WatchThis

**Índice**

[**Patrones de Diseño** 1](#_Toc63180662)

[ **Factory Method** 1](#_Toc63180663)

[ **Decorator** 4](#_Toc63180664)

[ **Observer** 5](#_Toc63180665)

# **Patrones de Diseño**

Para el desarrollo del proyecto se ha llevado a cabo la implementación de tres patrones de diseño, los cuales se han aplicado para la parte de Backend ya que se encargaba de la administración y creación de los modelos de datos, así como su gestión para los propios usuarios de la aplicación:

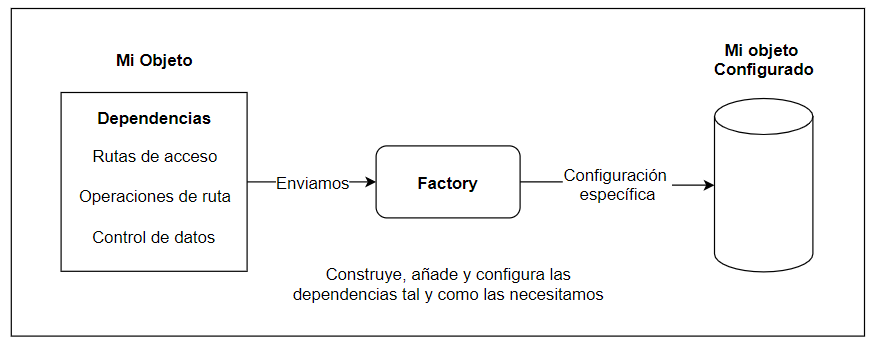
## **Factory Method**

Flask, la librería implementada para el desarrollo del backend en Python, cuenta con un patrón llamado “Factory” el cual puede ser útil para crear las instancias de “app” desde Flask o manejar sus “plugins” en casos necesarios (enrutamientos, decoradores…).

El patrón “Factory” o “Application Factory” nos ofrece una serie de ventajas para el propio desarrollo de la aplicación, así como una posible escalabilidad en caso de extender y aumentar la complejidad del proyecto:

* Detiene / Evita las importaciones circulares.
* Mejora y facilita la configuración de la aplicación, así como sus pruebas y los diferentes entornos.
* Permite un mejor entendimiento del proyecto, teniendo una mejor organización de clases y componentes, así como legibilidad del código.

Con este patrón hemos aplicado el termino “Factory” al cual le damos una serie de datos o dependencias, obteniendo un producto totalmente configurado como salida:



*Diagrama de Factoría*

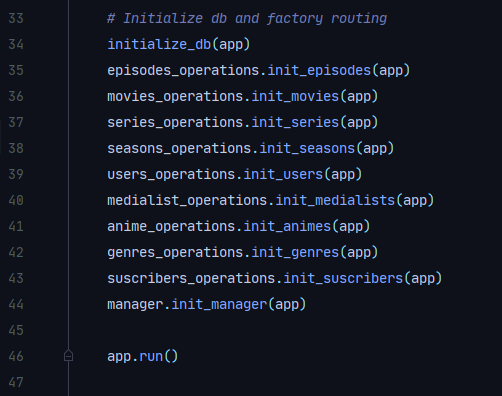
En este caso nuestro objeto va a ser “app”, la aplicación de Flask la cual vamos a necesitar configurar para adaptar las dependencias acordes a los modelos de datos creados, así como las rutas de la página, contando con las posibles operaciones CRUD.

