



# TRABAJO FINAL IP

Buscador Rick & Morty

## Fecha de Entrega

02-12-2024

## Docentes

Bressky, Daniel

Fassio, Esteban

Velazquez Espinola, Santiago

## Alumnas

Maidana Ulpo, Mara Carolina

Navas, Joana Nerina

Ozore, Nahara Ariana

## Introducción

El trabajo consiste en implementar una aplicación web usando Django que permita buscar imágenes de los personajes de la serie Rick & Morty, usando su API homónima. La información que provenga de esta API será renderizada por el framework en distintas cards que mostrarán -como mínimo- la imagen del personaje, el estado, la última ubicación y el episodio inicial. Adicionalmente -y para enriquecerla- se prevee que los estudiantes desarrollen la lógica necesaria para hacer funcionar el buscador central y un módulo de autenticación básica (usuario/contraseña) para almacenar uno o más resultados como favoritos, que luego podrán ser consultados por el usuario al loguearse. En este último, la app deberá tener la lógica suficiente para verificar cuándo una imagen fue marcada en favoritos.

## Condiciones de entrega

Requisitos de aprobación y criterios de corrección

1. El TP debe realizarse en grupos de 2 o 3 integrantes (no 1). Para aprobar el trabajo se deberán reunir los siguientes ítems:
  - La galería de imágenes se muestra adecuadamente (imagen, nombre, estado, última ubicación y episodio inicial de cada personaje).
  - El código debe ser claro. Las variables y funciones deben tener nombres que hagan fácil de entender el código a quien lo lea -de ser necesario, incluir comentarios que clarifiquen-. Reutilizar el código mediante funciones todas las veces que se amerite.
  - No deben haber variables que no se usan, funciones que tomen parámetros que no necesitan, ciclos innecesarios, etc.
  - El 'correcto' funcionamiento del código NO es suficiente para la aprobación del TP, son necesarios todos los ítems anteriores.

❖ Al iniciar la aplicación y hacer clic sobre Galería, verás lo siguiente:



## Lo que falta hacer (OBLIGATORIO)

### (1) views.py:

home(request): obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los usa para dibujar el correspondiente template. Si el opcional de favoritos no está desarrollado, devuelve un listado vacío.

```
# esta función obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los usa para dibujar el correspondiente template.
# si el opcional de favoritos no está desarrollado, devuelve un listado vacío.
def home(request):
    images = []
    favourite_list = []

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

## (2) services.py:

getAllImages(input=None): obtiene un listado, con formato de Card, de imágenes de la API. El parámetro input, si está presente, indica sobre qué imágenes/personajes debe filtrar/traer.

```
def getAllImages(input=None):
    # obtiene un listado de datos "crudos" desde La API, usando a transport.py.
    json_collection = []

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

    return images
```

## (3) home.html:

La tarjeta debe cambiar su border color dependiendo del estado del personaje. Si está vivo (alive), mostrará un borde verde; si está muerto (dead) mostrará rojo y si es desconocido (unknown), será naranja. Se sugiere consultar la documentación de Bootstrap sobre Cards cómo generar condicionales en Django para tener un mejor acercamiento a la solución.

