UNIVERSIDAD NACIONAL
DE GENERAL SARMIENTO

INTRODUCCIÓN A LA
PROGRAMACIÓN
COM-07

INFORME DE TRABAJO

BUSCADOR RICK & MORTY

Alumnos: Leonel Rolón, Nicolas Rolón

Docentes: Sergio Santa Cruz, Nahuel Sauma, Nancy Nores

Fecha de presentación: 29 de noviembre de 2024

ÍNDICE

INTRODUCCIÓN

Este informe tiene como objetivo detallar el proceso de desarrollo del trabajo realizado, que consiste en completar o corregir porciones de algoritmos para hacer funcionar una página web. A lo largo del documento, se exponen los desafíos enfrentados durante el proyecto y las estrategias implementadas para superarlos. Además, se describe la lógica utilizada para desarrollar y completar el código, abordando cada uno de los elementos propuestos y alcanzados.

No obstante, este informe no se limita únicamente a aspectos técnicos; también incluye una reflexión sobre el aprendizaje adquirido durante el desarrollo del proyecto. Se busca identificar las habilidades desarrolladas, tanto en el manejo de herramientas y lenguajes de programación, como en la capacidad de resolver problemas de manera efectiva. Este análisis pretende resaltar el progreso logrado y la importancia formativa de este ejercicio, tanto a nivel personal como profesional.

DESARROLLO

El desarrollo del informe se organizó con base en el orden de las modificaciones que se realizaron, según lo establecido en la consigna del trabajo práctico. Por lo tanto, los apartados que se describen a continuación son:

- Familiarización con las aplicaciones y código implementado
- Visualización de las cards en la página web
- Color del punto y borde de las cards
- Opcional de buscador
- Opcional de inicio de sesión
- Opcional de favoritos

Familiarización con las aplicaciones y código implementado

En primer lugar, se dedicó un tiempo considerable a indagar respecto de los programas que se debían implementar, debido al desconocimiento que se tenía sobre estos. Desde el comienzo, se siguió cuidadosamente el tutorial ubicado en “readme”, otorgado por los docentes de la cátedra, y aún así, en algunas ocasiones, la instalación se vió dificultada; concretamente las primeras complicaciones aparecieron con la apertura de la consola en la carpeta del repositorio local, o bien, con la implementación del Visual Studio Code en dicho repositorio. Posteriormente, también presentó conflicto -aunque el término es exagerado en este contexto- encontrar la consola del VS Code e iniciar la página web. En este sentido, algunos tutoriales fueron de ayuda para abrir la línea de comandos de Windows (cmd) en la carpeta del proyecto y, desde allí, abrir VS Code con el comando “*code .”*
También es de importancia remarcar, que la parte más escabrosa de la investigación previa, fue encontrar con código completamente desconocido de HTML, Java e incluso de Python, puesto a que se implementan muchas funciones no trabajadas hasta el momento. Lo cual llevó a una exhaustiva investigación de cada línea del algoritmo.

En marco de lo anterior dicho, es crucial mencionar que se hizo uso de la inteligencia artificial (chatGPT, Copilot, GeminAI) con fines orientativos respecto de la naturaleza del código presentado por parte de los docentes; esto fue necesario para entender la estructura y el funcionamiento del proyecto.

Visualización de las cards en la página web

En lo que respecta al código desarrollado, en principio, se indaga un tiempo considerable sobre dónde se podría encontrar la solución, observando entonces cada función, o archivos interconectados con los mencionados en los primeros dos puntos obligatorios (*views.py y*

service.py). Este procedimiento llevó al grupo a aprender sobre las API, cómo en este caso utilizan JSON como formato para obtener información de cada personaje en cada capítulo de la serie Rick & Morty. De esta forma, se logró conectar la información que proviene de la API (ver imagen A.1 del ANEXO), la cual ya está implementada en el módulo *config.py* y utilizada en *transport.py* para la obtención de datos crudos y filtrado estos mediante *service.py*.
En este sentido, se entendió que había que hacer algo con esa información filtrada como datos crudos en el archivo *service.py*, por ende, se implementó las siguientes líneas de código:

```
def getAllImages(input=None):
    json_collection = transport_getAllImages(input)
    images = []
    for imagenesCrudas in json_collection:
        cards_traducidas=translator.fromRequestIntoCard(imagenesCrudas)
        images.append(cards_traducidas)
    return images
```

Por lo cual, se logró devolver una lista de cards (*images*) mediante la función *getAllImages()*. debido a que se utiliza la función de obtención de datos crudos(*transport_getAllImages*) de *transport.py,* en la variable *json_collection*. Entonces, *transport_getAllImages(input)* obtiene la información desde la API y retorna una lista de JSON (objeto iterable). Por último, se recorre cada dato (JSON), se utiliza la función *fromRequestIntoCard* de *translator.py*, la cual se encarga de transformar los datos crudos en cards y se agregan a una lista denominada “images”.

