



INTRODUCCIÓN A LA PROGRAMACIÓN

COMISIÓN 04

PROFESORES: OMAR ARGAÑARAS; NANCY NORES

INTERGRANTES: BRUNO CAVICCHIOLI, ROMINA CHIALVO
OLINIK, KARLA NICOLE LÓPEZ CONTRERAS.

FECHA DE ENTREGA: 27 DE NOVIEMBRE, 2024.

ÍNDICE

INTRODUCCIÓN	3
VISUALIZACIÓN DE IMÁGENES EN LA GALERÍA.....	4
BUSCADOR	5
LOGIN Y LOGOUT	6
BOTÓN Y LISTA DE FAVORITOS	7
COMENTARIOS EN LOS FAVORITOS.....	10
CAMBIOS EN LA PRESENTACIÓN VISUAL DE LA PÁGINA.....	15
CONCLUSIÓN	20

INTRODUCCIÓN

El siguiente escrito tiene como finalidad explicar paso a paso la realización del Trabajo Práctico de la materia “Introducción a la Programación”. Dicho trabajo consistió en la implementación de una aplicación web que permite buscar a los personajes de la serie “Rick & Morty”, usando su API homónima, para ver una imagen del miembro del cast consultado, su estado (vivo, muerto, desconocido), su primera aparición y la última ubicación donde se sabe que estuvo.

Lo anterior implicaba tomar el código otorgado por los profesores y terminar o editar las líneas y bloques de código necesarios para poder, como mínimo, visualizar las imágenes en la página web.

Para el desarrollo de este Trabajo Práctico fueron utilizados Visual Studio Code, para editar los archivos que conforman la página web; y Copilot y Chat GPT, para consultar posibles dudas que las lecturas recomendadas para la clase y los tutoriales no pudieran esclarecer.

VISUALIZACIÓN DE IMÁGENES EN LA GALERÍA.

Para lograr que se muestren las imágenes en el apartado de “galería”, se necesitó editar las funciones en el archivo views.py, el cual, estaba codificado de la siguiente manera:

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

Como las listas que se encargan de mostrar las imágenes y los favoritos estaban vacías, no se visualizaban dichas imágenes; el código fue modificado como se muestra a continuación:

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

De esta forma, se llama a las funciones definidas en la capa de servicio para que se muestren las imágenes en la galería. Además, también necesitamos cambiar la capa de servicio, ya que no traía los datos desde la API que luego debían procesarse en services.py y mostrarse en views.py. El código original era:

```
from ..persistence import repositories  
from ..utilities import translator  
from django.contrib.auth import get_user
```

```
def getAllImages(input=None):  
    json_collection = []  
    images = []  
    return images
```

A este código le hacía falta la función que se encarga de transportar los datos crudos de la API, convertirlos en cards y agregarlos a images; para luego llamar a esta función en views.py y que puedan visualizarse las imágenes de cada personaje. Así quedó con nuestras modificaciones:

```
from ..persistence import repositories  
from ..utilities import translator
```

```

from ..transport import transport
from django.contrib.auth import get_user

def getAllImages(input=None):
    json_collection = transport.getAllImages(input)
    images = []
    for datoCrudo in json_collection:
        cardConversion = translator.fromRequestIntoCard(datoCrudo)
        images.append(cardConversion)
    return images

```

Hicimos las correspondientes importaciones para que se pueda llamar a la función desde transport.py que se encarga de traer los datos crudos desde la API. Luego, recorreremos esa lista de datos crudos y los vamos convirtiendo con la función de translator.py que convierte la información de la API en una card. Finalmente, agregamos dichas cards a una lista llamada images, que es la misma que se verá en views.py cuando se llame a esta función presente en services.py.

BUSCADOR

Para poder desarrollar la función del buscador nos centramos, nuevamente, en los archivos views.py y services.py. Volviendo a la porción de código que exploramos anteriormente, podemos observar que en el código original la lista de datos crudos traídos desde la API está vacía.

