Base de Datos

Trabajo Práctico

Nicolás Khalil Chaia
110768

Gonzalo Nicolas Crudo
110816

Matías Besmedrisnik
110487

Ezequiel Martín Aragón
110643

Maryuris Artiles
110188

---

1. Explicación de las Tecnologías.

Se opto por usar la herramienta Django, ya que es un framework web gratuito y de código abierto escrito en python (razón principal por la que se codeo en este lenguaje).

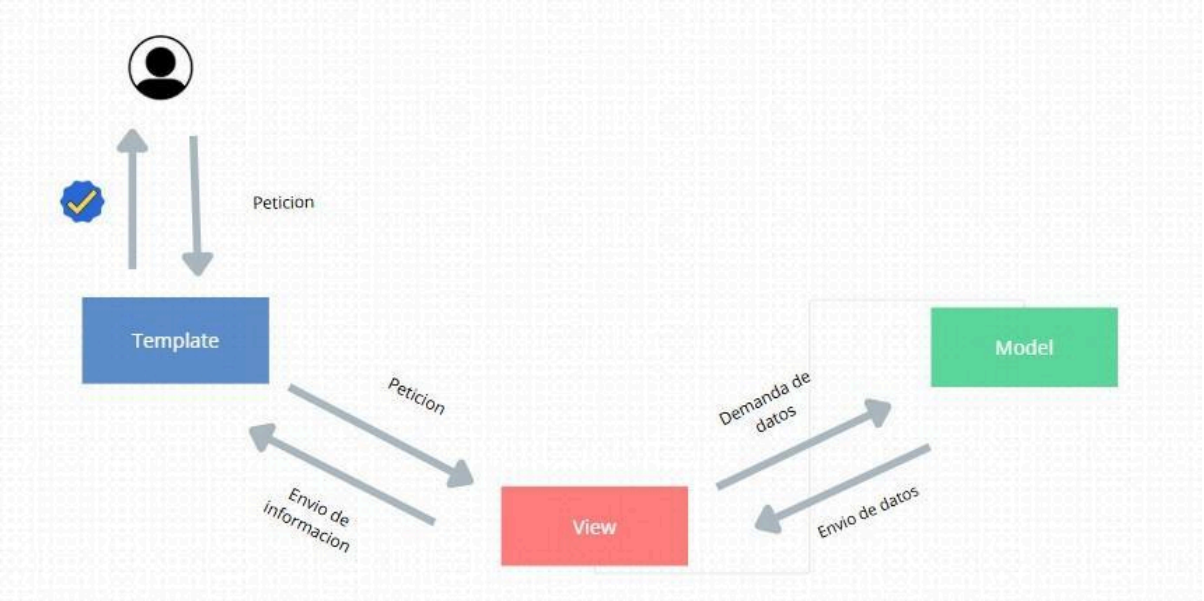
¿Pero por qué elegimos Django y no otro framework? Lo elegimos porque nos permite crear sitios web de forma rápida y sencilla. Las tareas que son repetitivas, pesadas y comunes al momento de crear diferentes sitios web, con Django, la realización de dichas tareas se facilita.

Elección de bases:

- La base de datos relacionales que se eligió para nuestra aplicación fue Sqlite3, ya que es la Django usa/crea por defecto, si bien Django la crea automáticamente, aun seguimos teniendo la obligación de crear las tablas que están dentro de esa base datos, Y en Django esto lo conseguimos utilizando una clase, llamada model. También otra de las razones por la que decidimos usarla es que está optimizada para manejar datos de tamaño moderado con un rendimiento excelente, perfecto para aplicaciones que manejan información sobre películas, como sus nombres, géneros, directores y reparto.

- La base de datos no relacionales que se le eligió fue Firebase Firestore, ya que una de sus ventajas es la sincronización en tiempo real (significa que cualquier cambio en los datos se refleja instantáneamente en todas las aplicaciones conectadas). Esto nos beneficia porque tenemos la intención de que nuestra aplicación de reseñas de películas, con datos en tiempo real pueda ofrecer una experiencia muy enriquecedora para los usuarios. Usar Firebase Firestore nos da entonces, la ventaja de que los usuarios puedan ver las reseñas y calificaciones de otros usuarios al instante, lo que hace que la interacción sea más dinámica.

2. Diagrama de arquitectura.



Dividimos nuestra aplicación en tres grandes módulos, Temple, View y Model.

El model es el que se encarga de gestionar los datos, normalmente de obtener información de una base de datos.

El template es el módulo encargado de mostrar la información en el usuario, lo que el usuario ve y con lo que interactúa.

El view es el encargado de gestionar todas las comunicaciones entre el template y el modelo.

