



Universidad  
Nacional General  
Sarmiento

# “Trabajo práctico de Introducción a la Programación”: Informe

○ Comisión: 09 ○

○ Fecha de entrega: 26/11/2024 ○

○ Profesores ○:

- Rodríguez, Miguel
- Soria, Lucía

○ Alumnos ○:

- Almirón, Ariana
- Obregón, Lucas
- Santone, Juan
- Tinuco, Erica

- El trabajo práctico de la materia Introducción a la Programación consiste en completar un programa parcialmente desarrollado, utilizando Django y Python en Visual Studio Code.

En el mismo, se utiliza una API (*Interfaz de Programación de Aplicaciones*, en español), que actúa como un conector que permite que diferentes programas se comuniquen entre sí y compartan información de manera segura. En este caso, mediante la API, es posible acceder a información de los personajes de la serie “Rick & Morty”.

El objetivo principal del trabajo es completar las funcionalidades más importantes del programa, tal como:

- Mostrar imágenes de los personajes, con su respectiva información en “Cards” (Tarjetas). Además, las mismas deben contar con un borde de color verde, rojo o naranja dependiendo del estado del personaje.
- Implementar un buscador que permita localizar a los personajes dentro de la aplicación.
- Desarrollar un sistema donde los usuarios puedan iniciar sesión y guardar sus personajes favoritos.

A continuación, se describirán las funciones completadas en este trabajo práctico:

### • *Mostrar imágenes e información de las “Cards”* •

- Para esta funcionalidad, se nos pidió mostrar la información de los personajes en “Cards” (Tarjetas), donde se pueden ver detalles como el estado del personaje, el episodio donde aparece por primera vez y su última ubicación. Para implementar esta función, tenemos que tener en cuenta que:

La capa “Transport” es la encargada de conectarse con la API. Dentro de esta capa se encuentra la función “GetAllImages”, que devuelve una lista JSON (`json_collection`) para que el resto del programa pueda utilizarlo. En otras palabras, el objetivo de este archivo es traer los datos necesarios en formato JSON, el cual incluye información sobre los personajes de la serie.

```
8 def getAllImages(input=None):
9     # obtiene un listado de datos "crudos" desde la API, usando a transport.py.
10    json_collection = []
11    json_collection = transport.getAllImages(input)
12    # recorre cada dato crudo de la colección anterior, lo convierte en una Card y lo agrega a images.
13    images = []
14    for elemento in json_collection:
15        images.append(translator.fromRequestIntoCard(elemento))
16    return images
```

- En el archivo “Services.py” logramos ver que, inicialmente, hay una función que no está completa. Dentro de la misma, se utilizó una lista vacía llamada “**`json_coleccion`**”, sin embargo, se sobrescribe para llamar a la función “**`GetAllImages`**” del archivo “*Transport*” para lograr llenar esa lista vacía de información “cruda” sobre los personajes de Rick & Morty.

Una vez que tenemos el listado “json\_coleccion”, se va a recorrer cada elemento a través de un For. Durante este recorrido, se utiliza una función de la capa “Translator”, *diseñada para convertir cada elemento JSON en una “Card”* (Tarjeta). Finalmente, estos objetos tipo “Cards” se agregan a una lista llamada “Images”. Esta lista representa los datos que fueron procesados para que se puedan mostrar como tarjetas.

Por último, completamos la función “home” en la capa “views.py, con el objetivo de mostrar las Cards con información en la pagina principal.

```
11 # esta función obtiene 2 listados que corresponden a las imágenes de la API y los favoritos del usuario, y los usa para dib
12 # si el opcional de favoritos no está desarrollado, devuelve un listado vacío.
13 def home(request):
14     images = []
15     favourite_list = []
16     images=services.getAllImages()
17
18     return render(request, 'home.html', { 'images': images, 'favourite_list': favourite_list })
19
```

○ La función comienza con una lista vacía llamada “images”. Sin embargo, esta se sobrescribe llamando a la función “GetAllImages” (desde la capa services.py), por lo tanto, trae toda la información sobre los personajes (como la foto, estado, etc). A su vez, esta función renderiza el archivo, mostrando las imágenes y detalles organizados como Cards en la página principal.

### • Cambiar el borde de las “Cards” (Tarjeta) •

○ En esta funcionalidad las “Cards” cambiarán el color de su borde dependiendo del estado del personaje.


