

INFORME:

GALERÍA DE IMÁGENES DE LA NASA

Materia: Introducción a la programación

Docentes: José Luis Mieres - Lucas Bidart Gauna

Comisión: 14

Ciclo lectivo: 1/2024

Alumnos: Jeremías Vladimir Chazarreta - Martín Edgardo Quiroga - Juan José Benítez - Marcos Nahuel Collis

**Correos electrónicos: chazarreta010720@gmail.com -
martinedgardoquiroga2@gmail.com - juanjosebenitez712@gmail.com -
marcoscollis2020@gmail.com**

Universidad Nacional de General Sarmiento

Introducción

El siguiente informe consiste en la documentación del progreso conseguido, con respecto al trabajo que tiene como finalidad el correcto funcionamiento de una página web utilizando Django Framework, que permite acceder a las imágenes proporcionadas por la API de la NASA. Las metas propuestas consistían en lograr que la información proveniente de la API sea renderizada por el framework (el cual fue trabajado en Visual Studio Code, utilizando Python, Django Framework, Git, html y CSS) en forma de “cards”, mostrando título, imagen y descripción de cada una. También han habido metas adicionales, algunas hemos podido concretar mientras que otras no, por distintas dificultades que se han presentado al momento de tratar de desarrollar dichas funcionalidades extra.

Cambios realizados

- Funcionamiento de la galería

Al iniciar con el trabajo, lo prioritario fue conseguir que la galería de imágenes funcione y muestre las imágenes, títulos y descripciones correspondientes, para conseguirlo en primer lugar lo que hicimos fue en **services_nasa_image_gallery.py** completar lo que faltaba de código, tal como se ve en la imagen:

```
def getAllImages(input=None):
    # obtiene un listado de imágenes desde transport.py y lo guarda en un json_collection.
    # ¡OJO! el parámetro 'input' indica si se debe buscar por un valor introducido en el buscador.
    json_collection = transport.getAllImages(input)

    images = []

    # recorre el listado de objetos JSON, lo transforma en una NASACard y lo agrega en el listado de images. Ayuda: ver mapper.py.
    for card in json_collection:
        nasa_card = mapper.fromRequestIntoNASACard(card)
        images.append(nasa_card)
    return images
```

La función “getAllImages” obtiene un listado de imágenes desde **transport.py** (el cual se comunica con la API de la NASA para obtener imágenes en formato JSON), al cual almacenamos en la variable “**json_collection**”. Luego en “**images**” iniciamos una lista vacía que tomará las imágenes resultantes del ciclo “**for card in json_collection**”, este ciclo recorre cada elemento de “**json_collection**”, y estos elementos son luego convertidos a “**nasa_card**”, una nueva variable la cual invoca al módulo **mapper** y su función “**fromRequestIntoNASACard**” que toma como parámetro a los elementos recorridos en el ciclo anteriormente mencionado. Una vez realizado esto, la función retorna las imágenes de la API en forma de “**nasa_card**”.

Para posibilitar el correcto funcionamiento de la galería fue de igual manera necesario hacer modificaciones en el archivo **views.py**:

```
# auxiliar: retorna 2 listados -> uno de las imágenes de La API y otro de los favoritos del usuario.
def getAllImagesAndFavouriteList(request):
    images = services_nasa_image_gallery.getAllImages()
    favourite_list = services_nasa_image_gallery.getAllFavouritesByUser(request) if request.user.is_authenticated else []
    return images, favourite_list

# función principal de la galería.
def home(request):
    # Llama a la función auxiliar getAllImagesAndFavouriteList() y obtiene 2 listados: uno de las imágenes de La API y otro de favoritos por usuario*.
    # (*) este último, solo si se desarrolló el opcional de favoritos; caso contrario, será un listado vacío [].
    images, favourite_list = getAllImagesAndFavouriteList(request)
    return render(request, 'home.html', {'images': images, 'favourite_list': favourite_list})
```

La función “getAllImagesAndFavouriteList” que toma como parámetro a “request”, llama a la función que habíamos completado en **services_nasa_image_gallery.py**, es decir a “getAllImages”, y toma a las “nasa_card” anteriormente obtenidas y las convierte a la nueva variable “images”. La variable “favourite_list” llama a la función “getAllFavouriteByUser” de **services_nasa_image_gallery.py** para obtener una lista con las imágenes marcadas como favoritas por el usuario, para eso primero verifica si el usuario en cuestión está autenticado, en caso contrario devuelve una lista vacía. Finalmente, devuelve dos listas, por una parte “images” que devuelve todas las imágenes que provienen desde la API de la NASA, y por el otro “favourite_list”, que como dijimos devuelve sólo las imágenes favoritas correspondientes al usuario.

La función “home” que tiene como argumento a “request”, llama a la función **getAllImagesAndFavouriteList** la cual habíamos desarrollado con anterioridad; toma como argumento a “request” y obtiene dos listas, “images” (que como hemos señalado contiene todas las imágenes de la API), y “favourite_list”, que contiene las imágenes favoritas del usuario si está autenticado, o una lista vacía si no lo está. Por último, “return” utiliza la función **render** de Django con el template “home.html”, usando como contexto a las listas provenientes de “images” y “favourite_list”, para mostrar las imágenes y los favoritos del usuario (en caso de haber uno). Con todo lo anterior, fue posible acceder a la galería y sus imágenes.