Luego de llamar a la función correspondiente y de traer dichos datos desde transport.py, agregamos un input. Esto quiere decir que si el input es vacío (es decir, “Input=None”) traerá todas las imágenes sin ningún filtro. Pero si este input tiene una palabra presente en el nombre de algún personaje, esto filtrará los datos traídos desde la API con la función getAllImages de transport.py:

```

def getAllImages(input=None):
    json_collection = transport.getAllImages(input)
    images = []
    for datoCrudo in json_collection:
        cardConversion = translator.fromRequestIntoCard(datoCrudo)
        images.append(cardConversion)
    return images

```

Esto, a su vez, está acompañado de un código con una función similar en views.py:

```
def search(request):
    search_msg = request.POST.get('query', '')
    if (search_msg != ''):
        images = services.getAllImages(search_msg)
        favourite_list = services.getAllFavourites(request)
        return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
    else:
        return redirect('home')
```

Esta función procesa la búsqueda del usuario y nos dice que, si la búsqueda no es vacía, entonces la utilice como filtro en la función getAllImages de services.py, es decir, que las imágenes traídas de la API correspondan con ese término de búsqueda.

LOGIN Y LOGOUT

En cuanto al inicio de sesión, las capas involucradas son dos: views.py y login.html. La capa login.html ya nos vino configurada con anterioridad, por lo tanto no le hicimos ninguna modificación. Lo que sí hicimos fue agregar una función en views.py que mostrara el funcionamiento del inicio de sesión. En el código original no estaba presente esta función, por lo que su implementación entera fue hecha por nosotros.

```
from django.http import HttpResponseRedirect
def user_login(request):
    if request.method == 'POST':
        username = request.POST.get('username')
        password = request.POST.get('password')
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home') # Redirige a la página principal u otra página
        else:
            return HttpResponseRedirect('Nombre de usuario o contraseña incorrectos')
    return render(request, 'login.html')
```

Lo primero que hicimos fue importar la clase “HttpResponse” para enviarle una respuesta al usuario desde nuestro servidor Django. Esta función llamada “user_login”

convierte los requests del usuario en su nombre de usuario y contraseña y luego autentica que, efectivamente, estos correspondan a la base de datos. Si la autenticación es correcta, es decir, pertenecen a la base de datos, se redirecciona al usuario a home. Caso contrario, se le envía, con la clase “HttpResponse”, una respuesta incorrecta al usuario y se le redirecciona nuevamente al apartado del inicio de sesión.

Además, configuramos el logout del sitio, que originalmente era así:

```
@login_required
```

```
def exit(request):
```

```
    pass
```

Lo cambiamos para que, al salir del sitio, redirija al usuario a la página de inicio:

```
@login_required
```

```
def exit(request):
```

```
    logout(request)
```

```
    return redirect('index-page')
```

BOTÓN Y LISTA DE FAVORITOS

Para trabajar con la lógica de los favoritos, hicimos cambios en las capas views.py y services.py. Comenzando con esta última, originalmente estaba configurada únicamente la lógica de borrar un favorito, pero no la de añadir un personaje a favorito ni la de visualizar todos los favoritos del usuario:

```
def saveFavourite(request):
```

```
    fav = "
```

```
    fav.user = "
```

```
    return repositories.saveFavourite(fav)
```

```
def getAllFavourites(request):
```

```
    if not request.user.is_authenticated:
```

```
        return []
```

```
    else:
```

```
        user = get_user(request)
```

```
        favourite_list = []
```

```
        mapped_favourites = []
```

```
        for favourite in favourite_list:
```

```
            card = "
```

```

        mapped_favourites.append(card)
    return mapped_favourites
def deleteFavourite(request):
    favId = request.POST.get('id')
    return repositories.deleteFavourite(favId)

```

En primer lugar, llamamos a las funciones correspondientes para poder añadir un personaje a favoritos:

```

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

```

Esta función, llamando a “fromTemplateIntoCard” de translator.py, convierte el request que viene del template en una card y lo almacena en una variable llamada “fav”. Luego, a ese favorito le asigna un usuario, que es aquel que guardó el favorito. Finalmente retorna una función que llama desde repositories.py, que se encarga de guardar ese favorito elegido por el usuario como un favorito “propriadamente dicho”, es decir, con todos los datos que se tienen (nombre, estatus, primera aparición, última aparición, etc.) y se lo guarda en la base de datos.

```

def getAllFavourites(request):
    if not request.user.is_authenticated:
        return []
    else:
        user = get_user(request)
        favourite_list = repositories.getAllFavourites(user)
        mapped_favourites = []
        for favourite in favourite_list:
            card = translator.fromRepositoryIntoCard(favourite)
            mapped_favourites.append(card)
        return mapped_favourites