En relación con *views.py*, la función antes mencionada de *services.py* sirve para filtrar la información que se muestra en home, debido a que, se van a mostrar todas las cards en la lista o solo las relacionadas al input. En este sentido, la función *home(request)* toma las *cards* correspondientes para retornarlas hacia el frontend, como se aprecia en las siguientes líneas de código incorporadas:

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

Por último, se comprendió que estos datos obtenidos en el backend se utilizan en el frontend(template), específicamente en *home.html*, donde se ciclan en el siguiente código de la línea 40: *{% for img in images %}*.

Es importante destacar en este punto que, debido a que todavía no se había desarrollado el opcional del buscador, el código de *getAllImages* no tenía “input”:

```
json_collection = transport_getAllImages()
```

Por ende, posteriormente presentó problemas al lograr dar con la lógica para que funcione correctamente el buscador. También hubo dificultad al hacer commit y push, porque el caché presentaba problema, para lo cual se debió investigar y agregarlo al *.gitignore*. Más adelante, se agregaron las líneas correspondientes para que no se suba al repositorio remoto el entorno virtual.

Información agregada al *.gitignore*:

```
**/__pycache__/
*.pyc
# Ignorar el entorno virtual
/env/
/.env/
```

Además, otro problema que no se pudo solucionar y que fue percibido al intentar descargar en la otra computadora el repositorio remoto, es que se debe subir 2 veces el commit, para que, al descargarlo desde el otro ordenador, figure el primer commit cargado. Es decir, no se descargan todos los cambios al repositorio local, si sólo se subieron en un commit.

Color del punto y borde de las cards

En cuanto a el color de las cards, no hubo mayor complicación que un poco de investigación respecto al formato que utilizan los archivos html; Por lo cual, se fue deduciendo que, en principio tienen muchas anidaciones o tabulaciones y que cada porción de código se abre y se cierra mediante la utilización de etiquetas, por ejemplo: . En este caso, el código ya estaba prácticamente terminado, por lo que, las modificaciones no fueron muchas.

En este sentido, dentro de la etiqueta <*strong*></*strong*> se encontraba la información para configurar el color del punto de la *card* según el estado del personaje (vivo, muerto o

desconocido). Esta información no cumplía con la estructura básica del uso de condicionales en Django. Las condiciones del *{% if %}* no estaban correctamente indentadas, ya que cada *{{ img.status }}* debía estar dentro de la condición y no en la misma línea de código.

Ocurría también que el código tenía como condición *{% true == 'Alive' %}*, lo que comparaba un booleano con “alive”, “dead” o “unknown”, por lo cual, se cambió la palabra true por *img.status* en cada condición (ejemplo:*{% if img.status == 'Alive' %}*). Por ende, se trae la clave “status” del JSON, que tiene en cada caso uno de los tres valores mencionados anteriormente.

A continuación se muestran las líneas de código que se reemplazaron y las que se insertaron, respectivamente:

~~~~
~~{% if true == 'Alive' %}~~ 🟢 ~~{{ img.status }}~~
~~{% elif true == 'Dead' %}~~ 🔴 ~~{{ img.status }}~~
~~{% else %}~~ 🟠 ~~{{ img.status }}~~
~~{% endif %}~~
~~~~

```
<strong>
    {% if img.status == 'Alive' %}
        🟢 {{ img.status }}
    {% elif img.status == 'Dead' %}
        🔴 {{ img.status }}
    {% else %}
        🟠 {{ img.status }}
    {% endif %}
</strong>
```

Se corrigió también la estructura de un bucle for dentro de la etiqueta * * ya que el ciclo *{% for img in images %}* se abría en la misma línea en la que se encontraba el *{% else %}*. Entonces, es dentro de este ciclo que se agregaron las condiciones para el borde de la imagen. Concretamente se reemplazó la línea de código * * por la línea *{% if img.status == 'Alive' %}border-success{% elif img.status == 'Dead' %}border-danger{% else %}border-warning{% endif %}" style="max-width: 540px;">*