- **Buscador:**

Para conseguir que el buscador de la galería funcione, fue necesario completar la función “search” en **views.py**:

```
# función utilizada en el buscador.
def search(request):
    images, favourite_list = getAllImagesAndFavouriteList(request)
    search_msg = request.POST.get("query", "")

    if search_msg != "":
        images_filtered = services_nasa_image_gallery.getImagesBySearchInputLike(search_msg)
        return render(request, "home.html", {"images": images_filtered, "favourite_list": favourite_list})
    else:
        return redirect("home")
# si el usuario no ingresó texto alguno, debe refrescar La página; caso contrario, debe filtrar aquellas imágenes que posean el texto de búsqueda.
```

La función “search” tiene como propósito buscar y filtrar imágenes basadas en la entrada del usuario en el buscador. Si el usuario no ingresa ninguna palabra, se lo debe redirigir a la página principal de la galería. Al igual que en la función “home”, “search” llama a “getAllImagesAndFavouriteList” para obtener todas las imágenes disponibles y la lista de favoritos del usuario, y se las asigna respectivamente a “images” y “favourite_list”. Luego la variable “search_msg” obtiene el término de búsqueda ingresado por el usuario a través del método **POST** del formulario; en caso de que el usuario no ingrese nada, “search_msg” será una cadena vacía.

Añadimos el ciclo ‘if search_msg != ””:’ para que en caso de que “search_msg” no esté vacío, proceda a filtrar las imágenes llamando a “getImagesBySearchInputLike” del módulo **services_nasa_image_gallery.py** tomando a “search_msg” como argumento, con el fin de que devuelva una lista que coincida con el término de búsqueda ingresado. Luego return renderiza al template “home.html” con las imágenes anteriormente filtradas guardadas en la variable “images_filtered” y en “favourite_list”. En caso contrario, es decir que “search_msg” sea una cadena vacía, return redirige al usuario a la página “home” mostrando las imágenes sin filtrar.

- **Traducción del buscador:**

Para conseguir que en el caso de que el usuario ingrese una palabra, por ejemplo, en español en el buscador de la galería y obtenga resultados vinculados a esa palabra en inglés, fue necesario primero acceder a Google Cloud, crear una cuenta y habilitar la extensión Cloud Translation API, luego crear una cuenta de servicio con la cual generar una clave que se descargó en formato JSON, esta clave tiene el siguiente nombre: **“dulcet-hulling-427203-s4-92fa254ae49c.json”** (aunque debemos hacer una aclaración importante, y es que al subir esta clave al repositorio, fue inhabilitada por Google Cloud, pero en teoría el método funciona) y la guardamos en una nueva carpeta de nombre **credentials** en el directorio principal:

A su vez en la carpeta **nasa_image_gallery** creamos un nuevo archivo en python al cual llamamos **google_translate.py**, y cuyo código es el siguiente:

```
from google.cloud import translate_v2 as translate
import os

# Establece la ruta al archivo de credenciales JSON
os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = os.path.join(os.path.dirname(__file__), '...', 'credentials', 'dulcet-hulling-427203-s4-28996fa8de55.json')

def translate_text(text, target_language='en'):
    translate_client = translate.Client()
    result = translate_client.translate(text, target_language=target_language)
    return result['translatedText']
```

from google.cloud import translate_v2 as translate: Importa la clase **translate_v2** del paquete **google.cloud** y la asigna un alias denominado **translate**. Esto permite usar las funciones y métodos proporcionados por la API de traducción de Google Cloud. **import os** permite interactuar con el sistema operativo, en este caso para interactuar con variables de entorno.

`os.environ['GOOGLE_APPLICATION_CREDENTIALS']`: Establece la variable de entorno `GOOGLE_APPLICATION_CREDENTIALS`, que es necesaria para autenticar y autorizar el acceso a los servicios de Google Cloud Platform, incluida la API de Traducción; y dentro de esta variable, `os.path.join(os.path.dirname(__file__), '..', 'credentials', 'dulcet-hulling-427203-s4-92fa254ae49c.json')` se encarga de construir la ruta al archivo de credenciales JSON que contiene la información de autenticación para acceder a la API. `os.path.dirname(__file__)` devuelve el directorio actual del archivo de Python en ejecución, y luego se construye la ruta relativa a partir de ahí.

La función `"translate_text"`, que toma como parámetro al texto ingresado por el usuario (`text`), y el idioma al que se desea traducir dicho texto (`target_language='en'`), cuenta con dos variables, la primera `translate_client = translate.Client()`, que crea una instancia del cliente de la API de Traducción de Google Cloud utilizando la clase `Client` del módulo `translate`. Esta instancia se usa para realizar llamadas a la API de traducción. La segunda variable `result = translate_client.translate(text, target_language=target_language)` llama al método `translate` del cliente de traducción para traducir el texto proporcionado (`text`) al idioma especificado (nos interesa en nuestro caso que el texto se traduzca a inglés); esta variable se encarga de devolver un objeto `Translation` que contiene el resultado de la traducción. Por lo que en resumidas cuentas `"translate_text"` se encarga de manejar las traducciones utilizando la API de Google Cloud Translation