Para ello, con el fin de poder configurar el objeto será necesario importarlo en cada una de las clases que quieran modificar y tratar con él, en este caso, con el fin de evitar la importación cíclica o circular vamos a crear unas factorías en las cuales vamos a pasar nuestro objeto como parámetro:

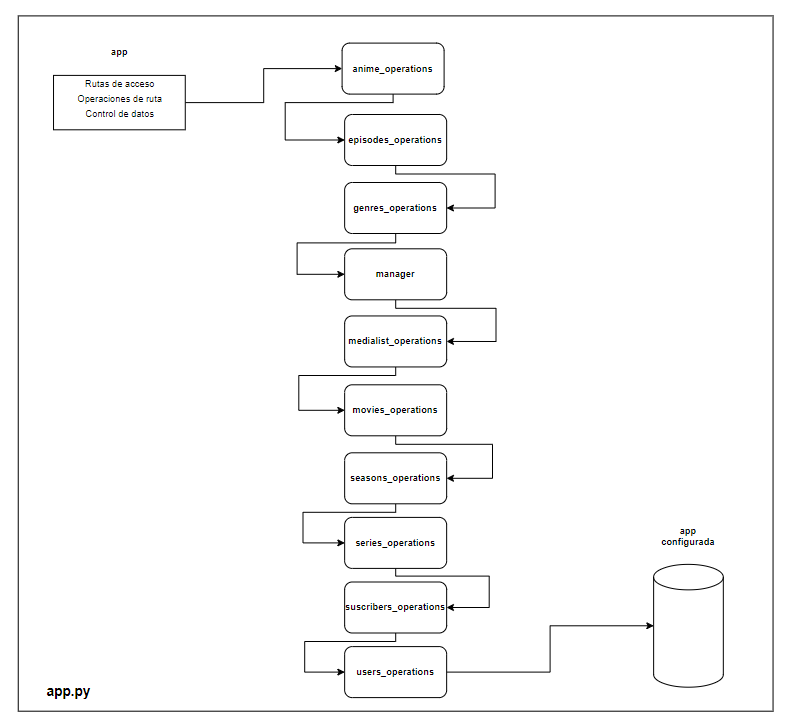
* Clase app.py: Clase principal de la aplicación, en la cual se instancia el objeto y se importan las clases que lo configuran.
* Database Models: Contiene todos los modelos de datos.
* Database Operations: Contiene todas las factorías, aplicando una configuración específica para cada una de las dependencias que se contemplan.

Para la configuración de nuestro objeto se sigue el siguiente flujo de ejecución:

1. Instanciación del objeto “app” en “app.py”
2. Inicialización de los modelos de base de datos para nuestro objeto “app”
3. Configuración del objeto para todos y cada uno de los modelos de datos, inicializando el método “init” de cada uno de ellos agregando cada una de las operaciones a nuestra aplicación.
4. Ejecución de la aplicación