- Si un personaje está vivo (“Alive”), la tarjeta tendrá un borde verde.
- Si está muerto (“Dead”), el borde será rojo.
- Si el estado del personaje es desconocido (“Unknown”), el borde será naranja.

```
35 <div class="row row-cols-1 row-cols-md-3 g-4">
36     {% if images|length == 0 %}
37     <h2 class="text-center">La búsqueda no arrojó resultados...</h2>
38     {% else %} {% for img in images %}
39     <div class="col">
40         {% if img.status == 'Alive' %}
41         <div class="card mb-3 ms-5" style="max-width: 540px; border-color: green;">
42         {% elif img.status == 'Dead' %}
43         <div class="card mb-3 ms-5" style="max-width: 540px; border-color: red;">
44         {% else %}
45         <div class="card mb-3 ms-5" style="max-width: 540px; border-color: orange;">
46         {% endif %}
47     <div class="row g-0">
```

○ En la capa “home.html” logramos ver que el programa contiene un condicional que verifica si la lista de imágenes (“Images”) se encuentra vacía o no. Si no hay resultados, se mostrará en pantalla: “La búsqueda no arrojó resultados”. Si fuera el caso contrario, se ejecuta un bucle For para recorrer todos los elementos de la lista Images. Dentro de este bucle, nosotros utilizamos un condicional de Django que determina el color del borde de cada tarjeta según el estado del personaje (**img.status**).

- Si el personaje está vivo (***if img.status == 'Alive'***), el borde será de color verde.
- Si el personaje murió (***elif img.status == 'Dead'***), el borde será rojo.
- Si ninguna de las condiciones anteriores se cumple, el borde será naranja para aquellos personajes con estado desconocido.




```

52 <div class="col-md-8">
53 <div class="card-body">
54 <h3 class="card-title">{{ img.name }}</h3>
55 <p class="card-text">
56 <strong>
57 {% if img.status == 'Alive' %}  {{ img.status }}
58 {% elif img.status == 'Dead' %}  {{ img.status }}
59 {% else %}  {{ img.status }}
60 {% endif %}
61 </strong>

```

Además del color del borde, se implementó una funcionalidad para mostrar un emoji y el estado del personaje dentro de la “Card”, dependiendo del estado del personaje.

El condicional, en un principio, utilizaba “*True*”. Sin embargo, lo reemplazamos por “***img.status***”, para que el programa determine y muestre el símbolo correspondiente junto con el estado del personaje.

- Si el personaje se encuentra vivo se mostrará en la tarjeta:  “Alive”.
- Si está muerto se mostrará:  “Dead”.
- Y si el estado del personaje es desconocido entonces se mostrará en la Card:  “Unknown”.

En esta sección utilizamos recursos como “How to Use If/Else Conditions In Django Templates” para comprender cómo se implementan los condicionales en Django y “Bootstrap Cards Utilities”, para la personalización de los bordes de las tarjetas.

### • **Buscador** •

- Para completar esta funcionalidad, se nos pide que el buscador filtre adecuadamente las imágenes según una serie de criterios básicos:
  - Si el usuario no ingresa algún dato y realiza una búsqueda, se generarán las mismas imágenes como si hubiera entrado sobre el enlace “Galería”.
  - Por otro lado, si el usuario ingresa algún dato en el buscador, se desplegarán las imágenes guardadas relacionadas a dicho dato.

Antes de comenzar con el desarrollo del buscador, primero es necesario comprender la función “GetAllImages”, la cual está ubicada en la capa “services.py”. Esta función se encarga de transformar los datos “crudos” obtenidos desde “Transport.py”, en un formato llamado “Card”. *El parámetro de “GetAllImages”, el cual es (input=None), indica que, al no recibir un argumento, devolverá todas las imágenes disponibles en la lista que maneja esta función.*

Una vez entendido esto, continuaremos en la capa “*views.py*”, específicamente en la función “*search*”, la cual está encargada de gestionar la búsqueda.

```
20 def search(request):
21     search_msg = request.POST.get('query', '')
22
23     # si el texto ingresado no es vacío, trae las imágenes y favoritos desde services.py,
24     # y luego renderiza el template (similar a home).
25     if (search_msg != ' '):
26         images=services.getAllImages(search_msg)
27         return render(request, 'home.html', { 'images': images })
28     else:
29         return redirect('home')
30
```

- En esta función se puede observar la variable “***search\_msg***”, que captura el texto ingresado por el usuario en el buscador. En la misma, se aplicó un condicional:

- Si el usuario ingresa un texto, la función llamará a “*GetAllImages*” desde la capa “*services.py*”, pasándole como parámetro el texto ingresado (*search\_msg*). *Esto permite obtener una lista de resultados relacionados con la búsqueda del usuario.*

- En caso de que el usuario no escriba nada en el buscador, *el programa redirigirá al usuario a la página principal.*

### • Inicio de sesión •

- Para que esta función trabaje correctamente, se nos pidió completar la “*feature*” o característica de inicio de sesión de la aplicación. En este caso, el usuario y contraseña se encuentran previamente almacenados en la base SQLite, con los correspondientes nombre y contraseña: *admin / admin*. Se nos pidió también tener una serie de consideraciones a seguir.