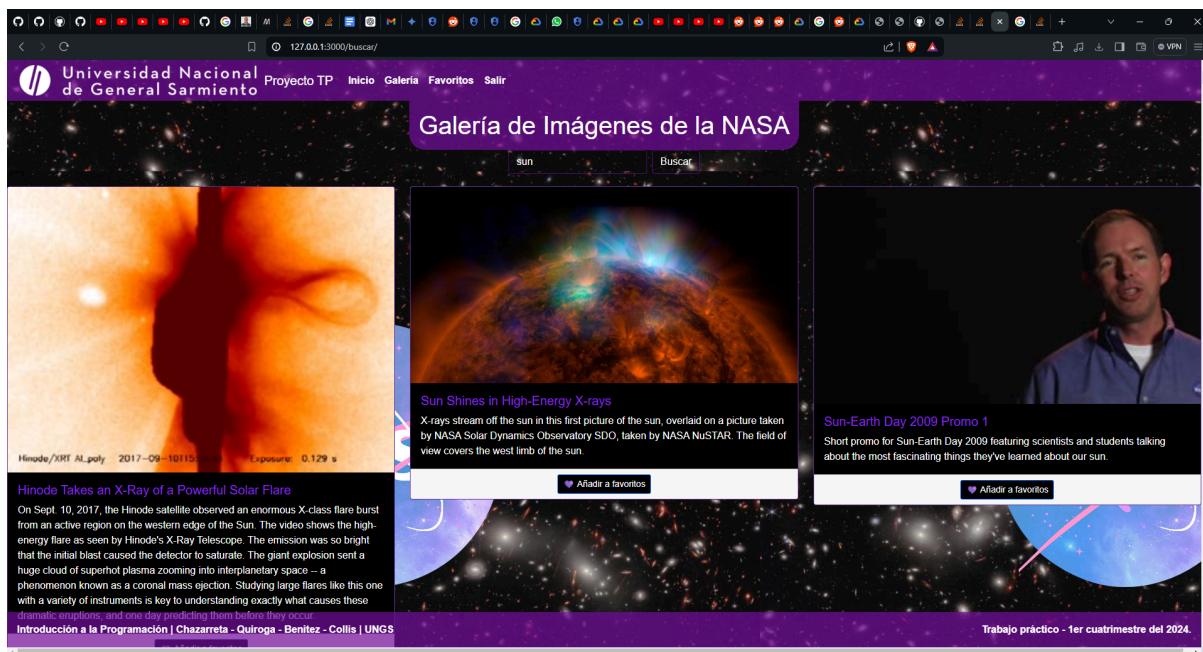
Luego en `views.py` importamos `"translate_text"` de `.google_translate`, y realizamos cambios a la función `search (request)`:

```
if search_msg != "":
    translated_search_msg = translate_text(search_msg, 'en')
    images_filtered = services_nasa_image_gallery.getImagesBySearchInputLike(translated_search_msg)
    return render(request, "home.html", {"images": images_filtered, "favourite_list": favourite_list})
else:
    return redirect("home")
```

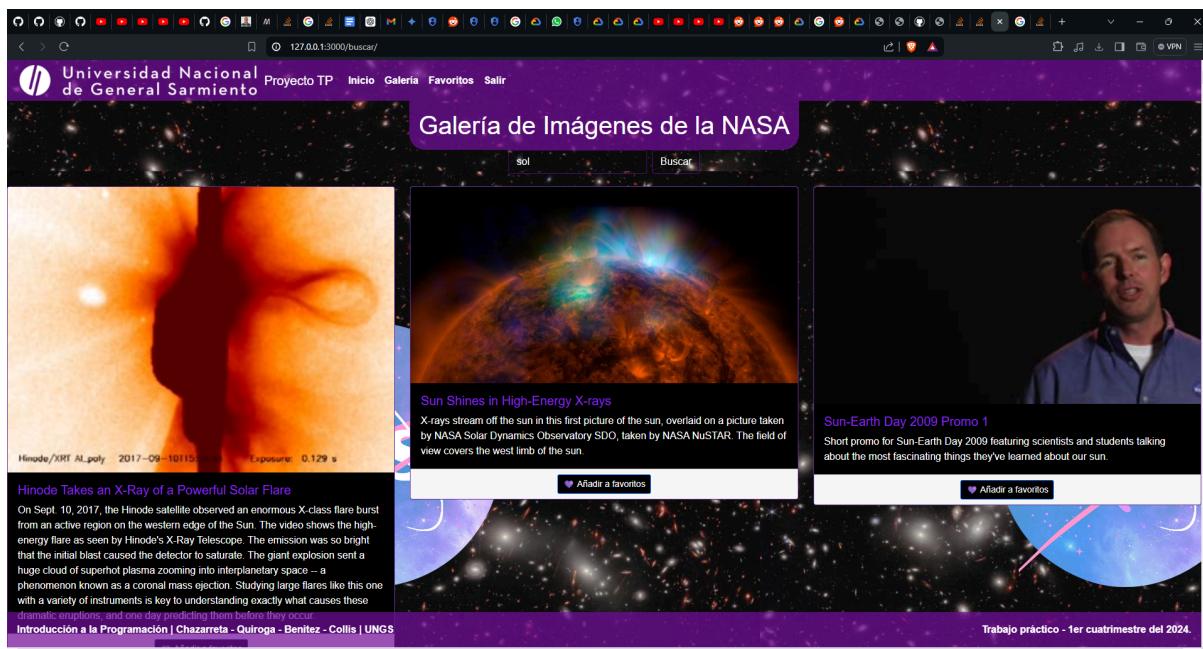
Agregamos una nueva línea de código dentro del condicional `"if search_msg != "":"`, la cual es `translated_search_msg = translate_text(search_msg, 'en')`, que consiste en una variable que llama a la función importada anteriormente, `translate_text`, y se encarga de traducir el contenido de `"search_msg"` a inglés. En la variable `images_filtered` el único cambio que realizamos fue cambiar el parámetro que `getImagesBySearchInputLike` toma, de `"search_msg"` a `"translated_search_msg"`, con tal de que si por ejemplo un usuario buscara una palabra en un idioma ajeno al inglés (por ejemplo, sol), tome en su lugar la palabra ya traducida.