Código incompleto y mal indentado:

```
<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">
```

Código corregido:

```
<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
                        {% if img.status == 'Alive' %}border-success{% elif img.status == 'Dead'
%}border-danger{% else %}border-warning{% endif %}"
                    style="max-width: 540px;">
                    <div class="row g-0">
```

Por último, cabe mencionar que, se tuvieron que hacer varias modificaciones en el orden e indentaciones de cada etiqueta para que funcione correctamente el *home.html*. Cada etiqueta debe cerrarse con la misma indentación que se abrió; muy similar a lo que se trabajó en la cursada con Python. Lo cual se encuentra reflejado en los commits del repositorio remoto (ver imagen A.2 en el ANEXO).

Opcional de buscador

En esta ocasión, encontramos mayor complejidad a la hora de resolverlo, ya que intentamos hacer una comparación sobre los datos de las *cards* con lo que se coloca en el input(buscador). Entonces, intentamos implementar en la función que retorna las *cards*, en *services.py*, un ciclo que recorra las cards enlistadas y no logramos dar con la solución definitiva. Posteriormente, al revisar y consultar con el docente, nos percatamos que el módulo *transport.py* ya tenía implementado este código donde podía entrar por la rama del input, filtrando los datos JSON obtenidos (ver imagen A.3 en el ANEXO), por lo cual, lo

único que había que hacer, era agregar a la función *getAllImages* de *services.py*, en la línea 12 del código, un input para que, en caso de que haya búsqueda, entre por esa rama. Lo cual se ve de la siguiente manera: *json_collection = transport_getAllImages(input)*

Opcional de inicio de sesión

En este apartado se buscó que funcione correctamente el inicio y el cierre de sesión. Se encontró que estaba habilitado el inicio de sesión, pero no era posible cerrar la sesión. Al intentarlo, la página generó un error, el cual fue copiado y analizado para identificar su causa.

Dentro de la carpeta *layers>templates* en *views.py* se halló, entre otras funciones sin completar, la función *def exit(request)* en la que el contenido sólo era la palabra clave *pass*, indicando que esa porción de código debía saltarse. Su nombre dió una idea que se trataba de la función para el cierre de sesión. Por ende, la línea de código *pass* fue removida y cambiada por *logout(request).* La siguiente y última línea de la función se completó con *return redirect('login')*, tal como se muestra a continuación:

~~@login_required~~
~~def exit(request):~~
~~ pass~~

```
def exit(request):
    logout(request)
    return redirect('login')
```

La función *logout(request)* ya estaba configurada e incluida en el framework Django y sirve para eliminar los datos de la sesión del usuario (ver imagen A.4 del ANEXO). Dicha función se logró hallar junto con muchas otras funciones dentro de una carpeta oculta llamada *_init_.py* ubicada en la siguiente ruta:

> *AppData > Local > Programs > Python > Python313 > Lib > site-packages > django > contrib > auth > _init_.py*

Al investigar sobre esta carpeta se detectó que forma parte del manejo de autenticación y sesiones de los proyectos en Django. En ella se observó también la función *login(request,*

user, backend=None) entre otras funciones de permisos, credenciales, seguridad y gestión de claves de los usuarios (ver imagen A.5 del ANEXO).

La función *redirect('login')* estaba incluida también en Django (ver imagen A.6 del ANEXO) y redirige al usuario nuevamente al inicio de sesión (aunque podría ser configurada para ser redirigida a otra URL). Esta función se encuentra en la ruta:

> *AppData > Local > Programs > Python > Python313 > Lib > site-packages > django > shortcuts.py > redirect*

Se desconoce la utilidad del módulo *shortcuts.py*, pero se estima que sigue la línea de modularidad y mantiene el código limpio.

Opcional de favoritos

En este apartado se buscó que, al iniciar sesión, se le permita al usuario agregar o quitar favoritos desde cada *card* de los personajes; Además, que se muestre una lista de favoritos en su correspondiente ventana y que se puedan eliminar los mismos y puedan volver a estar disponibles para ser agregados cuando el usuario lo desee.

Al revisar los diferentes módulos se encontró que el módulo *repositories.py* ubicado en la ruta *app > layers > persistance > repositories.py* tenía desarrolladas las funciones para guardar favoritos, eliminar favoritos y armar una lista con todos los favoritos agregados; Concretamente las funciones encontradas fueron: *saveFavourite(image), deleteFavourite(id), getAllFavourites(user)*. Cada una de estas funciones necesita utilizar la estructura de un modelo de favoritos (*Favourite*) que se encuentra ubicada en el módulo *models.py* y que almacena ciertos datos como url, nombre, ubicación, importando a su vez ciertas configuraciones de Django que se desconoce su funcionamiento.

Se utilizaron las funciones de *repositories.py* en los módulos *services.py* y *views.py* con distintos fines. En el primer caso se completaron las funciones *saveFavourite(request) y getAllFavourites(request)* de la siguiente manera:

- Para la función *saveFavourite(request)* se completó la variable *fav* con la función *fromTemplateIntoCard(request)* importada desde *translator.py*. Fue de gran ayuda el comentario "transformamos un request del template en una Card." que ya se encontraba escrito a la derecha de la variable *fav*; esto daba idea de que el parámetro correcto para la función era *request* y que debíamos buscar la función

donde teníamos todas las funciones que transformaban a *Card* la información. De la misma manera, en el módulo *translator.py*, arriba de la función *fromTemplateIntoCard(templ)* fue de gran ayuda el comentario “usado cuando la info. viene del template, para transformarlo en una Card antes de guardarlo en la base de datos.” Luego se agregó en la variable *fav.user* la solicitud de usuario *request.user* como bien lo decía también el comentario “le asignamos el usuario correspondiente.”

```
def saveFavourite(request):
    fav = ''
    fav.user = ''
    return repositories.saveFavourite(fav)

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

- En la función *getAllFavourites(request)*, en la rama del *else*, se completó la lista *favourite_list* con la función *getAllFavourite(user)* importada desde *repositories.py*. Al igual que en el caso descrito en el párrafo anterior, el comentario “buscamos desde el repositories.py TODOS los favoritos del usuario (variable 'user')” no sólo indicaba el módulo donde buscar sino también la variable. Luego, en la variable *card*, con ayuda también del comentario, se buscó en *translator.py* la función que transforme cada favorito en una card que sería “appendeada” en la lista *mapped.favourites*; En este caso la variable debía ser favourite ya que cada *card* se creaba dentro de un ciclo for que recorría la lista *favourite_list*.

```
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 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
```

En el caso del módulo *views.py* hubieron modificaciones en todas las funciones, aunque algunos de esos cambios no fueron útiles y se tuvieron que borrar. Las modificaciones en este módulo, respecto del opcional de favoritos, fueron:

- En cada función que se encontró la lista vacía *favourite_list* se completó con *getAllFavourites(request.user)*. Fue correcto completar dicha lista dentro de la función *getAllFavouritesByUser(request)*; El error fue completarla también en las funciones *search(request)* y *home(request)*, ya que al hacerlo dejaba de visualizarse la galería de cards cuando no había un usuario ingresado. Esto ocurrió porque *search* y *home* no eran funciones que requerían del inicio de sesión, entonces se borró el cambio en dichas funciones y se volvió a dejar vacía la lista *favourite_list*.

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

def getAllFavouritesByUser(request):
    favourite_list = repositories.getAllFavourites(request.user)
```

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

- Las funciones que se completaron en *service.py* se utilizaron para completar, en *views.py*, *def deleteFavourite(request)* y *def saveFavourite(request)*. Para ambos casos el *return* es un *redirect* que nos envía nuevamente a la página de favoritos.

```
~~def saveFavourite(request):~~
~~    pass~~

~~def deleteFavourite(request):~~
~~    pass~~

def saveFavourite(request):
    services.saveFavourite(request)
    return redirect('favoritos')

def deleteFavourite(request):
    services.deleteFavourite(request)
    return redirect('favoritos')
```

Es importante destacar que se comprendió que los parámetros pasados a las funciones *redirect* en *views.py* tienen que ver con la información encontrada en las *urlpatterns* de *urls.py*. Si los parámetros no figuran en las urls, no funcionan los redirect. Tal es el caso que se dió al colocar *redirect(‘favourites’)* en lugar de *redirect(‘favoritos’)*. El error pudo solucionarse consultando a una fuente de inteligencia artificial que respondió que revisemos los *path* de las urls (ver imagen A.7 del ANEXO).

CONCLUSIONES

El desarrollo de este proyecto nos permitió aprender sobre diversos aspectos relacionados con la programación, el trabajo colaborativo y el aprendizaje autónomo. Un aspecto clave, para lograr integrar las distintas partes del proyecto, fue entender que el correcto nombramiento de carpetas, archivos, funciones y variables no solo facilita la comprensión del código, sino que también evita errores al momento de importarlas o referenciarlas en distintos módulos.

Además, la modularidad y los comentarios claros en ciertas partes del código fueron de gran ayuda, especialmente al tratar con un lenguaje que inicialmente ignorábamos por completo, como HTML. No obstante, se encontró desafío en la interpretación del tutorial en inglés, pero pudo ser comprendido. Así también, el uso de recursos digitales como ChatGPT, Copilot y Gemin(IA) fue esencial para resolver interpretaciones a priori sobre gran parte del código dispuesto por los docentes. Estas tecnologías nos brindaron orientación inmediata y contribuyeron a superar barreras y a optimizar los tiempos. En este sentido, el proceso de instalación de herramientas y frameworks como Visual Studio Code, Python y Django fue uno de los primeros desafíos que permitió entender la importancia de seguir guías detalladas y conocer los requisitos previos para evitar futuros conflictos.

En cuanto a la gestión del tiempo y la priorización de tareas, aunque se enfrentaron desafíos y limitaciones, se administraron los esfuerzos y las horas disponibles de manera eficiente, implementando la mayoría de las funciones propuestas. Con más tiempo se habrían explorado nuevas funcionalidades como el paginado, el añadir comentarios y el spin de carga. Y, aunque no se logró solucionar todos los problemas, como el “doble commit” necesario para reflejar cambios en otros dispositivos, cada inconveniente representó una oportunidad de aprendizaje. Debido a esto, comprendimos mejor las configuraciones de Git.

En términos de habilidades adquiridas, este proyecto nos permitió desarrollar habilidades técnicas, como la interpretación y el manejo de APIs, la integración de frontend y backend, y la depuración de código. Y, a modo de autocrítica, el hecho de redactar el informe luego de haber completado el código (y no a la par) dificultó más la tarea; Detalle que pudo solucionarse gracias al registro del repositorio remoto inicial compartido por los docentes, donde figura toda la información sin editar para un mejor contraste.

En resumen, este proyecto representó un avance significativo en nuestra formación, tanto a nivel técnico como personal. Nos brindó una valiosa experiencia que nos motiva a continuar

explorando el mundo del desarrollo de software y a seguir perfeccionando nuestras habilidades tanto en backend como en frontend.

BIBLIOGRAFÍA

Microsoft. (n.d.). Marketplace de extensiones de Visual Studio Code. Visual Studio Code. https://code.visualstudio.com/docs/editor/extension-marketplace

MitoCode. (2021, febrero 23). Curso de Git y Github - 13 fork. [Video]. YouTube. https://www.youtube.com/watch?v=9YUaf-uxuRM

CS Wong. (2020, octubre 26). Multilayered software architecture. Medium. https://medium.com/@e0324913/multilayered-software-architecture-1eaa97b8f49e

Bootstrap. (n.d.). Cards. Bootstrap. https://getbootstrap.com/docs/4.0/components/card/

Victor Nagorny. (2019, mayo 8). If-else conditions in Django templates. Medium. https://medium.com/powered-by-django/if-else-conditions-in-django-templates-5b3658cbf287

Code Band. (2020, agosto 13). Django Authentication Tutorial | Part 1 : Login and Logout in Django | Example By Code Band. [Video]. YouTube. https://www.youtube.com/watch?v=oKuZQ238Ncc

Introd. a Programación UNGS. (2021, junio 18). Introducción a JSON. [Video]. YouTube. https://www.youtube.com/watch?v=FGG-UTCwlJw

Introd. a Programación UNGS. (2021, mayo 24). Introducción a GIT. [Video]. YouTube. https://www.youtube.com/watch?v=mzHWafbVRyU

Introd. a Programación UNGS. (2024, octubre). Presentación del TP - 2C2024 - Introducción a la Programación [Video]. YouTube. https://www.youtube.com/watch?v=7qMqPtmPNwA

Introd. a Programación UNGS. (2022, febrero 10). GIT - Ramas (branches). [Video]. YouTube. https://www.youtube.com/watch?v=BRY9gamL9PE