El desarrollo de la función está parcialmente hecho, la dificultad en el inicio de sesión radica en la salida de la sesión del usuario, donde al no estar programada salta un error. Para la solución de ello, realizamos los siguientes cambios:

- Primero importamos el cierre de sesión “*logout*” para permitir que los usuarios pudieran cerrar sesión. Para ello, utilizamos el módulo “*django*”, y sus submódulos “*contrib*”, y dentro de esta, el módulo “*auth*”, donde manejamos la autenticación de los usuarios, contraseñas y permisos para poder salir.

```
6 from django.contrib.auth import logout
7
```

- En segundo lugar, definimos la función “***exit***” que originalmente contenía solo un “*pass*”, un marcador de posición que no hace nada. Para luego de modificarlo y redirigir a la URL asociada a la vista “*logout*”, asegurarnos que al acceder a “*exit*”, un usuario autenticado sea redirigido al proceso de cierre de sesión.

```

46 @login_required
47 def exit(request):
47 -     pass
48 +     return redirect('logout')

```

### • Favoritos •

○ La funcionalidad de favoritos está conectada con la función “Inicio de sesión”, ya que para que los usuarios puedan guardar a los personajes en su lista de favoritos, es necesario que estén logueados en el sistema. El objetivo es que un usuario pueda seleccionar uno o varios personajes como “favoritos” mediante un botón ubicado en la parte inferior de cada Card. Hay dos observaciones que debemos cumplir

- Si una imagen ya ha sido añadida, debe aparecer un botón que impida reañadirlo.
- Debe existir una sección llamada “Favorito” que permita listar todos los agregados del usuario, en formato de tabla. En la misma, debe existir un botón que permita eliminar el personaje favorito.

Si bien no completamos el opcional, avanzamos con la función “saveFavourite” en la capa services.py, que es una parte fundamental. Sin embargo, esta función por sí sola no es suficiente para que el sistema de favoritos funcione en su totalidad.

```

app/layers/services/services.py
@@ -17,8 +17,8 @@ def getAllImages(input=None):
17
18 # añadir favoritos (usado desde el template 'home.html')
19 def saveFavourite(request):
20 -     fav = '' # transformamos un request del template en una Card.
21 -     fav.user = '' # le asignamos el usuario correspondiente.
20 +     fav = translator.fromTemplateIntoCard(request) # transformamos un request del template en una Card.
21 +     fav.user = request.POST.get("user") # le asignamos el usuario correspondiente.
22
23     return repositories.saveFavourite(fav) # lo guardamos en la base.
24

```

El código de esta función tiene el propósito de añadir imágenes como favoritos y nosotros realizamos los siguientes cambios.

○ Modificamos las variables “fav” y “fav.user” en la función “saveFavourite(request)”:

- fav: Traducimos el “request” del usuario en un formato comprensible por el sistema, esto por medio de “translator.fromTemplateIntoCard(request)”. Dicho de

otra forma, mapea el conjunto de datos de la “Card” y almacenarlos en otro formato comprensible en la lista de favoritos.

- fav.user: Asignamos el valor del campo “user” del formulario enviado previamente en el “POST” por medio de “request.POST.get(“user”)”, para identificar al usuario que agregó el favorito y almacenarlo en la base de datos.

```
app/views.py
@@ -37,7 +37,8 @@ def getAllFavouritesByUser(request):
37
38 @login_required
39 def saveFavourite(request):
40 - pass
40 + services.saveFavourite(request)
41 + return getAllFavouritesByUser(request)
42
43 @login_required
44 def deleteFavourite(request):
```

- Listado dinámico: Por medio del archivo “views.py”, terminamos de definir la función “saveFavourite(request)”, al primero delegar el procesamiento del favorito a la función del mismo nombre en el archivo “services.py”; y luego una vez guardado el favorito, actualizar la lista por medio de “getAllFavouritesByUser(request)” e ir agregando un nuevo favorito.

### **Conclusión:**

A lo largo del desarrollo y resolución del trabajo práctico, no solo aprendimos a utilizar nuevas herramientas como Git, Github o incluso Visual Studio Code de una manera más dinámica, sino también pusimos en práctica toda la teoría asimilada sobre los fundamentos de Python a lo largo de la cursada. Esto incluye el uso de los condicionales, operadores y funciones. Además, el hecho de llevar a la práctica los cambios que uno hace fuera del pensamiento en Seudo-Código y el papel, fue una experiencia enriquecedora que nos llevó a pensar y ver de una manera más amplia la programación y lo que refleja tanto en la vida cotidiana como en las redes.

El trabajo presentó un grado de dificultad adecuado en relación con los conocimientos que fuimos adquiriendo a lo largo de la cursada, lo que facilitó la comprensión de cada función y su propósito. De igual modo, enfrentamos complicaciones al manejar la relación con los archivos, su cantidad, la conexión entre ellos y las funciones específicas de cada uno. Esto resultó especialmente desafiante en la implementación de funciones como “Favoritos”, lo que nos obligó a detenernos y reflexionar más profundamente en su resolución.