obtiene TODOS los personajes

    else:

        json_collection = transport.getAllImagesHome(config.DEFAULT_REST_API_URL) #Si
no hay input, solo devuelve los primeros 20 que da la API



    # recorre cada dato crudo de la colección anterior, lo convierte en una Card y lo
agrega a images.

    images = []

    for object in json_collection:

        card = translator.fromRequestIntoCard(object)

        images.append(card)



    return images
```

Acá hicimos que la función extraiga todas las imágenes en crudo desde la API de Rick & Morty y las transformara en Cards con las funciones de *translator*.

Esta función la hicimos con ayuda del profesor Lucas en clase, gracias a esto pudimos seguir nosotros solos con las siguientes funciones.

INICIO DE SESIÓN:

VIEWS.PY:

```
@login_required
def exit(request):
    logout(request)
    return redirect('index-page')
```

La parte de completar el inicio de sesión fue bastante sencillo gracias a que la mayor parte de la *feature* estaba ya implementada en el código, solo tuvimos que completar el deslogueo de la cuenta en caso de que el usuario esté logueado.

AÑADIR A FAVORITOS:

VIEWS.PY:

```
# 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 })


@login_required
def saveFavourite(request):
    if request.method == 'POST':
        services.saveFavourite(request)
        redirect_url = request.POST.get('redirect_url', '/')
        if redirect_url:
            return redirect(redirect_url)
    return redirect('home')



@login_required
def deleteFavourite(request):
    if request.method == 'POST':
        services.deleteFavourite(request)
    return redirect('favoritos')
```

En el archivo **views.py** definimos que las funciones de agregar y borrar los favoritos requieran de que el usuario esté logueado.

SERVICES.PY:

```
    # recorre cada dato crudo de la colección anterior, lo convierte en una Card y lo
agrega a images.
    images = []
    for object in json_collection:
        card = translator.fromRequestIntoCard(object)
        images.append(card)

    return images


# añadir favoritos (usado desde el template 'home.html')
def saveFavourite(request):
    fav = translator.fromTemplateIntoCard(request) # transformamos un request del
template en una Card.
    fav.user = request.user # le asignamos el usuario correspondiente.

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


# usados desde el template 'favourites.html'
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 TODOS los favoritos del usuario (variable 'user').
        mapped_favourites = []

        for favourite in favourite_list:
            card = translator.fromRepositoryIntoCard(favourite) # transformamos cada
favorito en una Card, y lo almacenamos en card.
            mapped_favourites.append(card)

        return mapped_favourites


def deleteFavourite(request):
    favId = request.POST.get('id')
    return repositories.deleteFavourite(favId) # borramos un favorito por su ID.
```

En *services.py* creamos todas las funciones que se usaron en *views.py*, la función para guardar favoritos es *savefavourite* donde transforma el request del template en una Card.

Después la función de ver todos los favoritos comprueba que el usuario esté logueado y busca sus favoritos desde el *repositories.py*. Por último la función de deleteFavourite borra un favorito por su ID.

PAGINACIÓN DE RESULTADOS:

HOME.HTML:

```
<main>

    <h1 class="text-center">Buscador Rick & Morty</h1>


    <div class="d-flex justify-content-end" style="margin-bottom: 1%; margin-right:
2rem;">

        <nav aria-label="Paginación">

            <ul class="pagination">

                <!-- Página anterior -->

                {% if images.has_previous %}

                    <li class="page-item">

                        <a class="page-link" href="?page={{
images.previous_page_number }}&query={{ search_msg }}">⬅️</a>

                    </li>

                {% else %}

                    <li class="page-item disabled">

                        <a class="page-link">⬅️</a>

                    </li>

                {% endif %}


                <!-- Números de página -->

                {% for page_num in paginator.page_range %}

                    <li class="page-item {% if page_num == images.number %}active{%
endif %}">

                        <a class="page-link" href="?page={{ page_num }}&query={{
search_msg }}">{{ page_num }}</a>

                    </li>

                {% endfor %}


                <!-- Página siguiente -->

                {% if images.has_next %}

                    <li class="page-item">

                        <a class="page-link" href="?page={{ images.next_page_number
}}&query={{ search_msg }}">➡️</a>

                    </li>

                {% else %}
```

```
                    <li class="page-item disabled">
                        <a class="page-link">➡️</a>
                    </li>
                {% endif %}
            </ul>
        </nav>
    </div>
```

Con la ayuda de ChatGPT y la documentación de Django pudimos hacer la paginación de los resultados, haciendo que si tiene más resultados para seguir te deja la opción de presionar el botón, llevándote a los demás resultados; Y en caso de no tener más resultados, deshabilitando la opción de presionar el botón. Lo mismo con ir para atrás.

LOADING SPINNER PARA LA CARGA DE IMÁGENES:

HOME.HTML:

```
<!-- Círculo de carga -->
<div id="loading-circle" class="loading-overlay" style="display: none;">
    <div class="spinner"></div>
</div>


<!-- Script -->
<script>
    document.addEventListener('DOMContentLoaded', function () {
        const images = document.querySelectorAll('img'); // Selecciona todas las imágenes
        const loadingCircle = document.getElementById('loading-circle'); // El círculo de carga

        // Mostrar el círculo inmediatamente al cargar la página
        loadingCircle.style.display = 'flex';

        let totalImages = images.length;
        let loadedImages = 0;

        // Función para verificar si todas las imágenes están cargadas
        const checkIfAllLoaded = () => {
            loadedImages++;
            if (loadedImages === totalImages) {
                loadingCircle.style.display = 'none'; // Ocultar el círculo de carga
            }
```

```
        };

        // Iterar por cada imagen
        images.forEach((img) => {
            // Verifica si la imagen ya está completamente cargada (por caché o
precarga)
            if (img.complete) {
                checkIfAllLoaded(); // Considerar la imagen como cargada
            } else {
                // Escuchar eventos onload y onerror para imágenes no cargadas
                img.onload = checkIfAllLoaded;
                img.onerror = checkIfAllLoaded;
            }
        });

        // Si no hay imágenes en la página, ocultar el círculo de carga
        if (totalImages === 0) {
            loadingCircle.style.display = 'none';
        }
    });
</script>
```

El círculo de carga lo hicimos todo con ChatGPT, que nos dio un código de javascript, que hace aparecer el spinning antes de que cargue la imágen, además nos dio un código para poner en *styles.css* el cuál le da el color, la forma y la animación al spinner, el código es el siguiente:

```
.spinner {
    width: 50px;
    height: 50px;
    border: 5px solid #f3f3f3;
    border-top: 5px solid #3498db; /* Color del spinner */
    border-radius: 50%;
    animation: spin 1s linear infinite;
}

@keyframes spin {
    0% {
        transform: rotate(0deg);
    }
    100% {
        transform: rotate(360deg);
```

```
    }
}
```

RENOVACIÓN DE INTERFAZ GRÁFICA:

Usando *CSS* con los conocimientos que adquirimos en un curso pasado de Front End, pudimos darle un lavado de cara a la página con un fondo nuevo y animaciones, más otros detalles más.

No tuvimos problemas en esta instancia, fue bastante sencillo.

PROBLEMAS:

Tuvimos varios problemas al querer completar el trabajo, la mayoría los pudimos resolver volviendo sobre nuestros pasos y con ayuda de ChatGPT. Intentamos hacer todos los opcionales pero no conseguimos lograr el *Alta para nuevos usuarios* y los *Comentarios de Favoritos* para los usuarios logueados, para la primera vimos el video que venia con el *github* de ayuda en el *Readme.md* pero no logramos conseguir que funcione, y en la casilla de Comentarios no logramos hacer que lo que subía el usuario se guarde en la nube.

El único ‘bug’ que no pudimos resolver fue un bug visual relacionado con la búsqueda de personajes y el estado de “Agregar a Favoritos” y “Ya está agregado a favoritos”, donde en la página principal sí te aparece si un personaje está en tu lista de favoritos pero en los resultados de búsqueda aparece como si no estuviera agregado ese mismo personaje, cuando en realidad sí lo está.