ANEXOS

ANEXO 1: imágenes

Imagen A.1: captura de pantalla de una parte de la página de la API

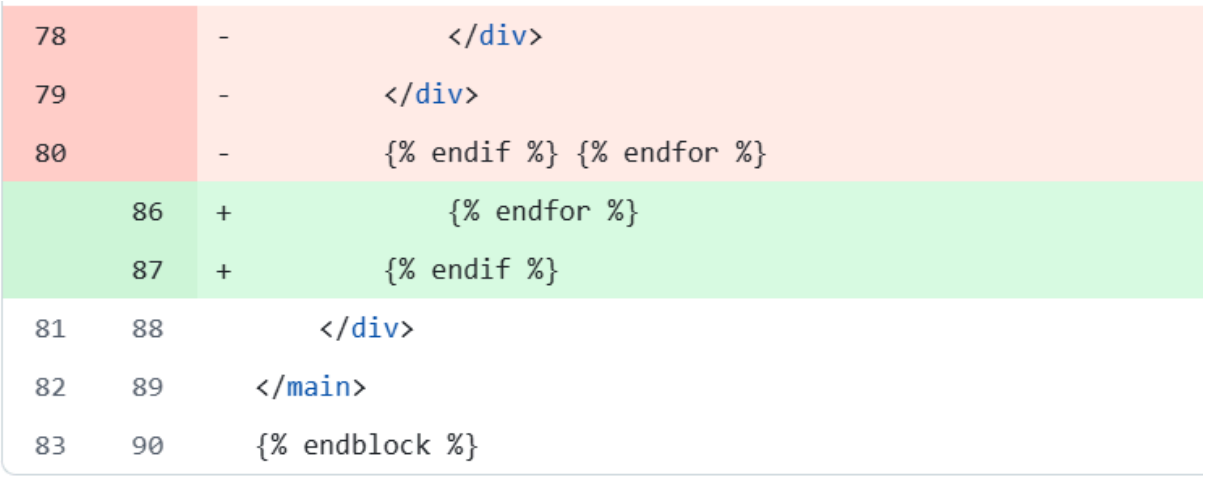

Imagen A.2: corrección de cierres de ciclos, condicionales e indentaciones en HTML

```python
# comunicación con la REST API.
# este método se encarga de "pegarle" a la API y traer una lista de objetos JSON crudos (raw).
def getAllImages(input=None):
    print (input)
    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 = []
```

Imagen A.3: código de transport.py que obtiene los datos crudos de la API

```python
def logout(request):
    """
    Remove the authenticated user's ID from the request and flush their session
    data.
    """
    # Dispatch the signal before the user is logged out so the receivers have a
    # chance to find out *who* logged out.
    user = getattr(request, "user", None)
    if not getattr(user, "is_authenticated", True):
        user = None
    user_logged_out.send(sender=user.__class__, request=request, user=user)
    request.session.flush()
    if hasattr(request, "user"):
        from django.contrib.auth.models import AnonymousUser
```

Imagen A.4: parte de la función logout

```python
def login(request, user, backend=None):
    """
    Persist a user id and a backend in the request. This way a user doesn't
    have to reauthenticate on every request. Note that data set during
    the anonymous session is retained when the user logs in.
    """
    session_auth_hash = ""
    if user is None:
        user = request.user
    if hasattr(user, "get_session_auth_hash"):
        session_auth_hash = user.get_session_auth_hash()

    if SESSION_KEY in request.session:
        if _get_user_session_key(request) != user.pk or (
            session_auth_hash
            and not constant_time_compare(
                request.session.get(HASH_SESSION_KEY, ""), session_auth_hash
```

Imagen A.5: parte de la función login

```python
def redirect(to, *args, permanent=False, **kwargs):
    """
    Return an HttpResponseRedirect to the appropriate URL for the arguments
    passed.

    The arguments could be:

        * A model: the model's `get_absolute_url()` function will be called.

        * A view name, possibly with arguments: `urls.reverse()` will be used
          to reverse-resolve the name.

        * A URL, which will be used as-is for the redirect location.

    Issues a temporary redirect by default; pass permanent=True to issue a
    permanent redirect.
    """
```

Imagen A.6: función redirect configurada previamente en Django.

```
5   urlpatterns = [
6       path('', views.index_page, name='index-page'),
7       path('login/', views.index_page, name='login'),
8       path('home/', views.home, name='home'),
9       path('buscar/', views.search, name='buscar'),
10
11      path('favourites/', views.getAllFavouritesByUser, name='favoritos'),
12      path('favourites/add/', views.saveFavourite, name='agregar-favorito'),
13      path('favourites/delete/', views.deleteFavourite, name='borrar-favorito'),
14
15      path('exit/', views.exit, name='exit'),
16  ]
```

Imagen A.7: captura de pantalla de las URLpatterns