Por último, en la terminal introducimos el siguiente comando para integrar Google Translate en nuestro repositorio: `pip install google-cloud-translate`

Por lo que al buscar una palabra, por ejemplo `"sun"`, en el buscador, devuelve los siguientes resultados:



Y en caso de ingresar la misma palabra en español, es decir “sol”, devuelve los mismos resultados:



- **Inicio/cierre de sesión:**

Esta fue la característica más compleja de implementar, ya que requirió llevar a cabo cambios en varios archivos dentro del repositorio, añadir nuevas funciones y variables, introducir ciertos comandos en la terminal, y múltiples pruebas.

Lo primero en realizar fue introducir en la terminal los siguientes comandos y en el siguiente orden:

1. **python manage.py showmigrations**: Permite verificar qué migraciones están pendientes y cuáles han sido aplicadas, asegurando que la estructura de la base de datos esté actualizada.
2. **python manage.py migrate**: Aplica las migraciones pendientes para actualizar la estructura de la base de datos según los modelos definidos en Django, y asegura que la base de datos tenga las tablas y columnas necesarias para la aplicación.
3. **python manage.py createsuperuser**: Crea un usuario con permisos de superusuario, que tiene acceso total al sitio de administración de Django. Pide al usuario que ingrese un nombre de usuario, correo electrónico (aunque este elemento en particular no tuvo usos prácticos) y contraseña; y crea un registro en la tabla de usuarios con permisos de superusuario. En nuestro caso, el superusuario que creamos se llamó “carlitos”, y tuvo como contraseña “gatogamer44”.

Una vez hecho esto, abrimos la aplicación DB Browser (SQLite) y verificamos que el superusuario haya sido creado exitosamente abriendo la base de datos ya implementada en el repositorio:

The screenshot shows the DB Browser for SQLite interface. The title bar reads "DB Browser for SQLite - C:\Users\Vladimir\Desktop\ip-public-repo-main\ip-public-repo-main\db.sqlite3". The menu bar includes File, Edit, View, Tools, and Help. The toolbar has buttons for New Database, Open Database, Write Changes, Revert Changes, and Open Project. Below the toolbar are tabs for Database Structure, Browse Data, Edit Pragmas, and Execute SQL. The "Table:" dropdown is set to "auth_user". The main area displays a table with four columns: password, last_login, is_superuser, and username. There are two rows of data:

	password	last_login	is_superuser	username
1	\$600000\$OYNC8t10bvVvagEbHu5...	2024-02-17 22:48:04.011700	1	admin
2	\$600000\$GEedPyGQnIYIUbzdm9U...	2024-06-21 03:44:04.915486	1	carlitos

Luego, ejecutamos el servidor y lo abrimos desde el navegador con el URL “127.0.0.1:3000/admin/”, esto nos lleva al sitio de administración de Django en donde deberemos iniciar sesión con nuestro recientemente creado superusuario “carlitos”, una vez dentro debe aparecer lo siguiente:

Django administration

Home > Authentication and Authorization > Users > Add user

Start typing to filter...

AUTHENTICATION AND AUTHORIZATION

Groups	+ Add
Users	+ Add

Add user

First, enter a username and password. Then, you'll be able to edit more user options.

Username:

Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

Your password can't be too similar to your other personal information.
Your password must contain at least 8 characters.
Your password can't be a commonly used password.
Your password can't be entirely numeric.

Password confirmation:

Enter the same password as before, for verification.

«

SAVE Save and add another Save and continue editing

Desde aquí podemos añadir nuevos usuarios, creamos uno nuevo llamado “pedroprograma” cuya contraseña es “panamatropical”, una vez hecho esto verificamos que se haya logrado nuevamente en DB Browser:

DB Browser for SQLite - C:\Users\Vladimir\Desktop\ip-public-repo-main\ip-public-repo-main\db.sqlite3

File Edit View Tools Help

New Database Open Database Write Changes Revert Changes Open Project

Database Structure Browse Data Edit Pragmas Execute SQL

Table: auth_user Filter in any column

password	last_login	is_superuser	username
1 \$OYNC8t10bvVvagEbHu5...	2024-02-17 22:48:04.011700	1	admin
2 \$GEedPyGQnIYIUbzdm9U...	2024-06-21 03:44:04.915486	1	carlitos
3 \$gSDeb3ahTKlpHxKet9MJ...	2024-06-21 02:54:16.839495	0	pedroprograma

Luego, en `views.py` primero importamos lo siguiente:

```
from django.http import HttpResponseRedirect
from .layers.services import services_nasa_image_gallery
from django.contrib.auth import authenticate, login, logout
```

HttpResponse: Permite devolver respuestas HTTP sencillas.

services_nasa_image_gallery: Importa funciones de servicios relacionadas con la galería de imágenes

authenticate, login, logout: Importaciones del sistema de autenticación de Django para manejar la autenticación del usuario, el inicio y cierre de sesión.