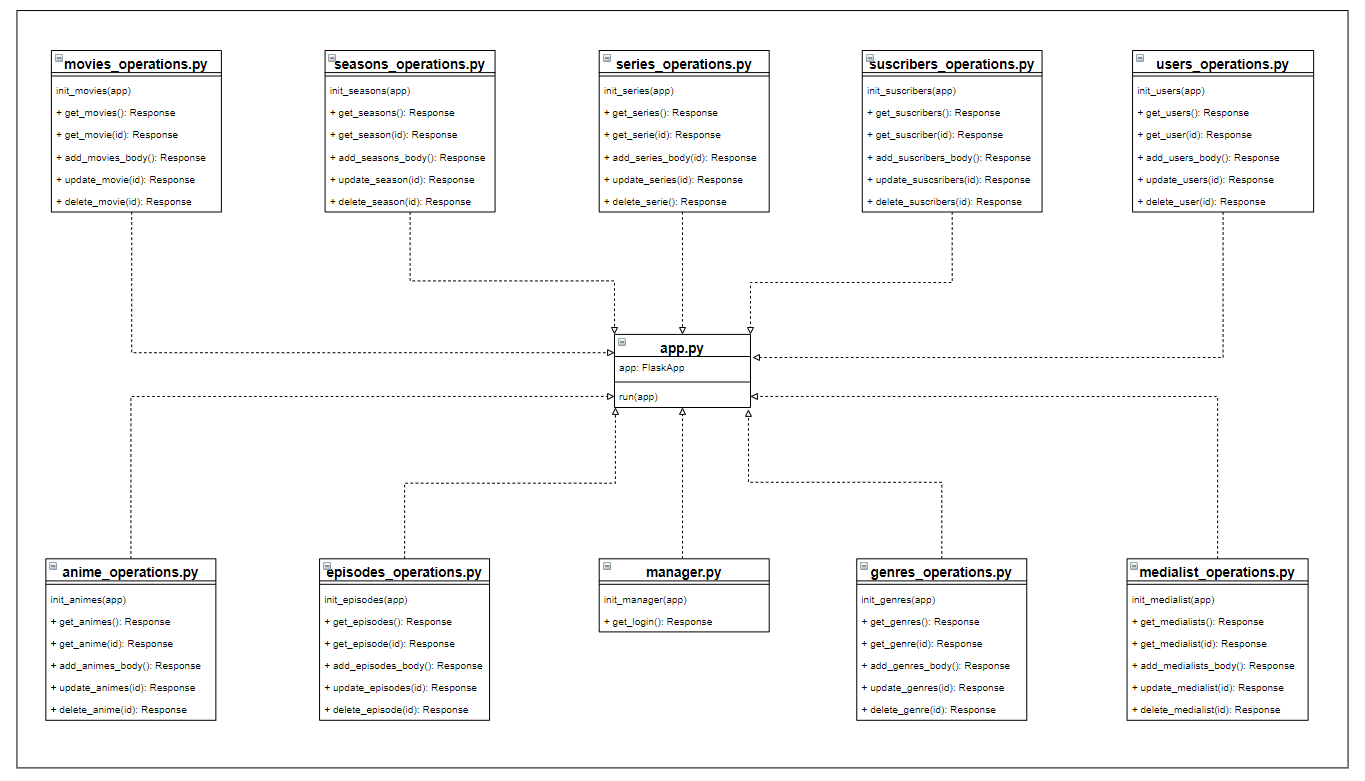


*Código de ejecución “app.py”*



*Ciclo de ejecución “app.py”*

Con esta implementación conseguimos desacoplar cada una de las operaciones para los modelos de datos implementados, así como los posibles nuevos modelos a incorporar.



*Modelo de Clases Factory*

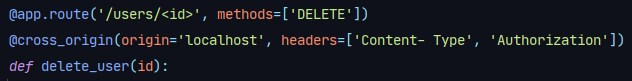
La clase “app.py” proporciona la instancia de “app” la cual va a ser implementada por cada una de las clases concretas las cuales van a agregar e implementar sus propias operaciones.

Dado a que estamos implementando la generación de un objeto a partir de una librería externa, se necesitan inyectar una serie de decoradores dependientes los cuales impiden la creación de una interfaz intermedia entre nuestras clases concretas y “app.py”, siendo “app.py” la propia interfaz la cual está siendo implementada por cada una de las clases concretas.

## **Decorator**

Los decoradores de Python son funciones que se utilizan para transformar otras funciones, con Flask el decorador “@app.route” se utiliza para hacer coincidir las URL con las funciones de las vistas en la aplicación instanciada.

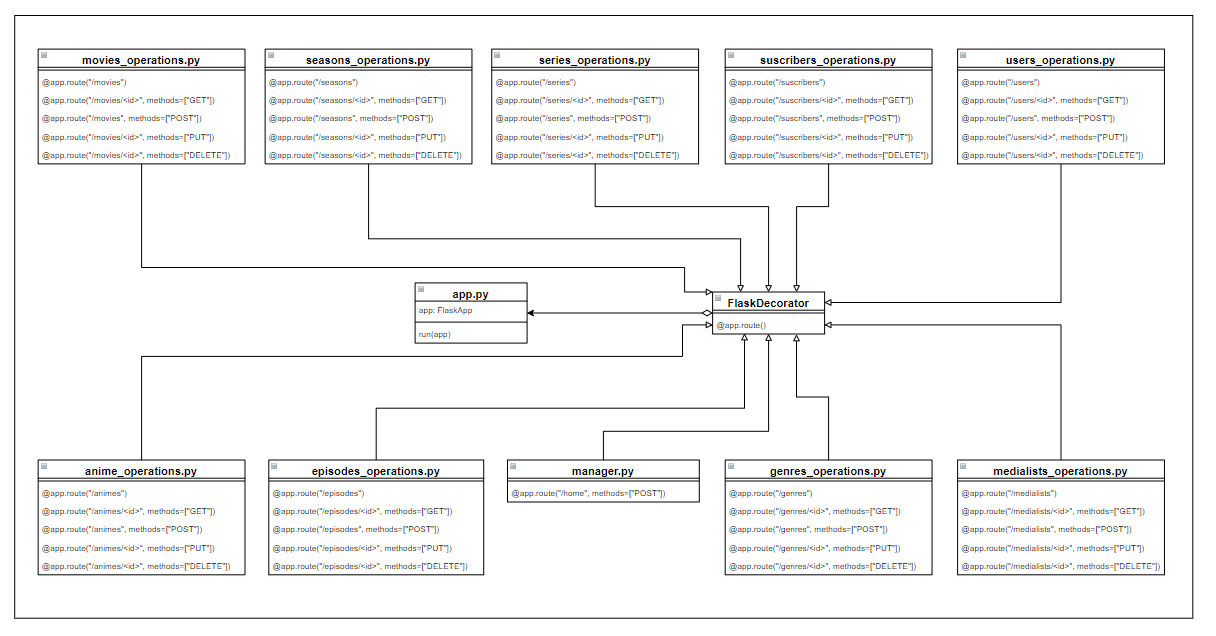
Las rutas definidas en la aplicación se han especificado utilizando variables de Python y pasadas a la función vista:



*Método “get\_users” de “users\_operations”*

En “@app.route” se está pasando el parámetro ‘/users/<id>, methods=[‘DELETE’]’, donde se especifica una ruta estática (‘users’) y un método (‘DELETE’), pero el campo <id> es una variable la cual va a depender del valor que le pasemos, en este caso el id del usuario que queramos eliminar.

Por otra parte, se ha incluido un segundo decorador también proporcionado por la librería de Flask, “@cross\_origin”, el cual nos permite añadir cabeceras en las respuestas entre las diferentes aplicaciones, verificando el intercambio de datos dado a los posibles problemas de seguridad de autenticación y verificación de contenido.