```
home.html X
app > templates > home.html > main > div.row.row-cols-1.row-cols-md-3.g-4 > div.col > div.card.border-success.m
1 {% extends 'header.html' %} {% block content %}
2 <main>
3 <h1 class="text-center">Buscador Rick & Morty</h1>
4
5 <div class="d-flex justify-content-end" style="margin-bottom: 1%; margin-right: 2rem;">
6 <!-- Selector de página -->
7 <nav aria-label="...">
8 <ul class="pagination">
9 <li class="page-item disabled">
10 <a class="page-link">1</a>
11 </li>
12 <li class="page-item active" aria-current="page">
13 <a class="page-link" href="#">1</a>
14 </li>
15 <li class="page-item">
16 <a class="page-link" href="#">2</a>
17 </li>
18 <li class="page-item"><a class="page-link" href="#">3</a></li>
19 <li class="page-item">
20 <a class="page-link" href="#">4</a>
21 </li>
22 </ul>
23 </nav>
24 </div>
25
```

## ADICIONALES

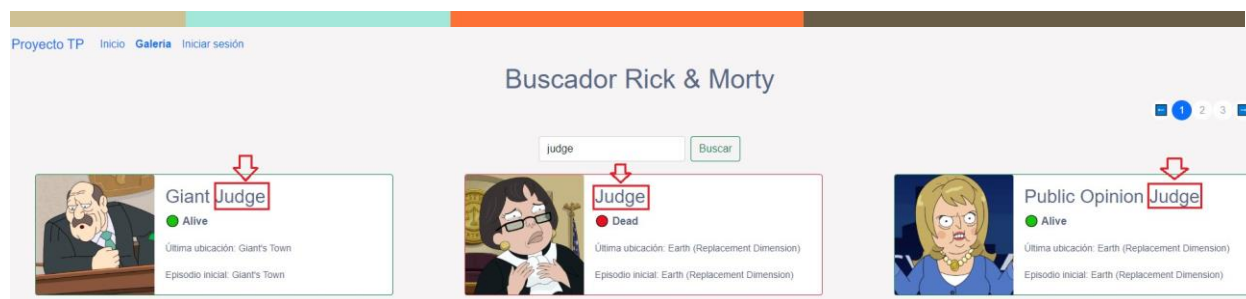
### Buscador ☆

Se debe completar la funcionalidad para que el buscador filtre adecuadamente las imágenes, según los siguientes criterios:

Si el usuario NO ingresa dato alguno y hace clic sobre el botón 'Buscar', debe mostrar las mismas imágenes que si hubiese hecho clic sobre el enlace Galería.

Si el usuario ingresa algún dato (ej. Samantha), al hacer clic en 'Buscar' se deben desplegar las imágenes filtradas relacionadas a dicho valor.

Ejemplo para judge (juez). ATENCIÓN, las imágenes pueden variar:



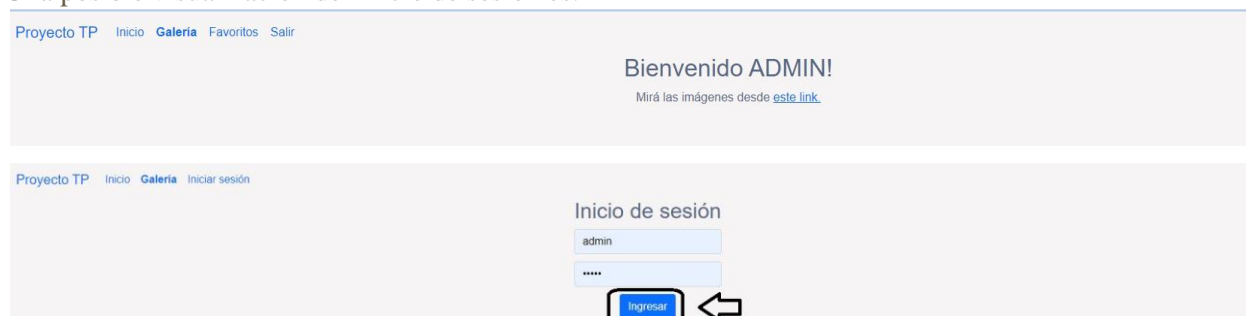
## Inicio de sesión ☆ ☆

Se debe completar la feature de inicio de sesión de la app. El usuario y contraseña a utilizar, preliminarmente, es admin/admin (ya se encuentra guardado sobre la base SQLite, tabla auth\_user).

Consideraciones:

- NO se permite utilizar Django Admin para emular la autenticación de los usuarios, la sección Iniciar sesión debe funcionar adecuadamente.
- Solo los usuarios que hayan iniciado sesión podrán añadir las imágenes como favoritos y visualizarlas en su sección correspondiente.

Una posible visualización del inicio de sesión es:



## Favoritos ☆ ☆:

Se debe completar la lógica presente para permitir que un usuario logueado pueda almacenar una o varias imágenes de la galería como favoritos, mediante el clic de un botón en la parte inferior.

Observaciones

Este punto puede realizarse SOLO si el ítem anterior (inicio de sesión) está desarrollado/funcionando bien.

Si el favorito ya fue añadido, debe mostrarse un botón que impida reañadirlo.

Debe existir una sección llamada 'Favoritos' que permita listar todos los agregados por el usuario, mediante una tabla. Además, debe existir un botón que permita removerlo del listado (si fue removido, desde la galería de imágenes podrá ser agregado otra vez).


Parte del código ya está resuelto. Revisar los archivos views.py, repositories.py y services.py.

Una posible visualización de este ítem resuelto es:


Proyecto TP Inicio **Galería** Favoritos Salir

## Buscador Rick & Morty


Escribí una palabra



**Rick Sanchez**  
● Alive  
Última ubicación: Citadel of Ricks  
Episodio inicial: Earth (C-137)  
[Añadir a favoritos](#)





**Morty Smith**  
● Alive  
Última ubicación: Citadel of Ricks  
Episodio inicial: unknown  
[Añadir a favoritos](#)



**Summer Smith**  
● Alive  
Última ubicación: Earth (Replacement Dimension)  
Episodio inicial: Earth (Replacement Dimension)  
[Añadir a favoritos](#)

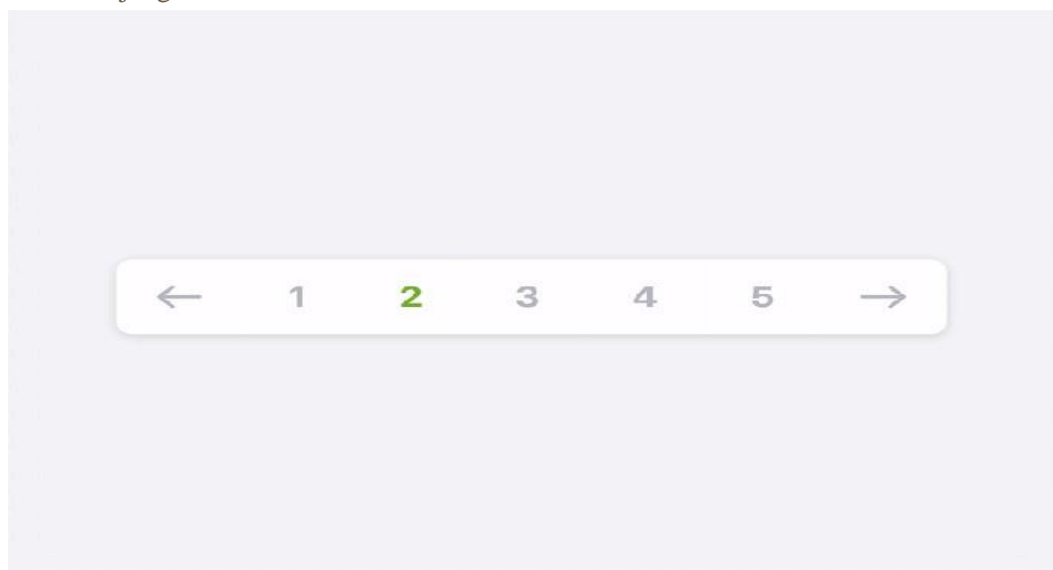
Listado de **FAVORITOS**

#	Imagen	Nombre	Status	Última ubicación	Episodio inicial	Acciones
-		Beth Smith	Alive	Earth (Replacement Dimension)	Earth (Replacement Dimension)	

## Paginación de resultados:

Por default, la API de Rick & Morty limita la cantidad de personajes a 20 por pagina (es decir, a lo sumo, se visualizarán 20 imágenes en el buscador ya que siempre se solicita la primer página). La idea de este punto es desarrollar un paginador de los resultados de búsqueda, de forma tal que:

- Liste la totalidad de páginas presentes. Ejemplo: si busco "Rick", y hay 10 páginas de 20 personajes cada una, entonces el usuario debe poder seleccionar entre las 10 enumeradas.
- Cada vez que se selecciona una página, se debe recargar la galería de imágenes con los resultados pertinentes a los personajes de dicha página.
- Tip: Tener en cuenta la documentación de la API y este ejemplo de paginación simple con Django.





## Renovar interfaz gráfica:

Se debe proponer una nueva interfaz gráfica para los distintos templates de la aplicación.

Recomendaciones:

- Pueden usar el framework CSS que deseen, sea Bootstrap, Tailwind, Foundation, etc., siempre y cuando consideren que el código debe resultar LEGIBLE para su corrección.
- Verificar que la lógica implementada en los templates funcione bien a medida que se modifica la interfaz.

### INTRODUCCIÓN:

En el trabajo se busca desarrollar una página web utilizando la API de Rick & Morty con un código ya comenzado donde nosotros los alumnos completamos las funciones para conseguir los puntos anteriormente mencionados. Para esto, tuvimos que informarnos con los links dejados como ayuda en el repositorio de GitHub del tp y con nuestra propia búsqueda en internet mediante diferentes fuentes. Teniendo en cuenta los contenidos de la cursada y la documentación externa, comenzamos a comprender el código de a poco, para luego realizar las consignas:

### (1) **views.py:** Lo que falta completar.

Para comenzar a armar la página, partimos de un código ya armado con diferentes archivos de python, html y css. En este punto se nos pedía completar en el apartado .views la función home:

```
# esta función obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los usa para dibujar el correspondiente template.
# si el opcional de favoritos no está desarrollado, devuelve un listado vacío.
def home(request):
    images = []
    favourite_list = []

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

En la función vemos que images y favourite\_list son dos listas vacías, que deberían obtener las imágenes de la API. Entonces, buscamos de dónde deberíamos obtener dicha información.

En la parte superior de .views podemos ver que se “trae”(importa) services desde layers, entonces vamos a revisar esa parte del código.

```
views.py > ...
# capa de vista/presentación

from django.shortcuts import redirect, render
from .layers.services import services
from django.contrib.auth.decorators import login_required
from django.contrib.auth import logout
```

### (2) **services.py:** Dentro de services, podemos ver que getAllImages (obtener todas las imágenes en español) convierte la información de la API en las cards que necesitamos. dentro de la misma función nos dicen que utiliza a transport.py para obtener los datos de la API, por lo que nos dirigimos ahí para ver cómo funciona y cómo implementarlo en Services.

```
# capa de servicio/lógica de negocio

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

def getAllImages(input=None):
    # obtiene un listado de datos "crudos" desde la API, usando a transport.py.
    json_collection = []

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

    return images
```

En Transport.py tenemos lo siguiente:

```
def getAllImages(input=None):
    if input is None:
        json_response = requests.get(config.DEFAULT_REST_API_URL).json()
    else:
        json_response = requests.get(config.DEFAULT_REST_API_SEARCH + input).json()

    json_collection = []
```

getAllImages en Transport.py tiene el json con las imagenes de la API, por lo que importamos transport a services.

```
from ..transport import transport

def getAllImages(input=None):
    # obtiene un listado de datos "crudos" desde la API, usando a transport.py.
    json_collection = []
    json_collection = transport.getAllImages(input)

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

    return images
```

para json\_collection llamamos a getAllImages desde transport para tener las imagenes, y a esos datos los recorremos en un for para agregar(.append) a la lista de imagenes, las mismas en formas de card con fromRequestIntoCard, la cual proviene de translator.py que tiene diversas funciones las cuales obtienen datos de la API y los transforman:

```

pp > layers > utilities > translator.py > fromTemplateIntoCard
3 from app.layers.utilities.card import Card
4
5 # usado cuando la info. viene de la API, para transformarla en una Card.
6 def fromRequestIntoCard(object):
7     card = Card(
8         url=object['image'],
9         name=object['name'],
10        status=object['status'],
11        last_location = object['location']['name'],
12        first_seen = object['origin']['name']
13    )
14    return card
15
16 # usado cuando la info. viene del template, para transformarlo en una Card antes de guardarlo en la base de datos.
17 def fromTemplateIntoCard(templ):
18     card = Card(
19         url=templ.POST.get("url"),
20         name=templ.POST.get("name"),
21         status=templ.POST.get("status"),
22         last_location=templ.POST.get("last_location"),
23         first_seen=templ.POST.get("first_seen")
24     )
25    return card
26
27 # cuando la info. viene de la base de datos, para transformarlo en una Card antes de mostrarlo.
28 def fromRepositoryIntoCard(repo_dict):
29     card = Card(
30         id=repo_dict['id'],
31         url=repo_dict['url'],
32         name=repo_dict['name'],
33         status=repo_dict['status'],
34         last_location=repo_dict['last_location'],
35         first_seen=repo_dict['first_seen'],
36     )
37    return card
38

```

Una vez completa la función getAllImages, volvemos a views.py:

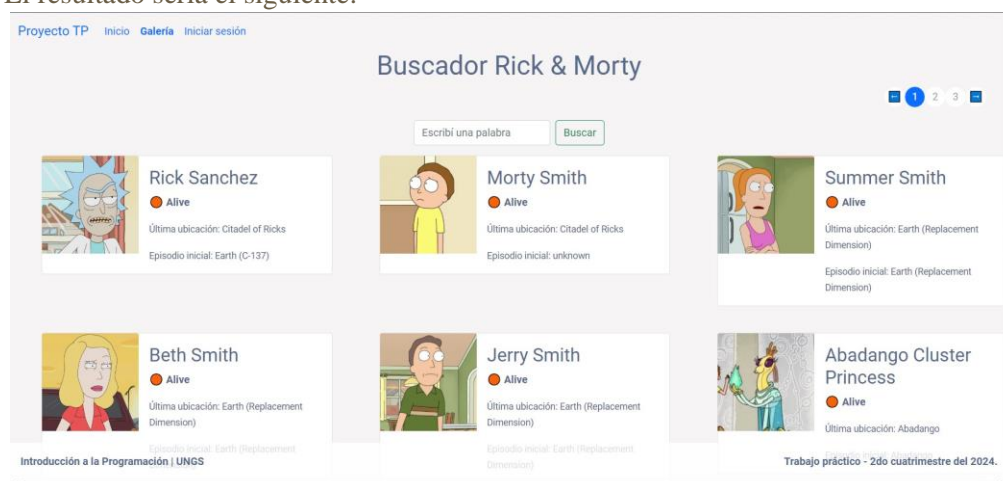
```

# esta función obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los usa para dibujar
# si el opcional de favoritos no está desarrollado, devuelve un listado vacío.
def home(request):
    images = services.getAllImages()
    favourite_list = services.getAllFavourites(request)

```

donde en images traemos desde services a la función getAllImages, que nos trae los datos de la API convertidos en cards.

El resultado sería el siguiente:



Complicaciones: Iniciar con el trabajo fue lo más complicado por no saber dónde empezar, pero cuando nos detuvimos a leer las ayudas dentro del código perdimos el miedo de cambiar el código. Para comenzar primero revisamos absolutamente todas las carpetas a ver que tenían, y cuando vimos que al leer la consigna y ayudas se mencionaban cosas que ya habíamos visto, resultó más fácil conectar hacia dónde ir a buscar la información.



Para comenzar a tener una idea sobre cómo arrancar, leímos información y vimos diferentes videos:

[Cómo utilizar requests de Python con una API Rest](#)

[5. El sistema de importación — documentación de Python - 3.13.0](#)

[javascript - Get collection of elements from JSON - Stack Overflow](#)

[Patrón de Arquitectura MTV \( Modelo - Template - View\) Explicación Simple](#) ✓

[Qué es una API en Programación y cómo funciona | La mejor explicación en español, para principiantes](#)

### (3) home.html:

Primero, se definieron las clases CSS en el archivo styles.css para los bordes de las tarjetas:

¿Qué son las clases?

Las clases CSS son una forma de aplicar estilos específicos a elementos HTML en una página web. Permiten agrupar y reutilizar estilos en múltiples elementos HTML sin tener que escribir el mismo código una y otra vez. Las clases son definidas en una hoja de estilo CSS y luego se aplican a los elementos HTML a través del atributo `class`. ¿Cómo se definen las clases CSS?

- Definición de una clase en CSS: En CSS, las clases se definen utilizando un punto (.) seguido del nombre de la clase. El nombre de la clase puede ser cualquier identificador válido, y puedes usar guiones o guiones bajos para separarlos. Archivos modificados : static/css/styles.css

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

CSS

```
.border-success { border: 2px solid green; }
.border-danger { border: 2px solid red; }
.border-warning { border: 2px solid orange; }
```

```
.border-danger {
  border: 2px solid red;
}
.border-warning {
  border: 2px solid orange;
}
```

Aplicación de Clases CSS en home.html:

Luego, en el archivo home.html, se utilizan estas clases CSS para aplicar diferentes colores a los bordes de las tarjetas, dependiendo del estado del personaje.

Archivo modificados: templates/home.html

Explicación del Código en home.html:

- Estructura de la Tarjeta:

html

```
<span class="border {% if img.status == 'Alive' %}border-success{% elif img.status == 'Dead' %}border-danger{% else %}border-warning{% endif %}">
```

Aquí, la clase `border` se utiliza junto con las clases condicionales `border-success`, `border-danger` y `border-warning` para aplicar un color diferente a los bordes de las tarjetas dependiendo del estado del personaje.

- Condiciones Lógicas:

html

```
{% if img.status == 'Alive' %} ● {{ img.status }}
{% elif img.status == 'Dead' %} ● {{ img.status }}
{% else %} ● {{ img.status }}
{% endif %}
```

Estas condiciones aseguran que se muestre el estado del personaje con el icono y el color adecuados: verde para `Alive`, rojo para `Dead`, y naranja para `Unknown`.

Logro

Las tarjetas ahora tienen bordes coloridos que proporcionan un indicativo visual del estado del personaje, mejorando la usabilidad y la estética del sitio. Los usuarios pueden identificar rápidamente el estado de cada personaje gracias a los bordes codificados por colores, lo que mejora la claridad y el atractivo visual de la información mostradas

Conflictos Encontrados:

- Clases CSS: Asegurarse de que las clases CSS aplicadas corresponden correctamente a los estilos definidos y que no haya errores tipográficos en las tarjetas las cuales ahora tienen bordes coloridos que proporcionan un indicativo visual del estado del personaje, mejorando la usabilidad y la estética del sitio. Los usuarios pueden identificar rápidamente el estado de cada personaje gracias a los bordes codificados por colores, lo que mejora la claridad y el atractivo visual de la información mostrada.