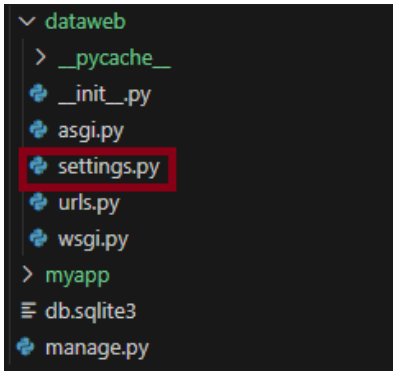
Que ventajas nos da dividir nuestra aplicación en tres grandes módulos:

Hace la aplicación más funcional, mantenible y escalable (si en un futuro necesitamos agregarle más funciones a la aplicación lo vamos a poder hacer de una forma más fácil)

3. Configuración y conexión a las bases de datos.

Sqlite3.

Django nos conecta a una base de datos utilizando su configuración en el archivo settings.py. Este archivo contiene una sección llamada DATABASES, donde definimos los detalles de la conexión a las bases de datos. Esto lo usamos para conectarnos a la base de datos sqlite3




```python
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': BASE_DIR / 'db.sqlite3',
    }
}
```

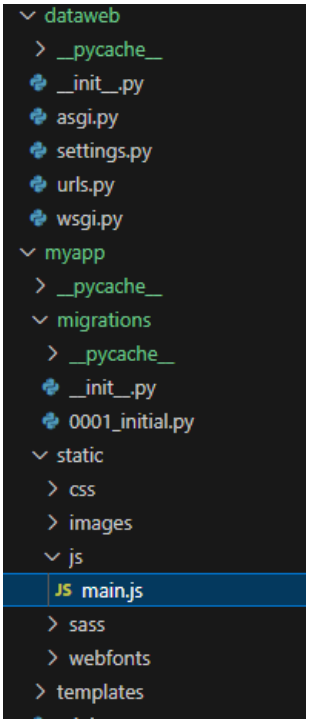
Firebase.

La configuración para conectarnos a la base de datos Firebase se encuentra en un archivo distinto .js, por lo tanto, la conexión a esta base de datos se estableció de forma programática:

a) Configuración del proyecto Firebase: proporcionan las claves de configuración en el código (en el objeto firebaseConfig). Estas claves se generan automáticamente cuando configuras un proyecto en la consola de Firebase.

b) Inicialización de Firebase: con estas claves, se usa el método `firebase.initializeApp(firebaseConfig)` para inicializar la aplicación y conectar el cliente a nuestro proyecto Firebase.

c) Referencia a Firestore: después de inicializar Firebase, obtenemos una referencia a la base de datos de Firestore, esto permite acceder y manipular las colecciones y documentos de Firestore.




```javascript
var firebaseConfig = {
    apiKey: "AIzaSyCCDmOpf0SJfMDfTQ6l3JVPQsowipHRfFI",
    authDomain: "base-de-datos-6bc22.firebaseapp.com",
    projectId: "base-de-datos-6bc22",
    storageBucket: "base-de-datos-6bc22.firebasestorage.app",
    messagingSenderId: "851585268399",
    appId: "1:851585268399:web:e30a568f25e986fedef25a",
    measurementId: "G-1QBL9TJE8N"
};
firebase.initializeApp(firebaseConfig);
const db = firebase.firestore();
```

4. Descripción de Funcionalidades CRUD

Sqlite3.

Como se mencionó anteriormente, gracias a Django podemos hacer uso de la clase model. La clase model tiene en su interior todas las herramientas, es decir funciones, propiedades, para poder manejar bases de datos. Entonces con ella podremos crear o eliminar tablas, modificar o eliminar campos, especificar el tipo de dato de un campo…Todo eso viene dentro de la clase model.

Para trabajar con la base de datos Sqlite3 usamos el archivo model.py. Dentro de este archivo crearemos una clase por cada tabla que necesito o quiero que tenga mi base de datos. Lo bueno de esto es que no tenemos que insertar ni una sola instrucción de código de SQL. Porque todo esto lo hace por detrás Django (otra de las grandes ventajas de usar este framework, es que el código SQL lo crea por nosotros).

Observación: Igual consideramos que para trabajar con un framework como este, es imprescindible conocer el lenguaje SQL (un desarrollador web no puede estar trabajando con un framework y no conocer el lenguaje SQL).

Entonces creamos nuestra Class Movie, la cual va a recibir un models.Model, para poder trabajar con la clase Model. Aquí empezamos a crear los campos que queremos que tenga nuestra tabla Movie, los tipos de datos que va almacenar en su interior en cada campo.

Aclaración: Charfield es la instrucción que nos permite almacenar caracteres, con ma_length le estamos diciendo que se va a poder introducir hasta 30 caracteres.

```
class Movie(models.Model):
    title = models.CharField(max_length=100)
    genre = models.CharField(max_length=100)
    director = models.CharField(max_length=100)
    def __str__(self):
        return self.title
```

Ya explicado de qué forma vamos a representar nuestra tabla, a continuación, explicamos la descripción de las funcionalidades CRUD:

Inicialización de Formularios y Variables:

- movie_form: Instancia de MovieForm para añadir nuevas películas.
- search_form: Instancia de MovieSearchForm para buscar películas.
- movies: Inicializada como None, se actualizará con los resultados de búsqueda.
- movie_list: Contiene todas las películas en la base de datos.

Estas variables inicializan los formularios y cargan todas las películas para mostrarlas en la página.

Manejo de solicitudes POST.

El bloque if request.method == 'POST' maneja las solicitudes de POST, lo que indica que se ha enviado un formulario. Es decir, detecta si el usuario ha enviado datos desde la página.

Añadir película:

Comprueba si el botón de añadir (`add_movie`) fue presionado.

Valida los datos del formulario `MovieForm` y, si son válidos, guarda la nueva película.

```
if 'add_movie' in request.POST:
    movie_form = MovieForm(request.POST)
    if movie_form.is_valid():
        movie_form.save()
        return redirect('home')  # Refresca la página tras añadir la película
```

Buscar película:

Filtra las películas según el criterio especificado en el formulario.

Los resultados se pasan al contexto para mostrarlos en la plantilla.

```
elif 'search_movie' in request.POST:
    search_form = MovieSearchForm(request.POST)
    if search_form.is_valid():
        search_by = search_form.cleaned_data['search_by']
        query = search_form.cleaned_data['query']
        if search_by == 'director':
            movies = Movie.objects.filter(director__icontains=query)
        elif search_by == 'genre':
            movies = Movie.objects.filter(genre__icontains=query)
        else:
            movies = Movie.objects.filter(title__icontains=query)
        return render(request, 'home.html', {
            'movie_form': movie_form,
            'search_form': search_form,
            'movie_list': movie_list,
            'movies': movies,
        })
```

Eliminar película:

Elimina las películas basándose en el criterio seleccionado.

```
elif 'delete_movies_by' in request.POST:
    delete_by = request.POST.get('delete_by')
    condition = request.POST.get('delete_condition')
    if delete_by == 'title':
        Movie.objects.filter(title__icontains=condition).delete()
    elif delete_by == 'director':
        Movie.objects.filter(director__icontains=condition).delete()
    elif delete_by == 'genre':
        Movie.objects.filter(genre__icontains=condition).delete()
    return redirect('home')
```

Actualizar película:

Actualiza las películas según el criterio seleccionado.

```
elif 'update_movie_by' in request.POST:
    update_by = request.POST.get('update_by')
    condition = request.POST.get('update_condition')
    new_value = request.POST.get('new_value')
    if update_by == 'director':
        Movie.objects.filter(director__icontains=condition).update(director=new_value)
    elif update_by == 'genre':
        Movie.objects.filter(genre__icontains=condition).update(genre=new_value)
    return redirect('home')
```

Firebase.

Añadir reseña (Create):

Se escucha el evento `submit` del formulario con el id `add-review-form`. Al enviar el formulario:

- Se obtienen los valores de los campos `film-name` y `review`. Si ambos campos están completos: se añade la reseña a la colección `reviews`en Firestore. Se muestra una alerta de éxito y se recarga la lista de reseñas llamando a `loadReviews()`.

Si algún campo está vacío, se muestra una alerta pidiendo que se completen los campos.

```
document.getElementById('add-review-form').addEventListener('submit', async (e) => {
    e.preventDefault();

    const filmName = document.getElementById('film-name').value.trim();
    const review = document.getElementById('review').value.trim();

    if (filmName && review) {
        try {
            await db.collection('reviews').add({ filmName, review });
            alert('Review añadida exitosamente');
            document.getElementById('add-review-form').reset();
            loadReviews(); // Recargar la tabla después de añadir una reseña
        } catch (error) {
            console.error('Error al añadir la reseña:', error);
            alert('Hubo un problema al añadir la reseña.');
        }
    } else {
        alert('Por favor completa todos los campos.');
    }
});
```

Buscar reseñas:

Se escucha el evento `submit` del formulario con el id `searchReviewForm`.

Al enviar el formulario, se obtiene el valor ingresado en el campo de búsqueda (`reviewInput`).