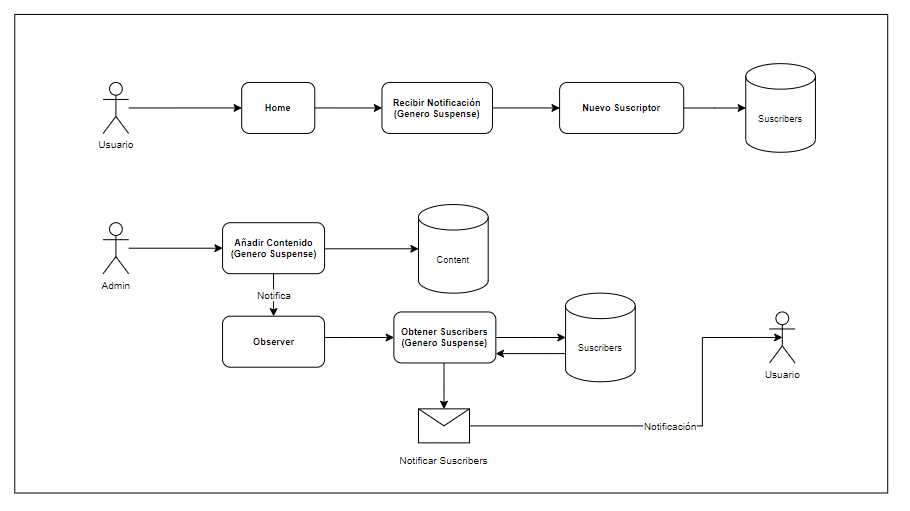
*Modelo de Clases Decorator*

## **Observer**

Para poder notificar a los usuarios que estén interesados en un tipo de genero independientemente del contenido, hemos implementado el patrón de diseño “Observer”, donde se van a crear suscriptores para cada uno de los géneros disponibles, el observador estará disponible para “Movies”, “Series” y “Animes”, en caso de que se incorpore un nuevo contenido y haya algún suscriptor con el mismo género que el contenido, este será notificado vía email.

En esta implementación se ha necesitado crear dos nuevos modelos de datos, “Genres” y “Suscribers”:

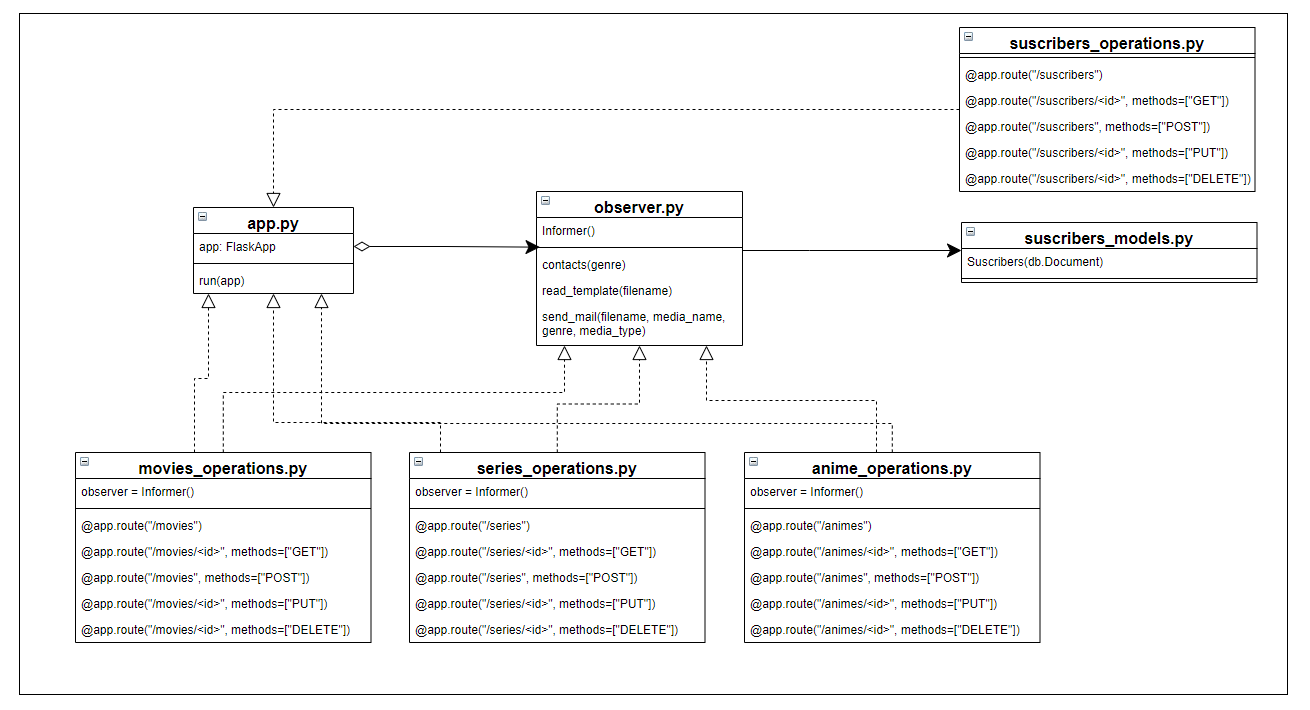
* “Genres”: Contiene todos los géneros de los contenidos de la aplicación.
* “Suscribers”: Usuarios que se han registrado para ser notificados cuando se incluya un nuevo contenido con un tipo de género concreto.