## Buscador ☆:

Para realizar el buscador, tenemos en views una función llamada `search`, que, por su traducción al español, sabemos que trata sobre el buscador:

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

    # si el texto ingresado no es vacío, trae las imágenes y favoritos desde services.py,
    # y luego renderiza el template (similar a home).
    if (search_msg != ''):
        pass
    else:
        return redirect('home')
```

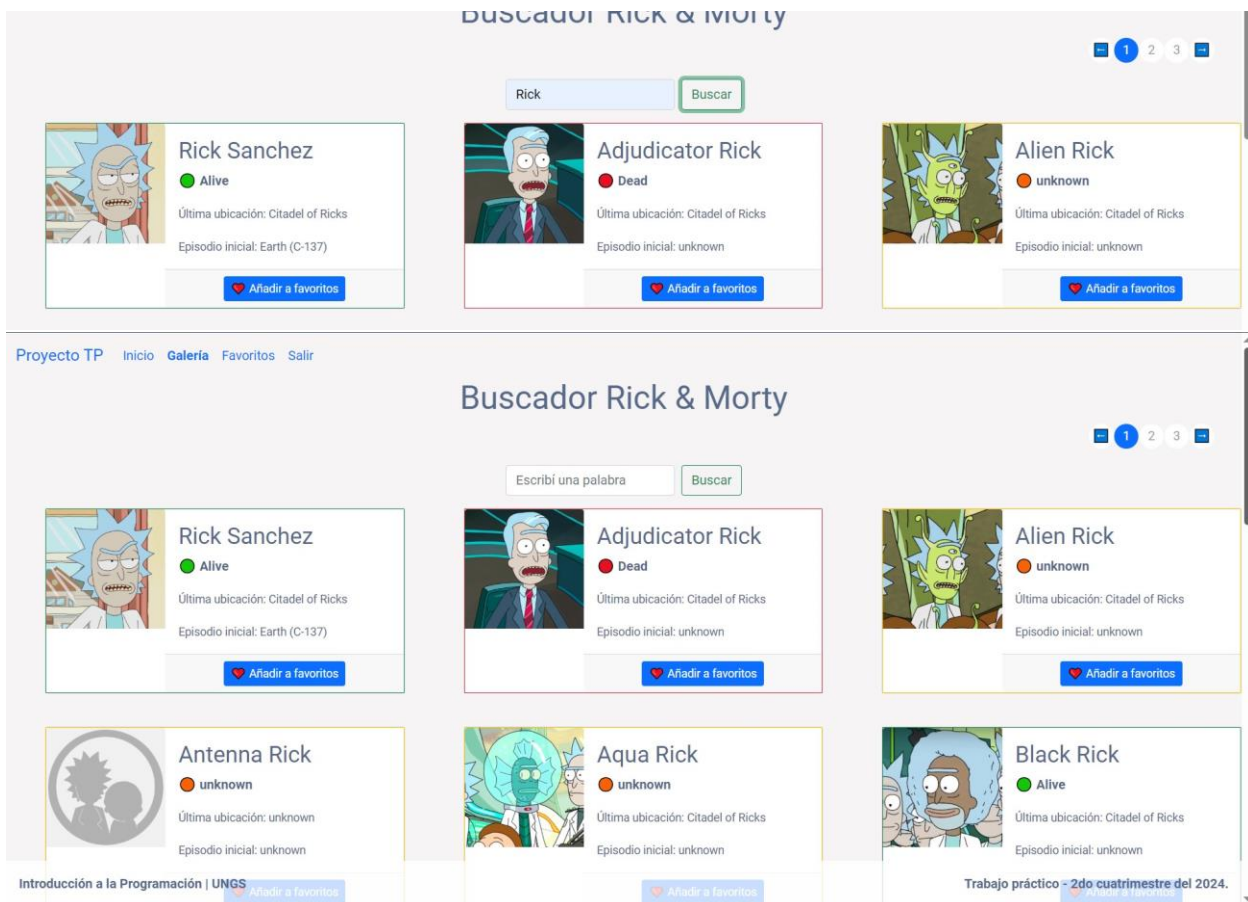
viendo la pista que nos dejan, sabemos que tenemos que realizar un trabajo similar al de home, llamando a services.py para obtener las cards:

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

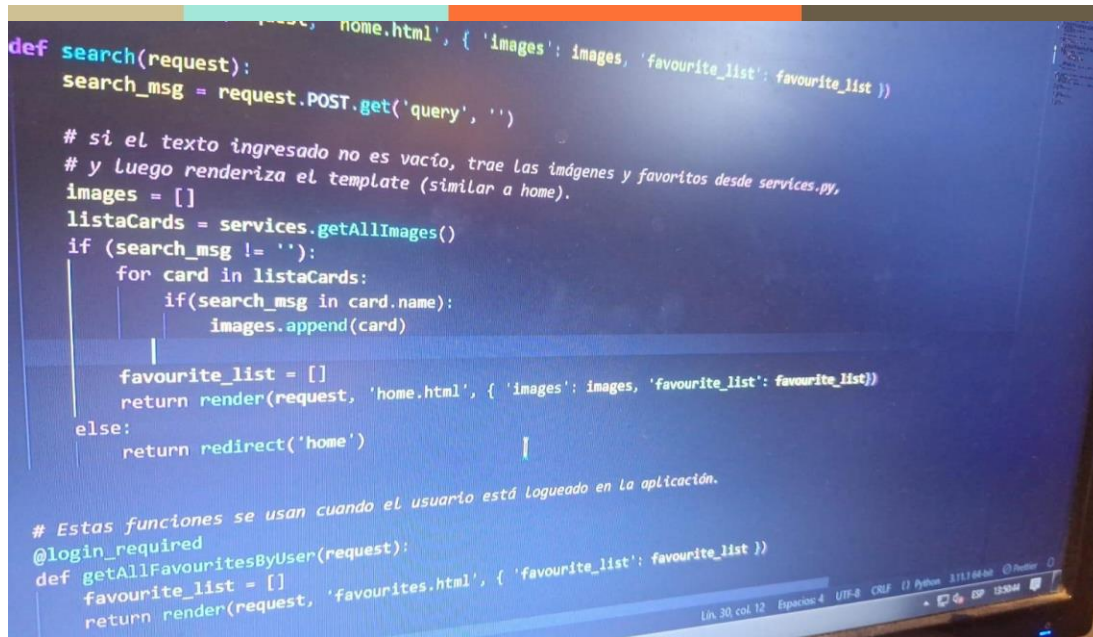
    # si el texto ingresado no es vacío, trae las imágenes y favoritos desde services.py,
    # y luego renderiza el template (similar a home).
    images = []
    favourite_list = []

    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')
```

Una vez que hayamos replicado lo de home dentro del if y agregándole a getAllImages el parámetro del mensaje buscado(search\_msg), así nos selecciona solo los personajes que correspondan a lo buscado, se verá así:



Dificultades: A la hora de realizar el buscador, en un comienzo llegamos a esta primera conclusión:



donde hacíamos una variable llamada “listaCards” sin sentido, ya que luego nos dimos cuenta que podíamos ahorrar código y utilizar el propio `images` para la búsqueda. Sin embargo, este primer acercamiento nos hizo entender mejor cómo recorrer las listas, ya que lo pensamos como una card dentro de esa lista, haciendo que la lógica del código ya esté presente y sepamos de donde partir para mejorar el código final.