Creamos una función nueva llamada “login_page” que maneja la autenticación de los usuarios, permitiéndoles iniciar sesión en la página:

```
def login_page(request):
    if request.method == 'POST':
        username = request.POST['username']
        password = request.POST['password']
        user = authenticate(request, username=username, password=password)
        if user is not None:
            login(request, user)
            return redirect('home')
        else:
            return HttpResponseRedirect('Usuario o contraseña incorrectos.')
    return render(request, 'login.html')
```

El primer condicional verifica si la solicitud es de tipo **POST**, lo que indica que el usuario ha enviado el formulario de inicio de sesión. En las variables “username” y “password” se extraen las credenciales del usuario (nombre de usuario y contraseña) desde los datos enviados en el formulario **POST**. En la variable “user” se utiliza la función “authenticate” de Django para verificar las credenciales; y si las credenciales son correctas, “authenticate” devuelve el objeto “user”. En caso contrario devuelve “none”.

En la segunda condicional, si el usuario ingresado es correctamente autenticado llama a **login** para iniciar la sesión del usuario tomando como parámetros a “user” y “request”; y redirige el usuario a la galería (“home”). Pero si el usuario es “none” (es decir, si las credenciales son incorrectas), se devuelve una respuesta HTTP con el mensaje “Usuario o contraseña incorrectos”. Finalmente, en la última línea de código si la solicitud no es de tipo **POST**, se renderiza el formulario de inicio de sesión (“login.html”).

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

El “@login_required” que se encontraba al final de **views.py** poseía la función exit, que en teoría debía asegurar que solo los usuarios autenticados puedan cerrar sesión, para completarla llamamos a la función “logout” de Django para poder cumplir con este propósito. Una vez hecho esto redirige al usuario a la página de inicio (es decir, la de bienvenida).

En **settings.py**, en las variables que se encuentran en la parte inferior del código, realizamos los siguientes cambios:

```
LOGIN_URL = 'login'  
LOGIN_REDIRECT_URL = 'home'  
LOGOUT_REDIRECT_URL = 'index-page'
```

En primer lugar, agregamos la variable “LOGIN_URL”, la cual se encarga de redirigir a un usuario que no está autenticado a la página de inicio de sesión con la URL “login” cuando intenta acceder a una vista protegida por el decorador “@login_required”. En segundo lugar, cambiamos la URL a la cual se redirige al usuario una vez que inicia sesión exitosamente, ahora se lo envía a la galería de imágenes (“home”), anteriormente se lo enviaba a la página de inicio (“index-page”).

Sin embargo, aún quedan pendientes cambios por realizar en **login.html** y **urls.py** para asegurar que el inicio y cierre de sesión funcionen adecuadamente, empezando por **urls.py**:

```
urlpatterns = [  
    path('', views.index_page, name='index-page'),  
    path('login/', views.login_page, name='login'),  
    path('logout/', views.logout_page, name='logout'),  
]
```

Antiguamente, **path('login/', views.index_page, name='login')** redirigía al usuario a la página de inicio, lo cual es incorrecto porque debería estar asociada con una vista que maneje el proceso de inicio de sesión, por lo que ahora asociamos la URL “login/” con la vista “login_page”.

En **login.html** decidimos mover dicho archivo a la carpeta donde se encontraba el resto de archivos html (la carpeta templates), ya que de otra manera no funcionaba, dicho esto, hemos realizado los siguientes cambios en el código:

```

{% extends 'header.html' %} {% block content %}


<form action="{% url 'login' %}" method="POST" style="display: inline-block;">
    {% csrf_token %}
    <h2 class="text-center">Inicio de sesión</h2>
    <div class="form-group" style="margin-bottom: 5%;">
        <input type="text" name="username" id="username" class="form-control" placeholder="Usuario" required="required">
    </div>
    <div class="form-group" style="margin-bottom: 5%;">
        <input type="password" name="password" id="password" class="form-control" placeholder="Contraseña" required="required">
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-primary btn-block">Ingresar</button>
    </div>
</form>


{% endblock %}

```

Primero, borramos `{% if request.user.is_authenticated %}` que se encontraba en la primera línea, ya que exigía que el usuario primero esté autenticado antes de poder acceder al inicio de sesión, lo cual es una contradicción, ya que no puede estar autenticado sin haber iniciado sesión antes. En consecuencia, también eliminamos las tres líneas de código anteriores a `{% endblock %}`, ya que en caso de que en efecto el usuario no se encontrara autenticado, mostraban el mensaje “por favor, inicia sesión de nuevo”, estas líneas eran:

```

{% else %}
<p>Por favor, inicia sesión.</p>
{% endif %}

```

Por último, borramos la duplicación innecesaria de `action="{% url 'login' %}"` en la etiqueta `<h2>`, ya que dicho fragmento de código ya se utiliza en la etiqueta `<form>` para especificar la URL a la que se enviará el formulario cuando se envíe. Por lo que colocarlo en una etiqueta `<h2>` es redundante y no cumple ninguna función.