*Ciclo de ejecución “Observer”*

En el ciclo de ejecución del “Observer” podemos ver que sigue dos fases:

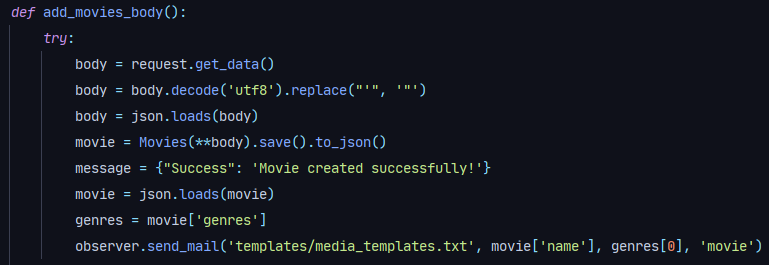
1. Un usuario accede al “Home” y registra su dirección de correo junto con un tipo de género determinado, este será suscrito y registrado dentro de la base de datos.
2. Un administrador añade un nuevo contenido a la aplicación con el mismo genero que el usuario que se había registrado, una vez añadido el “Observer” coge todos los “Suscribers” que se han dado de alta y los filtra por el nuevo género del contenido incorporado, en caso de que haya alguno este será notificado vía email.



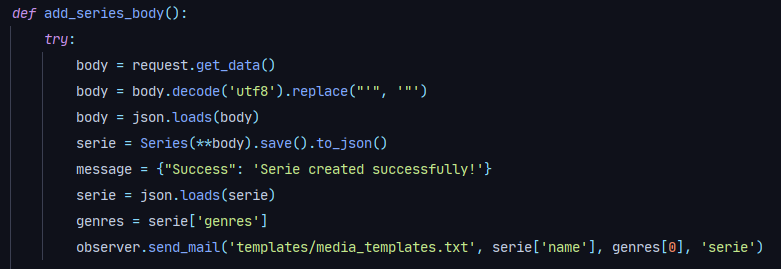
*Modelo de Clases “Observer”*

Las clases “movies\_operations.py”, “series\_operations.py” y “anime\_operations” instancian un objeto “Informer” de la clase “observer.py”, de la cual van a implementar los métodos de la clase acorde al contenido y género insertado.

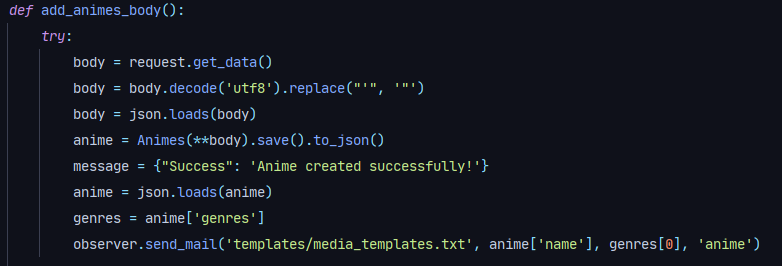
“Observer.py” por otra parte, va a utilizar la clase “suscribers\_models.py” para obtener los modelos de base de datos de “Suscribers” con el fin de poder gestionar todos aquellos usuarios registrados.



*Método “add\_movies\_body” de “movies\_operations.py”*



*Método “add\_series\_body” de “series\_operations.py”*



*Método “add\_animes\_body” de “animes\_operations.py”*

Como se puede observar en las tres capturas, cuando se ha conseguido realizar la inserción del nuevo contenido y se actualiza esto en base de datos, se implementa al “Observer” con el fin de notificar a los suscriptores correspondientes.