## Inicio de sesión ☆ ☆

El iniciar sesión ya estaba implementado en el código original, pero para cerrarla no tenía ninguna funcionalidad los usuarios no podían salir una vez iniciada la sesión.

Lo que implementamos para poder lograr que los usuarios puedan salir fue entrar al archivo de **views.py** en la función **logout**.

Se implementó de lo siguiente;

From `django.contrib.auth` import `logout`

@`login_required`

def `exit` (request):

`logout`(request) return

`redirect`('home')

Esta función lo que hace es que el usuario una vez iniciada sesión pueda salir y lo envíe nuevamente a inicio.

Este ejercicio fue sencillo de hacer gracias a la siguiente documentación:

[46.- Curso Django 2 | Login y Logout con Django](#)

## Favoritos ☆ ☆:

Para permitir que un usuario logeado pueda almacenar imágenes desde la galería como favoritos, trabajamos sobre los siguientes archivos:

### 1º) Services.py

En este archivo trabajamos sobre 2 funciones a saber:

- a- saveFavourite: Esta función guarda la información del personaje elegido por el usuario en la tabla de favoritos en la base de datos.

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

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

Para lograr esto, realizamos 2 modificaciones en la misma:

- Se le asigno a la variable fav la función “fromTemplateIntoCard (request)”, definida en el archivo translator.py. Esta función crea una Card, en la que a cada atributo se le asigna información expuesta en el template.

```
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
```

- Y se le asigno al atributo “user” de la Card contenida en la variable “fav”, el usuario logeado mediante el “get\_user (request)” b- getAllFavourites: Esta función devuelve una lista vacía si el usuario no este logeado. Por el contrario, si lo está, devuelve una lista que contiene los favoritos asociados al usuario en formato Cards.

```
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

```

Para ello, agregamos:

- A la variable “user” se le asigno el usuario en cuestión mediante el uso del “get\_user(request)”
- Además, a la variable “favourite\_list” se le asigno la función “getAllFavourites(user)”, definida en el archivo repositories.py, con el fin de obtener la lista de favoritos asociados al usuario almacenada en la base de datos.

```

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

```

- Por último, dentro del ciclo for que recorre la lista de favoritos, se le asigno a la variable “card” la función “fromRepositoryIntoCard (favourite)” definida en el archivo translator.py. De esta forma, se convierte cada elemento de la lista en formato 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

```

## 2) views.py

En este archivo modificamos 3 funciones, las cuales tienen como restricción que el usuario debe estar logueado para poder acceder a ellas:

- a- `getAllFavouritesByUser`: Esta función recibe la lista de favoritos del usuario y la muestra en la aplicación en formato de tabla en el apartado “Favoritos”.

```
def getAllFavouritesByUser(request):
    favourite_list = services.getAllFavourites(request)

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

Para lograr esto, se le asignó a la variable “`favourite_list`” la función “`getAllFavourites(request)`”, importada desde el archivo `service.py`. De esta forma, se le asigna a la variable una lista que contiene los favoritos del usuario que fueron almacenados en la base de datos, ya traducidos al formato Card.

Por último, esta información se pasa al archivo “`favorites.html`”, donde se define el formato en el que se presenta la información en pantalla.

- b- `saveFavourite`: Esta función guarda la información del personaje en la sección de favoritos de la base de datos asociada al usuario logueado y muestra en la aplicación la sección “Favoritos”.

```
def saveFavourite(request):
    services.saveFavourite(request)

    return redirect('favoritos')
```

Para lograr esto, agregamos:

- La función “`saveFavourite(request)`” importada desde el archivo `services.py`. De esta forma se traduce la información del personaje desde el template ‘home’ en formato card y luego se le asigna el usuario correspondiente para por último almacenarlo en la base de datos.
- Por otro lado, se agregó la función “`redirect('favoritos')`” para redirigir al url almacenado en el archivo “`urls.py`” nombrado ‘favoritos’, por lo que el usuario termina viendo en la aplicación la sección favoritos luego de agregar uno.

- c- `deleteFavourite`: Esta función elimina la información de un personaje almacenado en la base de datos asociado al usuario logueado desde la sección “Favoritos” y permanece en la misma.

```
def deleteFavourite(request):
    services.deleteFavourite(request)
```

```
return redirect('favoritos')
```

Para ello, agregamos 2 funciones:

- En primer lugar, la función “deleteFavourite(request)” importada desde el archivo services.py.

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

Esta función si bien ya estaba definida, se encarga de borrar el favorito de la base de datos según su ID. Para ello asigna a la variable “favId” el id del favorito que se quiere eliminar desde el template “favourites”. Y luego retorna la función “deleteFavourite(favId)” definida en el archivo repositories.py, la cual toma como parámetro el id del favorito desde el template y si en la base de datos existe un favorito con el mismo id entonces lo elimina.

```
def deleteFavourite(id):
    try:
        favourite = Favourite.objects.get(id=id)
        favourite.delete()
        return True
    except Favourite.DoesNotExist:
        print(f"El favorito con ID {id} no existe.")
        return False
    except Exception as e:
        print(f"Error al eliminar el favorito: {e}")
        return False
```

## Paginación de resultados:

Para realizar la paginación tanto de la galería como de los resultados de búsqueda, pensamos en utilizar el class Paginator de Django. Para ello, creímos necesario que al llamar la función home o serch, según corresponda, deberíamos darles una lista de imágenes con todos los personajes de la API general o que contengan el término que se está buscando.

Con el fin de obtener dicha lista, modificamos 4 archivos:

### 1) Config.py

Modificamos la URL asignada a la variable “DEFAULT\_REST\_API\_URL”, eliminando la concatenación con “DEFAULT\_PAGE”

```
DEFAULT_REST_API_URL = 'https://rickandmortyapi.com/api/character?page='
```

### 2) Transport.py

Dado que en esta capa es donde ocurre la comunicación con la API, pensamos en modificar la función `getAllImages(input=None)` original, la cual recibía únicamente los personajes de la API correspondientes a la primera página, tanto si se buscaba un término o no.

```
def getAllImages(input=None):

    json_collection = []

    lista_Urls_Api = []

    if input is None:

        lista_Urls_Api = getApiUrl('-1', getUltimaPag ('-1'))

    else:

        lista_Urls_Api = getApiUrl(input, getUltimaPag (input))

    for url in lista_Urls_Api:

        json_response = requests.get(url).json()

        # si la búsqueda no arroja resultados, entonces retornamos una lista vacía de elementos.

        if 'error' in json_response:

            print("[transport.py]: la búsqueda no arrojó resultados.")

            return json_collection

        for object in json_response['results']:

            try:

                if 'image' in object: # verificar si la clave 'image' está presente en el objeto (sin 'image' NO nos sirve, ya que no mostrará las imágenes).
```

```

        json_collection.append(object)

    else:

        print("[transport.py]: se encontró un objeto sin clave 'image', omitiendo...")

    except KeyError:

        # puede ocurrir que no todos los objetos tenga la info. completa, en ese caso descartamos
dicho objeto y seguimos con el siguiente en la próxima iteración.

        pass

return json_collection

```

Al cambiar la función, nuestro objetivo fue que recorra una lista con las diferentes paginas de resultados de la API para que al retornar la lista “json\_collection”, esta contenga todos los personajes de la API. Para lograr esto, definimos previamente 2 funciones que nos ayudaron a obtener la lista con las diferentes paginas de la API:

a- getUltimaPag(search): Mediante esta función obtenemos la cantidad de páginas de la API con resultado. Para ello, obtenemos la primera página de la API y la convertimos en formato json. Luego, accedemos al apartado ‘info’ y por último a la clave ‘page’.

Como intentamos paginar tanto la galería como los resultados de búsqueda, decidimos modificar la URL según si se busca un personaje o no. Además, agregamos el try y except con el fin que siempre devuelva por lo menos 1 página, necesario ya que modificamos el template ‘home’ para que nos dibuje la cantidad de páginas según la cantidad de personajes obtenidos.

```

def getUltimaPag(search):

    try:

        if search == '-1':

            json_response = requests.get(config.DEFAULT_REST_API_URL +
config.DEFAULT_PAGE).json()

            object = json_response['info']

            cantidadPaginas = object['pages']

```



```

    else:

        json_response = requests.get(config.DEFAULT_REST_API_URL + "1&name=" +
search).json()

        object = json_response['info']

        cantidadPaginas = object['pages']

    return int(cantidadPaginas)

except:

    return 1

```

b- `getApiUrl(search, cant_pages)`: Esta función recibe 2 argumentos, el término de búsqueda y la cantidad de páginas. Y se definió con el fin de crear una lista conformada por una URL por cada página de la API de la cual quiero obtener información.

```

def getApiUrl (search, cant_pages):

    url = config.DEFAULT_REST_API_URL

    lista_Urls = []

    if search == '-1':

        for page in range (1, 6): #Aca definimos el limite arbitrariamente por un problema que
explicamos en el informe

            lista_Urls.append(url + str(page))

    else:

        for page in range (1, cant_pages + 1):

            lista_Urls.append(url + str(page) + "&name=" + search)

    return lista_Urls

```

### 3) Views.py

En este archivo fue donde utilizamos el class `Paginator` para realizar la paginación de las imágenes en las funciones `home` y `search`. Para ello, primero importamos el class `Paginator`:

```
from django.core.paginator import Paginator
```

Luego agregamos en las 2 funciones, 3 variables:

- `images_por_pag`: A esta variable le asignamos la función `Paginator` para que divida la lista total de imágenes en 20 imágenes por página.
- `page`: En esta variable obtenemos el número de página desde la URL. Si el número no está, por defecto es 1.
- `images_page`: A esta variable le asignamos las imágenes que corresponden a la página 'page'

Y modificamos la información que se renderiza a home, antes se le pasaba la lista completa de imágenes, ahora solo se le pasa las imágenes correspondientes al número de página.

```
def home(request):
```

```
    images = services.getAllImages()
```

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

```
    images_por_pag = Paginator(images,20)
```

```
    page = request.GET.get('page',1)
```

```
    images_page = images_por_pag.get_page(int(page))
```

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

```
def search(request):
```

```
    search_msg = request.POST.get('query', request.GET.get('query', ""))
```

```
    # si el texto ingresado no es vacío, trae las imágenes y favoritos desde services.py,
```

```
    # y luego renderiza el template (similar a home).
```

```
    images = []
```

```
    favourite_list = []
```

```

if (search_msg != ""):

    images = services.getAllImages(search_msg)

    favourite_list = services.getAllFavourites(request)

    images_por_pag = Paginator(images,20)

    page = request.GET.get('page',1)

    images_page = images_por_pag.get_page(int(page))

    return render(request, 'home.html', { 'images': images_page, 'favourite_list': favourite_list,
'query': search_msg })

else:

    return redirect('home')

```

En el caso de la función search, tuvimos el inconveniente de que al pasar de una página a otra perdíamos el término de búsqueda y por tanto nos redirigía a home. Para solucionar esto pensamos en colocar el termino de búsqueda en la URL y así tenerlo disponible al pasar de página. Por ello, implementamos las siguientes modificaciones:

- a- Por un lado, el término de búsqueda ('query') se obtiene de la caja de búsqueda cuando el usuario ingresa un término o de la url. Y en caso de que no haya información disponible es vacío.
- b- Y por otro, se agregó "'query':search\_msg" en la información que se renderiza a home, con el fin de pasar el termino de búsqueda a home y allí agregarlo a la URL.