Si el valor de búsqueda no está vacío, se consulta la colección `reviews` y se filtran las reseñas cuyo nombre de película contenga el término de búsqueda (sin distinguir mayúsculas/minúsculas).

Se muestran los resultados encontrados en la lista `searchReviewResults`.

Si no se encuentran coincidencias, se muestra un mensaje indicando que no se encontraron resultados.

Si el campo de búsqueda está vacío, se muestra una alerta pidiendo ingresar un término de búsqueda.

```
document.getElementById('searchReviewForm').addEventListener('submit', async (e) => {
    e.preventDefault();

    const searchValue = document.getElementById('reviewInput').value.trim().toLowerCase();
    const resultsContainer = document.getElementById('searchReviewResults');
    resultsContainer.innerHTML = '';

    if (searchValue) {
        try {
            const snapshot = await db.collection('reviews').get();
            let results = '';

            snapshot.forEach((doc) => {
                const data = doc.data();
                if (data.filmName.toLowerCase().includes(searchValue)) {
                    results += `<li>${data.filmName}: ${data.review}</li>`;
                }
            });

            resultsContainer.innerHTML = results || '<li>No se encontraron resultados.</li>';
        } catch (error) {
            console.error('Error al buscar reseñas:', error);
            alert('Hubo un problema al buscar las reseñas.');
        }
    } else {
        alert('Por favor ingresa un término de búsqueda.');
    }
});
```

Cargar reseñas y eliminar reseñas:

La función `loadReviews` recupera todas las reseñas de la colección `reviews` en Firestore.

Se crean filas en la tabla para cada reseña, mostrando el nombre de la película y la reseña.

Se añade un botón de eliminación para cada reseña. Al hacer clic en el botón de eliminación, la reseña correspondiente se elimina de la base de datos y la tabla se recarga.

Cada botón de eliminar `delete-btn` está asociado al documento en Firestore mediante su id.

Al hacer click al botón se elimina la reseña correspondiente de Firestore, se recargan las reseñas llamando nuevamente a `loadreviews`.

```
async function loadReviews() {
    const tableBody = document.getElementById('reviewTableBody');
    tableBody.innerHTML = '';
    try {
        const snapshot = await db.collection('reviews').get();
        snapshot.forEach((doc) => {
            const data = doc.data();
            const row = document.createElement('tr');
            row.innerHTML = `
                <td>${data.filmName}</td>
                <td>${data.review}</td>
                <td>
                    <button class="delete-btn" data-id="${doc.id}">Eliminar</button>
                </td>
            `;
            tableBody.appendChild(row);
        });

        // Añadir eventos a los botones de eliminar
        document.querySelectorAll('.delete-btn').forEach((button) => {
            button.addEventListener('click', async (e) => {
                const id = e.target.getAttribute('data-id');
                try {
                    await db.collection('reviews').doc(id).delete();
                    alert('Reseña eliminada exitosamente');
                    loadReviews(); // Recargar la tabla después de eliminar
                } catch (error) {
                    console.error('Error al eliminar la reseña:', error);
                    alert('Hubo un problema al eliminar la reseña.');
                }
            });
        });
    } catch (error) {
        console.error('Error al cargar las reseñas:', error);
        alert('Hubo un problema al cargar las reseñas.');
```

5. Comparación entre Bases de Datos Relacionales y NoSQL

Base de datos relacional SQLite3.

Ventajas

- Se tiene un desempeño rápido para las consultas, ya que al ser una base de datos relacional es más rápida y eficiente para consultas estructuradas y transacciones complejas.
- Como SQLite está integrado fácilmente con Django, esto nos simplificó la configuración y el uso de la base de datos en nuestra aplicación.

Desventajas:

- SQLite3 es ideal para aplicaciones pequeñas o medianas, pero no está diseñado para aplicaciones de gran escala con miles de usuarios concurrentes, es decir nuestra página aplicación no podría soportar una gran cantidad de usuarios.

Base de datos relacional noSQL Firebase.

Ventajas

- Una base de datos NoSQL es más adecuada para aplicaciones de gran escala con muchas escrituras concurrentes, es decir si nuestra aplicación solo fuera de reseñas, podríamos usar solo noSQL.
- Permite la sincronización automática de datos en tiempo real entre la base de datos y los usuarios.

Desventajas

- Se tiene un costo elevado con gran uso, el costo de operaciones de lectura/escritura puede ser alto en aplicaciones con mucho tráfico.

6. Demostración del funcionamiento de la aplicación.

El siguiente lleva a un video donde se muestra el funcionamiento de nuestra aplicación:

https://drive.google.com/file/d/1i72Frnn8ccn1Ylh_gQ3XGf08VHz8WIjT/view?usp=drive_link