```

Esta función se encarga de traer a todos los favoritos que el usuario guardó, por lo que es importante que el usuario esté autenticado. De no estarlo, no devolverá nada. Si el usuario está efectivamente autenticado, se guarda, llamando a la función “getAllFavourites” de repositories.py, a todos los favoritos del usuario desde la base de

datos en una lista. Luego, se recorre dicha lista y se transforma a cada elemento de la lista en una card, ya que antes era un elemento propio de la base de datos (que se guardó al momento de guardarlo como favorito). Luego de hacer la conversión, se lo añade a una lista, ahora sí como card, que se mostrará en el template.

Con todo esto configurado, aún no podíamos ver los favoritos ni agregarlos o borrarlos. La capa `views.py`, encargada de la presentación o vista, era así originalmente:

```
@login_required
def getAllFavouritesByUser(request):
    favourite_list = []
    return render(request, 'favourites.html', { 'favourite_list': favourite_list })

@login_required
def saveFavourite(request):
    pass

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

Era importante tener configurado el inicio de sesión primero para poder ejecutar todas estas funciones relacionadas con los favoritos, ya que era necesario que el usuario esté con la sesión iniciada. Aun así, estas funciones por sí solas no hacen nada, ya que la lista de favoritos está vacía y no se pueden guardar ni eliminar favoritos. Era necesario cambiar la lógica en ambas capas, la de servicio y la de vista, para que todo funcione correctamente.

Así quedaron los cambios que implementamos:

```
@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)
    return redirect('home')
```

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

Para que los favoritos del usuario se muestren correctamente, llamamos a la función “getAllFavourites” desde services.py que, como ya dijimos, se encarga de traerlos desde la base de datos y transformarlos en una card que luego pueda ser mostrada en la vista.

Luego, para guardar un favorito también llamamos a la función homónima desde services.py que se encarga de transformar el request del template en un dato que se guarda en la base de datos, mientras que, para eliminar el favorito, llamamos a la función homónima desde services.py que transforma el request del template en un id con el cual, la función de repositories.py se encarga de eliminar el favorito seleccionado.

COMENTARIOS EN LOS FAVORITOS

Para añadir comentarios en las imágenes que son marcadas como favoritos modificamos todo lo relacionado con la base de datos y su conexión con los servicios y capa de vista (es decir, la capa translator.py) y no tanto el resto de las capas que modificamos cuando añadimos otras funcionalidades. Lo primero que modificamos fue la capa que manipula la base de datos (repositories.py). Así se veía originalmente:

```
def saveFavourite(image):
    try:
        fav = Favourite.objects.create(url=image.url, name=image.name,
        status=image.status, last_location=image.last_location, first_seen=image.first_seen,
        user=image.user)
        return fav
    except Exception as e:
        print(f"Error al guardar el favorito: {e}")
        return None

def getAllFavourites(user):
    favouriteList = Favourite.objects.filter(user=user).values('id', 'url', 'name', 'status',
    'last_location', 'first_seen')
    return list(favouriteList)
```

Lo único que hicimos fue añadir “message” como una variable más que se tiene en cuenta a la hora de guardar un favorito. Además, tuvimos que hacer migraciones para aplicar estos cambios a la base de datos. Así nos quedó el código cambiado:

```
def saveFavourite(image):
    try:
        fav = Favourite.objects.create(url=image.url, name=image.name,
        status=image.status, last_location=image.last_location, first_seen=image.first_seen,
        user=image.user, message=image.message)

        return fav
    except Exception as e:
        print(f"Error al guardar el favorito: {e}")
        return None

def getAllFavourites(user):
    favouriteList = Favourite.objects.filter(user=user).values('id', 'url', 'name', 'status',
    'last_location', 'first_seen', 'message')
    return list(favouriteList)
```

De esta manera, el mensaje también entra como una variable a tener en cuenta en un favorito. También tuvimos que modificar la capa translator.py, ya que hay que procesar el mensaje que se ingresa en el template y guardarlo en una base de datos, o también traerlo desde la base de datos y mostrarlo en el template. Así era la capa translator.py originalmente:

```
def fromTemplateIntoCard(templ):
    card = Card(
        url=templ.POST.get("url"),
        name=templ.POST.get("name"),
        status=templ.POST.get("status"),
        last_location=templ.POST.get("last_location"),
        first_seen=templ.POST.get("first_seen")
    )
    return card

def fromRepositoryIntoCard(repo_dict):
    card = Card(
        id=repo_dict['id'],
```

```

        url=repo_dict['url'],
        name=repo_dict['name'],
        status=repo_dict['status'],
        last_location=repo_dict['last_location'],
        first_seen=repo_dict['first_seen'],
    )
    return card

```

Y así nos quedó una vez modificada:

```

def fromTemplateIntoCard(templ):
    card = Card(
        url=templ.POST.get("url"),
        name=templ.POST.get("name"),
        status=templ.POST.get("status"),
        last_location=templ.POST.get("last_location"),
        first_seen=templ.POST.get("first_seen"),
        message=templ.POST.get('message')
    )
    return card

```

```

def fromRepositoryIntoCard(repo_dict):
    card = Card(
        id=repo_dict['id'],

        url=repo_dict['url'],
        name=repo_dict['name'],
        status=repo_dict['status'],
        last_location=repo_dict['last_location'],
        first_seen=repo_dict['first_seen'],
        message=repo_dict['message']
    )
    return card

```

De esta manera, cada vez que se convierta un request del template en una card o un dato de la base de datos en una card, se tendrá en cuenta el mensaje que ingresó el usuario. Cabe aclarar que esto no está presente en la función que convierte datos de la API en card porque no vienen mensajes dados desde la API, sino que son algo específico de nuestra aplicación.

Además, debe modificarse el modelo de cada card de favoritos. El modelo original es el siguiente:

```
class Favourite(models.Model):
    url = models.TextField()
    name = models.CharField(max_length=200)
    status = models.TextField()
    last_location = models.TextField()
    first_seen = models.TextField()

    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=
models.CASCADE)
```

```
class Meta:
```

```
    unique_together = (('user', 'url', 'name', 'status', 'last_location', 'first_seen'),)
```

Agregamos “message” para que a la hora de convertir un request o un dato de la base de datos en card no haya errores:

```
class Favourite(models.Model):
    url = models.TextField()
    name = models.CharField(max_length=200)
    status = models.TextField()
    last_location = models.TextField()
    first_seen = models.TextField()
    message = models.TextField(default="")

    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=
models.CASCADE)
```

```
class Meta:
```

```
unique_together = (('user', 'url', 'name', 'status', 'last_location', 'first_seen',
'message'))
```

Finalmente, agregamos en home.html los códigos necesarios para que el usuario pueda añadir el mensaje y para que lo visualice en el apartado de favoritos.

```
{% 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="url" value="{{ img.url }}">
            <input type="hidden" name="status" value="{{ img.status }}">
            <input type="hidden" name="last_location" value="{{
img.last_location }}">
            <input type="hidden" name="first_seen" value="{{ img.first_seen
}}">
            <textarea name="message" placeholder="Escribe tu
mensaje..."></textarea>
            {% if img in favourite_list %}
                <button type="submit" class="btn btn-primary btn-sm float-left"
style="color:white" disabled>✔ Ya está en favoritos</button> {% else %}
                <button type="submit" class="btn btn-primary btn-sm float-left"
style="color:white">❤ Añadir a favoritos</button> {% endif %}
            </form>
        </div>
    {% endif %}
```

En favourites.html:

```
<table class="table table-striped table-hover table-bordered">
    <thead>
        <tr>
            <th>#</th>
            <th>Imagen</th>
            <th>Nombre <i class="fa fa-sort"></i></th>
            <th>Status</th>
            <th>Última ubicación <i class="fa fa-sort"></i></th>
            <th>Episodio inicial </th>
```