#### 4) Home.html:

En esta parte del proyecto se modificó el código con el fin de que:

- a- Aparezcan los números de páginas según la cantidad de resultados que se obtiene de la API: Para ello se creó un ciclo que recorra el total de paginas creadas por paginator. Y se agregó como condición que si el número de pagina coincide con la pagina actual que muestra entonces el botón de esa pagina pasa a estar "activo", lo cual se relaciona con el style css, pasando a tomar un color blanco diferente al botón inactivo que es de color azul.

```
{% for page_num in images.paginator.page_range %}
```

```

{% if images.number == page_num %}

    <li class="page-item active" aria-current="page">

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

    </li>

{% else %}

    <li class="page-item">

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

    </li>

{% endif %}

{% endfor %}

```

- b- Aparezca en la URL el número de página y el término de búsqueda:

```

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

```

- c- Que funcionen las flechitas para moverse entre páginas: Para esto, se utilizó la función del paginador “has\_previous” y “has\_next”. En ambos casos, la lógica que se utilizó fue pensar si la imagen tiene una página anterior/próxima, según corresponda, entonces se modifica la URL con la información correspondiente. En caso contrario, el botón queda desactivado.

```

{% if images.has_previous %}

    <li class="page-item">

        <a class="page-link" href="?page={{ images.previous_page_number
}}&query={{ query }}" tabindex="-1">←</a>

    </li>

{% else %}

    <li class="page-item disabled" >

        <a class="page-link" href="#" tabindex="-1">←</a>

    </li>

{% endif %}

```

```

{% if images.has_next %}

    <li class="page-item">
        <a class="page-link" href="?page={{ images.next_page_number }}&query={{
query }}">→</a>
    </li>
{% else %}
    <li class="page-item disabled">

        <a class="page-link" href="#">→</a>
    </li>
{% endif %}

```

Algunos de los problemas que encontramos al realizar la paginación fueron:

- La idea inicial al realizar la paginación fue crear una lista que contenga todas las imágenes de la API y pasarle la misma a la función ‘home’, la cual modificamos para realizar la paginación con el uso de la función “paginator”. Si bien tuvimos éxito en la aplicación de nuestra idea, no nos dimos cuenta que al refrescar de página cuando cambiamos a la siguiente, llamamos a la función home, la cual llama a la función “getAllImages” y esta crea de cero la lista con todos los personajes de la API, lo cual ocasiona que la aplicación sea lenta. Por esta razón, decidimos no paginar home por completo y solo mostrar hasta la página 5 de la API. Esto lo resolvimos en transport.py en la función definida como “getApiUrl”.
- Otro problema que tuvimos fue encontrar la forma de movernos entre las páginas de resultado de una búsqueda. Ya que al pasar de una página a la siguiente perdíamos el termino de búsqueda. Para solucionar este problema tuvimos que buscar la forma de agregar esa información a la URL y la forma de obtener de la URL el termino de búsqueda para que muestre solo los personajes relacionados. Esto lo resolvimos y explicamos en views.py y en home.html.

## Renovar interfaz gráfica:

Unas de las cosas que modifique fue los fondos de la pantalla de recibimiento, galería y inicio sesión.

Para lograr lo tuve que investigar en (YouTube, internet).

Una de las dificultades de querer seguir estos tutoriales eran que no especificaba muy bien, por ello otra ayuda que recibí fue de una maestra particular.

La primera pantalla que cambiamos fue la de bienvenidos:

lo que hicimos para modificarla fue cerrar el visual que tenía el código abierto y en la carpeta



que tenía el código busqué static y dentro de ella hice una capeta llamada images dentro de esta puse unas imágenes llamada ricknegro de Rick y Morty wallpaper estas fueron descargadas de internet, pero uno de sus conflictos fue que no me di cuenta que las imágenes no estaba en JPG para poder cambiarlas fue abrir la imagen en Paint y poner guardar como y poner le JPG.

Una vez echo eso volví a abrir el visual con mi código y fui a index.html y al principio del código llame a <style> luego agregue lo siguiente background-image: url ("{% static 'images/mortynegro.jpg' %}");

Para galería hicimos lo siguiente:

En la misma capeta ya antes mencionada agregamos otra imagen llamada fondos de pantalla y de nuevo usamos al principio del código <style> y agregamos en el archivo home.html background-image: url ("{% static "images/fondosdepantalla, wallpapers, inspiracion, space, espacion, galaxy, galaxia, black, dark, stars, estrellas, night, noche.jpg" %}");

A este se le eligió un color negro y en este fondo de pantalla se nos dificulto que nos olvidamos el <style> no entendíamos porque no funcionaba hasta que nos dimos cuenta Por últimos el inicio sesión:

En la misma carpeta de images se agregó una imagen llamada ESTA SI en este caso fuimos al archivo de login.html y agregamos lo siguiente:

```
<style>

body {

    background-image: url ("{% static "images/ESTA SI.jpg" %}");
```

como en todos los demás archivos.

## Conclusión

A través de la investigación en distintas fuentes de información, logramos comprender cómo se interrelacionan las diferentes capas del proyecto, lo que nos permitió completar las consignas solicitadas. Además, aprendimos lo importante que es planificar y organizar cómo resolver un problema antes de empezar a programar, ya que esto ayuda a hacer el trabajo de manera más ordenada y eficiente.