Por último, aún queda por configurar la funcionalidad del botón Iniciar sesión, que está en el encabezado de la página web, para ello hemos realizado múltiples cambios en “header.html”:

```

{% endif %}
<li class="nav-item">
    {% if request.user.is_authenticated %}
        <form action="{% url 'exit' %}" method="post">
            {% csrf_token %}
            <button type="submit" class="nav-link" style="background-color: transparent; color: black; border: none;"><strong>Salir</strong></button>
        </form>
    {% else %}
        <a class="nav-link" href="{% url 'login' %}" style="color: black;"><strong>Iniciar sesión</strong></a>
    {% endif %}
</li>

```

`{% if request.user.is_authenticated %}` verifica que el usuario esté “autenticado” o en otras palabras que haya iniciado sesión exitosamente, en ese caso se mostrará el botón “salir” en el encabezado, en caso contrario se muestra un formulario de inicio de sesión.

Creamos un formulario para el botón salir que, al ser enviado, realiza una solicitud **POST** a la URL definida por `{% url 'exit' %}`. Añadimos un token de seguridad `{% csrf_token %}` que Django utiliza para proteger el formulario contra ataques CSRF, y convertimos a salir en

un botón de tipo “submit” que actúa como un enlace estilizado. Al hacer clic en este botón, se envía el formulario. Si el usuario no está autenticado, se muestra un enlace que lleva a la página de inicio de sesión, es decir `{% url 'login' %}`. Y por último, añadimos `{% endif %}` al final para cerrar la estructura condicional.

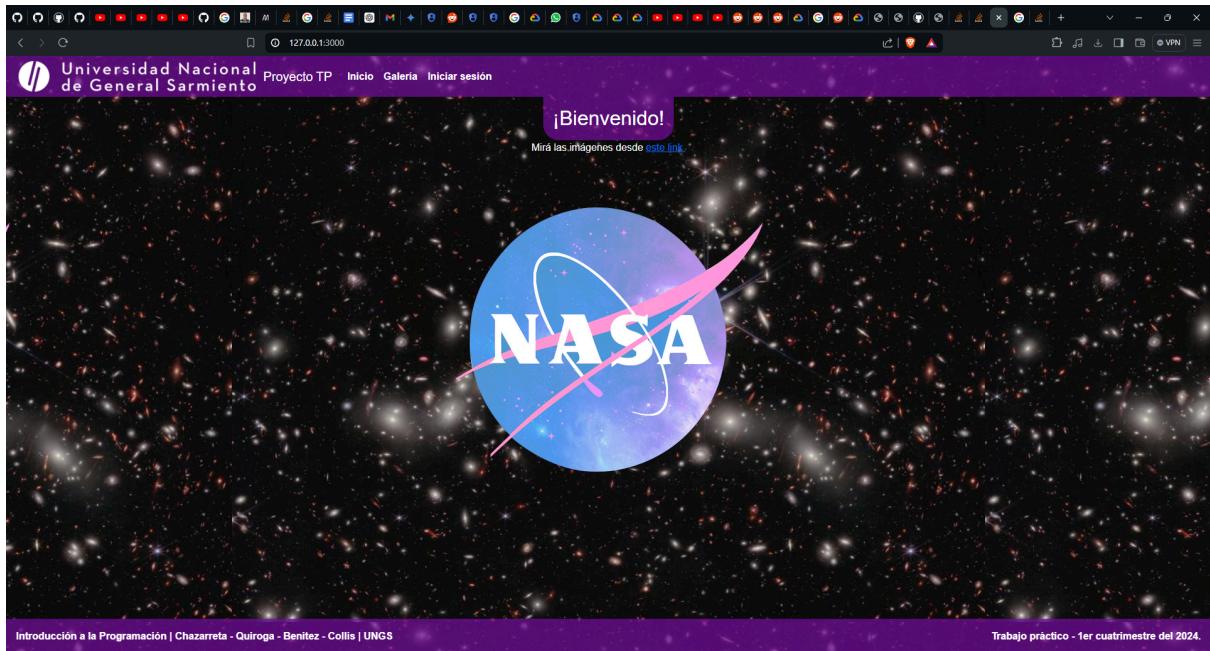
Con todos los cambios anteriormente explayados, fue posible obtener como resultado el correcto funcionamiento tanto del inicio de sesión como del cierre de sesión, y una vez dentro de la galería y con un usuario autenticado, debajo de la descripción de cada imagen, se encuentra disponible el botón para marcar como favorito a dicha imagen, sin embargo no hemos conseguido que funcione.