```

        <th>Mensaje </th>
        <th>Acciones </th>
    </tr>
</thead>
<tbody>
    {% for favourite in favourite_list %}
    <tr>
        <td>-</td>
        <td></td>
        <td>{{ favourite.name }}</td>
        <td>{{ favourite.status }}</td>
        <td>{{ favourite.last_location }}</td>
        <td>{{ favourite.first_seen }}</td>
        <td>{{ favourite.message }}</td>
        <td>

```

CAMBIOS EN LA PRESENTACIÓN VISUAL DE LA PÁGINA




Para la parte del código encargada de los círculos y bordes de colores que indican el estado actual del personaje, se tenía como base el siguiente bloque de código en el archivo home.html:

```

<div class="row row-cols-1 row-cols-md-3 g-4">
    {% if images|length == 0 %}
    <h2 class="text-center">La búsqueda no arrojó resultados...</h2>
    {% else %} {% for img in images %}
    <div class="col">
        <div class="card mb-3 ms-5" style="max-width: 540px;">
            <div class="row g-0">
                <div class="col-md-4">
                    
                </div>
                <div class="col-md-8">
                    <div class="card-body">
                        <h3 class="card-title">{{ img.name }}</h3>

```

```

<p class="card-text">
    <strong>
        {% if true == 'Alive' %}  {{ img.status }}
        {% elif true == 'Dead' %}  {{ img.status }}
        {% else %}  {{ img.status }}
        {% endif %}
    </strong>

```

Al analizarlo detenidamente se notó que la línea “`{% if true == 'Alive' %}`” estaba comparando un booleano con una string, por lo tanto, esa sección no arrojaba el resultado esperado al no poder cumplirse los if, elif y else; siendo que el booleano *True* nunca tomaría otro valor/significado.

Por otra parte, en el bloque anterior no existía alguna línea de código que generara el borde de colores deseado para los cards; para la realización de ello se dio lectura a la documentación de Bootstrap para cards anexada a la información del Trabajo Práctico, así como también del artículo “How To Use If/Else Conditions In Django Templates”, siendo ambos textos sugeridos por los profesores de la materia. Tras la lectura y comparando lo escrito por default en el código original de la página web con lo escrito por el equipo, se llegó a lo siguiente:

```

{% if img.status == 'Alive' %} border border-3 border-success
{% elif img.status == 'Dead' %} border border-3 border-danger
{% else %} border border-3 border-warning

```

Se identificó la etiqueta `<div>` correspondiente a los cards en el archivo `home.html` para poder ingresar la nueva porción de código para los bordes, se corrigió el uso del `True` por la variable correspondiente a la información sobre el estado del personaje obtenido de la API y el archivo json, y se obtuvo la siguiente versión del código:




```

<div class="card mb-3 ms-5"
    {% if img.status == 'Alive' %} border border-3 border-success
    {% elif img.status == 'Dead' %} border border-3 border-danger
    {% else %} border border-3 border-warning {% endif %} style="max-width:
540px;">
    <div class="row g-0">
        <div class="col-md-4">

```



```

        
    </div>
    <div class="col-md-8">
        <div class="card-body">
            <h3 class="card-title">{{ img.name }}</h3>
            <p class="card-text">
                <strong>
                    {% if img.status == 'Alive' %}  {{ img.status }}
                    {% elif img.status == 'Dead' %}  {{ img.status }}
                    {% else %}  {{ img.status }}
                    {% endif %}
                </strong>
            </p>
        </div>
    </div>

```

Para editar la presentación visual/layout de la página web se nos proporcionaron archivos en html y un archivo css. Se tenían conocimientos básicos previos que permitieron editar los archivos html, donde se añadieron id's a varias etiquetas para poder editar partes específicas de la página web con mayor precisión en el archivo css, como mover el footer, editar el color del container de los favoritos, editar la opacidad de la barra de navegación, agregar una imagen en la esquina inferior derecha de la pantalla, entre otros.

Al código del archivo styles.css se le agregaron (o editaron) los siguientes elementos, mismos que pueden verse reflejados en los cambios en todos los archivos html:

```

#banner {
    background: rgb(205, 247, 248);
    opacity: 65%;
    text-shadow: 2px 0 #c1c8eb;
}
h1 {
    color: rgb(205, 247, 248);
    text-shadow: 2px 0 #000000;
}
h2 {
    color: rgb(205, 247, 248);
    text-shadow: 2px 0 #000000;
}

```

```

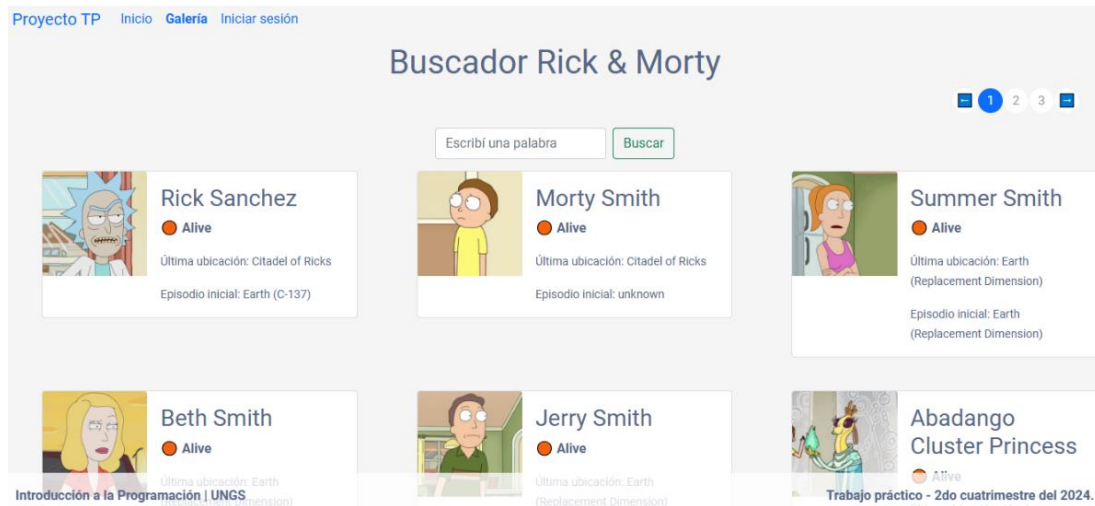
}
#mnsj-debajo {
    font-weight: bold;
    color: rgb(205, 247, 248);
    text-shadow: 2px 0 #000000;
}
#mnsj-debajo a:link {
    font-weight: bold;
    text-shadow: 2px 0 #ffffff;
}
#mnsj-debajo a:hover {
    text-shadow: 2px 0 #ffffff;
    color: rgb(91, 54, 177);
}
#borde-container-favos {
background-color: #1c1d2b;
}
#barra-footer {
    clear: both;
    position: fixed;
    bottom: 0px;
    width: 100%;
    z-index: 0;
}
#bailongo {
    float: right;
    position: relative;
    top: -40px;
    left: -50px;
    z-index: 1;
}

```

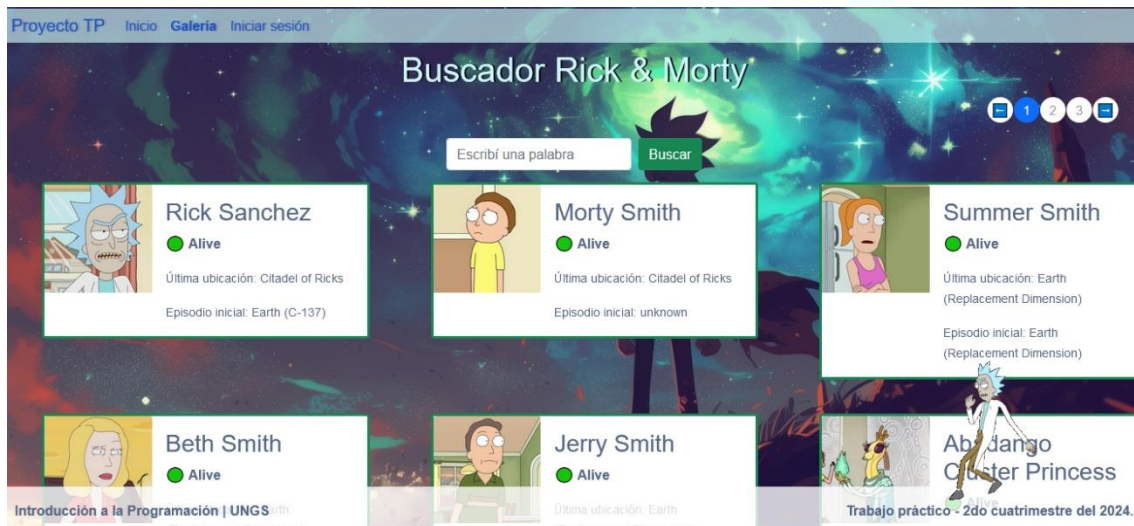
Donde “banner” hace referencia al header, h1 y h2 representan el texto utilizado en varios lugares de la página, “mnsj-debajo” es el mensaje que aparece debajo de la bienvenida en la página principal, “border-container-favos” es para editar el borde del

contenedor donde aparece la lista de favoritos cuando se inicia sesión y se agregan personajes a la misma, “barra-footer” es la barra del footer con la información de la materia que se movió para poder superponer la imagen agregada a la misma altura, y “bailongo” siendo el nombre clave para el gif de Rick Sánchez bailando en la esquina inferior derecha de la pantalla, apilada encima del footer. Además, se agregó un favicon y una imagen de fondo.

Con todos estos cambios pasamos de esto:



A esta versión del buscador:



El equipo intentó hacer funcionar la paginación, se observó el video recomendado y se integraron líneas del mismo al código del trabajo, se utilizó el Paginador de Django pero al final no fue posible implementar la función, incluso con todos los cambios y adiciones hechas los botones de paginación continuaron sin andar.

CONCLUSIÓN

Al dialogar sobre lo afrontado y lo aprendido durante la realización del Trabajo Práctico, el equipo llegó a las siguientes conclusiones. Primero, el consenso general sobre la dificultad de la tarea fue que realmente no era un desafío que fuera imposible, pero que al principio sí resultó abrumador; sobre todo por pasar de codificar programas pequeños en PyScripter para tareas sencillas, a usar aplicaciones que no habíamos utilizado antes, y más aún, tener que aprender a usarlas casi sobre la marcha.

Otra opinión compartida es que el trabajo podría haber sido realizado con mayor fluidez de haber tenido contacto previo con programas como Visual Studio Code o Git Bash, porque entre los intentos por comprender los softwares utilizados y los días en que se rindieron parciales de otras materias, la realidad es que se perdió mucho tiempo que sentimos pudo haber sido de provecho en el desarrollo de funciones adicionales para el buscador de personajes.

Como se mencionó antes, se intentó arreglar la paginación, pero lamentablemente no tuvimos tiempo suficiente para terminar de entenderlo e implementarlo, y los videos sugeridos no fueron suficientes para apoyarnos a entender cómo hacerlo. También, se tuvo la intención de agregar un spinner a la página, que tampoco pudo terminarse por tiempo y por tener muchas dudas a pesar de los tutoriales vistos.

Como puntos positivos, estamos de acuerdo en que un Trabajo Práctico tiene que presentarse como un reto que los estudiantes deben estar no sólo listos, sino también dispuestos a superar. En ese sentido, el equipo considera haber logrado superarlo, si bien con la sensación de que pudimos haber hecho más. Fue interesante poder ver cómo el llamar funciones trabajaba en tiempo real, ya que en clase sólo se vieron ejemplos hipotéticos con programas que no podían corroborarse si funcionaban o no; además, entrar en contacto con código html y manipularlo nos da una idea de cómo funciona y de cómo se relaciona este, directamente, al lenguaje css.

En términos generales, se considera haber tenido un buen desempeño, y se ve al Trabajo Práctico como un ejercicio muy completo que nos ayudó a terminar de poner en práctica muchos de los ejemplos realizados en clase; aunque nos hubiera agradado tener más acompañamiento y no depender tanto de las consultas realizadas a las IA's y a tutoriales en internet.