Conclusión

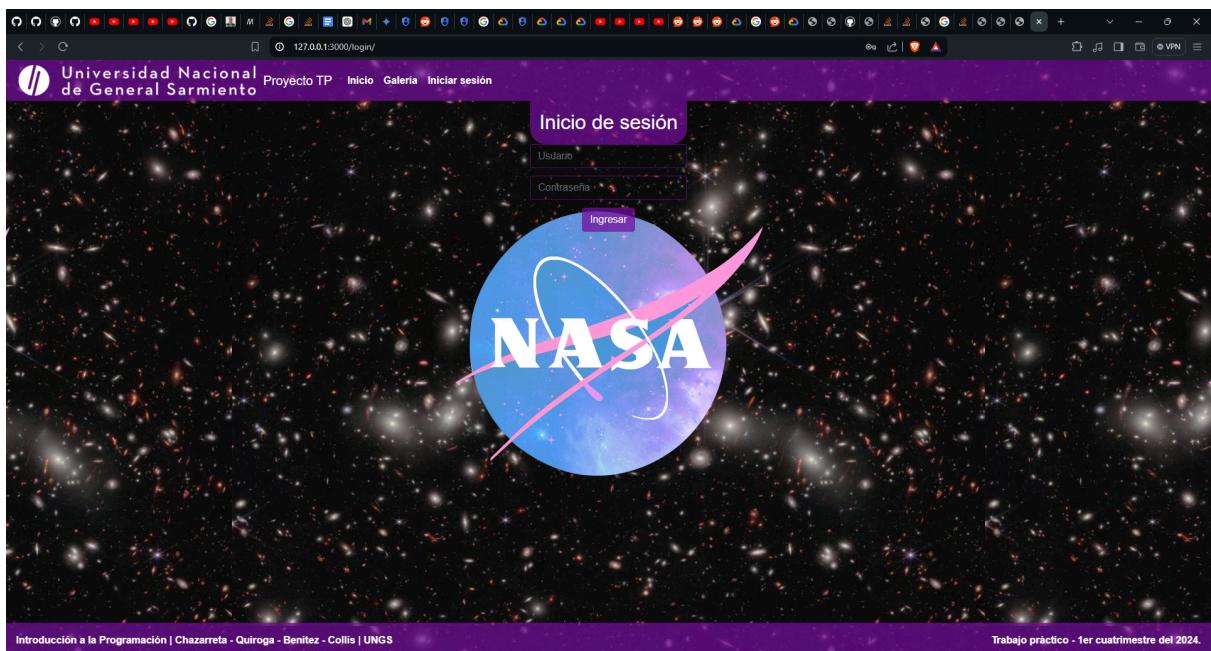
Hemos conseguido que los propósitos primordiales de la página funcionen adecuadamente, es decir la galería de imágenes de la NASA y el buscador (aunque debemos advertir que por motivos que no hemos logrado descifrar, el término “Station” en particular, al ser ingresado en el buscador no funciona). También hemos logrado cumplir con los objetivos adicionales de la traducción automática de palabras ingresadas en español, a inglés, para el correcto funcionamiento del buscador, el inicio y cierre de sesión, y renovamos la interfaz gráfica de la página, incluyendo un encabezado que baja a la par que el usuario se desplaza hacia abajo o arriba de la misma. Aunque no hemos podido cumplir con los objetivos opcionales de favoritos, la paginación, la incorporación de comentarios, el registro de nuevos usuarios, el loading spinner o la internacionalización de la página en su totalidad.

Resultado final

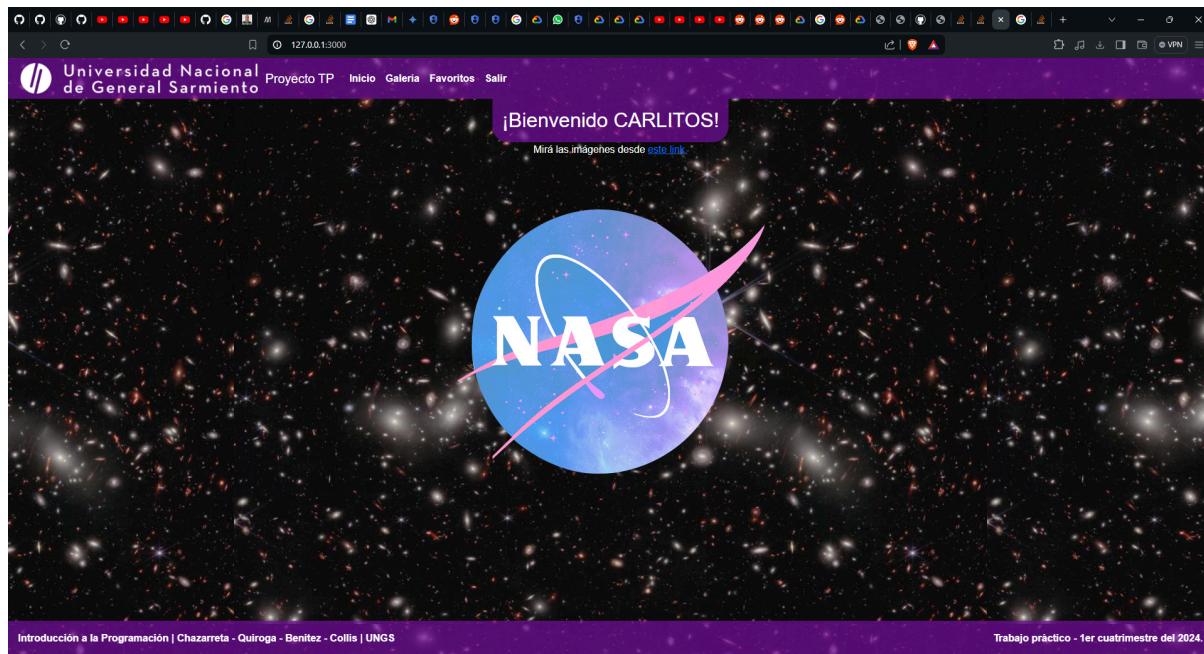
- Página inicial sin iniciar sesión:



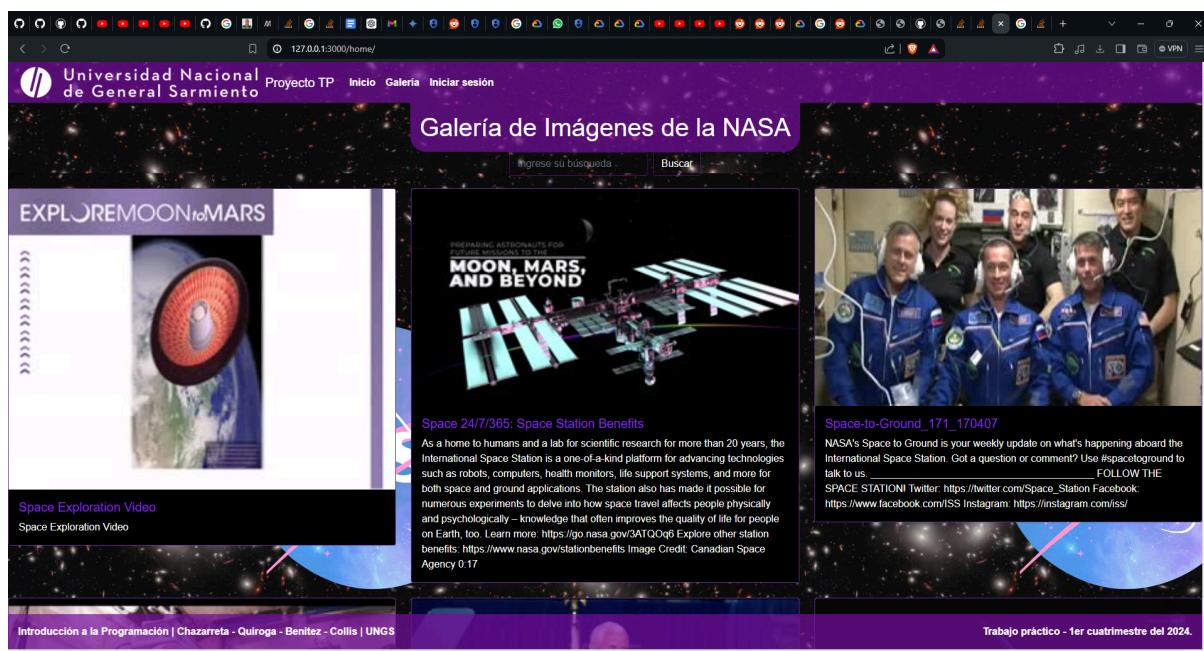
- Inicio de sesión:



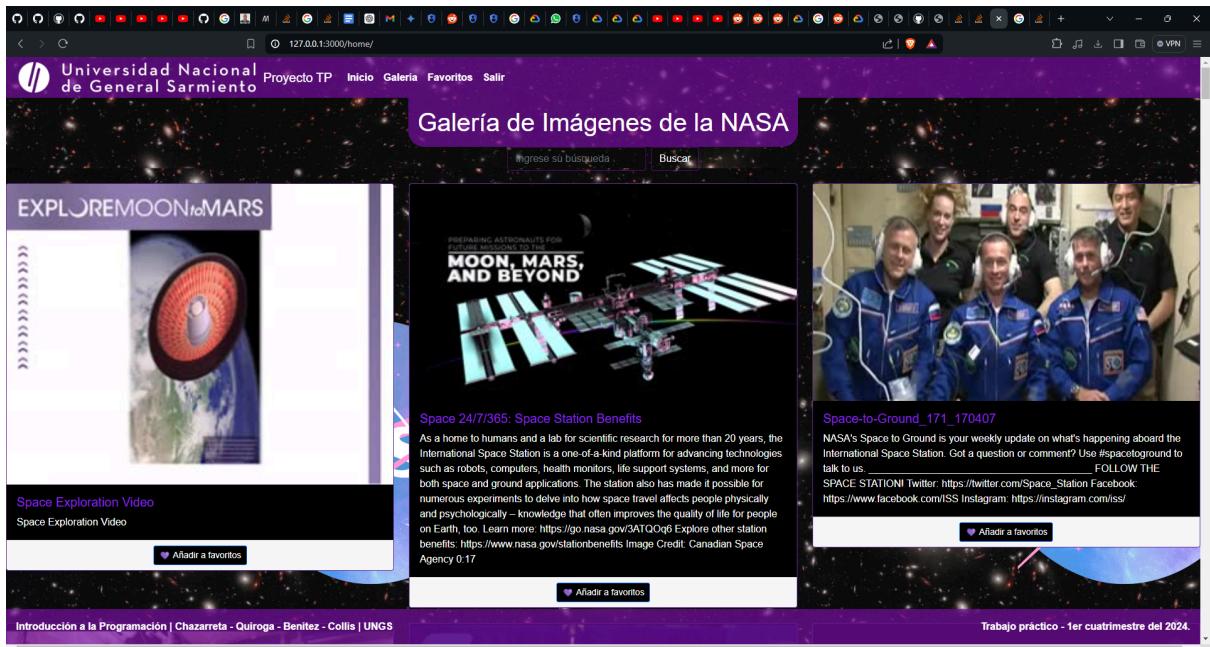
- Página inicial con usuario “carlitos” cargado:



- Galería de imágenes sin sesión iniciada:



- Galería de imágenes con sesión iniciada:



- Búsqueda en la galería, ejemplo buscando resultados relacionados a Júpiter:

