LA UNIVERSIDAD EN INTERNET

DISEÑO DE UNA APLICACIÓN CON PATRONES

Diseñar un servicio de series, películas y documentales online

Profesor: Luis Pedraza Gomara

Ernesto González Pradas
ernesto.gonzalez023@comunidadunir.net

Índice

Contenido

Introducción

Nos encontramos diseñando la arquitectura para una aplicación de servicios de distribución online de contenido audiovisual: series, películas y documentales. En nuestra aplicación, los usuarios se podrán autenticar en la misma, utilizando sus cuentas en las distintas redes sociales en las que estén registrados (Facebook, Instagram, Twitter, etc.), es decir, no hace falta que se registren en nuestra aplicación con un usuario y contraseña si no que con su cuenta en Facebook puede hacer login en nuestra plataforma. Los usuarios también pueden buscar dicho contenido audiovisual y reproducirlo en múltiples dispositivos, es decir, pueden ver una serie desde el móvil mediante la aplicación (para Android una, para iPhone otra), desde el ordenador mediante un navegador web o su aplicación de escritorio y desde una televisión mediante su aplicación para Smart TV. Finalmente, los usuarios podrán subir su propio contenido al servicio, es decir, podrán subir sus propias creaciones a la aplicación cloud y estas a su vez tendrán que ser procesadas para adaptar el vídeo a los diferentes dispositivos expuestos con anterioridad, comprobar si tienen copyright y extracción automática de subtítulos.

A continuación, vamos a indicar que modelos de servicio cloud serían necesarios para el desarrollo de este sistema descrito con sus funcionalidades.

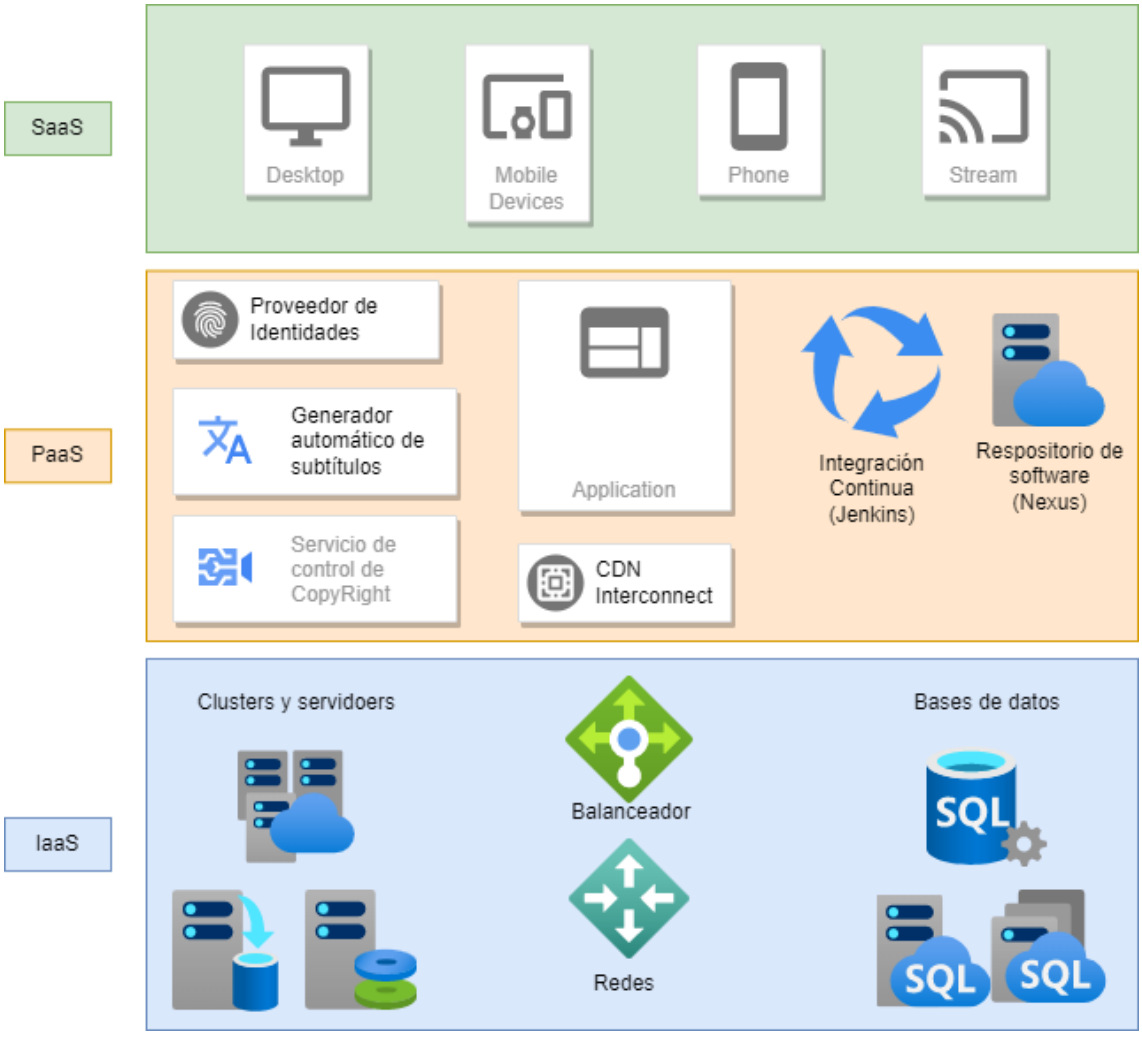
Por un lado, necesitaremos como desarrolladores recursos computacionales de bajo nivel, es decir, necesitaremos bases de datos para poder almacenar los contenidos (vídeos, comentarios de usuarios al contenido audiovisual o datos de los propios usuarios como número de tarjeta, email, edad, etc.) que suban los usuarios, redes y capacidad de procesamiento para poder balancear y redistribuir las peticiones que puedan realizar los usuarios (podríamos tener muchas peticiones desde dispositivos móviles y pocas desde la aplicación de escritorio) y así, evitar problemas de disponibilidad como podría ser “muerte por inanición” y sobre todo un entorno en el que poder desplegar nuestro software desarrollado que conectará con diferentes servicios como servicios de autenticación (para la funcionalidad de poder hacer loguin en nuestra aplicación con una cuenta de una red social), servicios de comprobación de copyright, etc. Como podemos observar necesitaremos Infraestructura como servicio (IaaS).

Como hemos visto en el párrafo anterior, como desarrolladores, necesitamos una plataforma dentro de la infraestructura para desplegar nuestro propio código de la aplicación y el acceso a otras aplicaciones de las que nos serviremos de sus

funcionalidades. Para ello necesitamos una plataforma como servicio (PaaS), en la que tendremos, además, instalados un repositorio de software (podríamos utilizar Nexus concretamente) para la gestión de dependencias y un servidor (por ejemplo, Jenkins para compilar y probar los desarrollos que se van realizando de nuestra aplicación) para la integración continua de nuestro software desarrollado. En esta plataforma también dispondremos de servicios de terceros como el proveedor de Identidades, el generador automático de subtítulos o los servicios de control de copyright. También dispondremos de nuestro CDN donde estarán nuestros archivos estáticos.

Finalmente, necesitaremos un software como servicio (SaaS), donde como desarrolladores tendremos expuesta la accesibilidad al distinto tipo de aplicaciones que va a utilizar el usuario final. Estas aplicaciones son las descritas al principio del epígrafe y serán accesibles desde el navegador web, la aplicación para Android y iPhone, la aplicación de la Smart TV y la aplicación de escritorio.

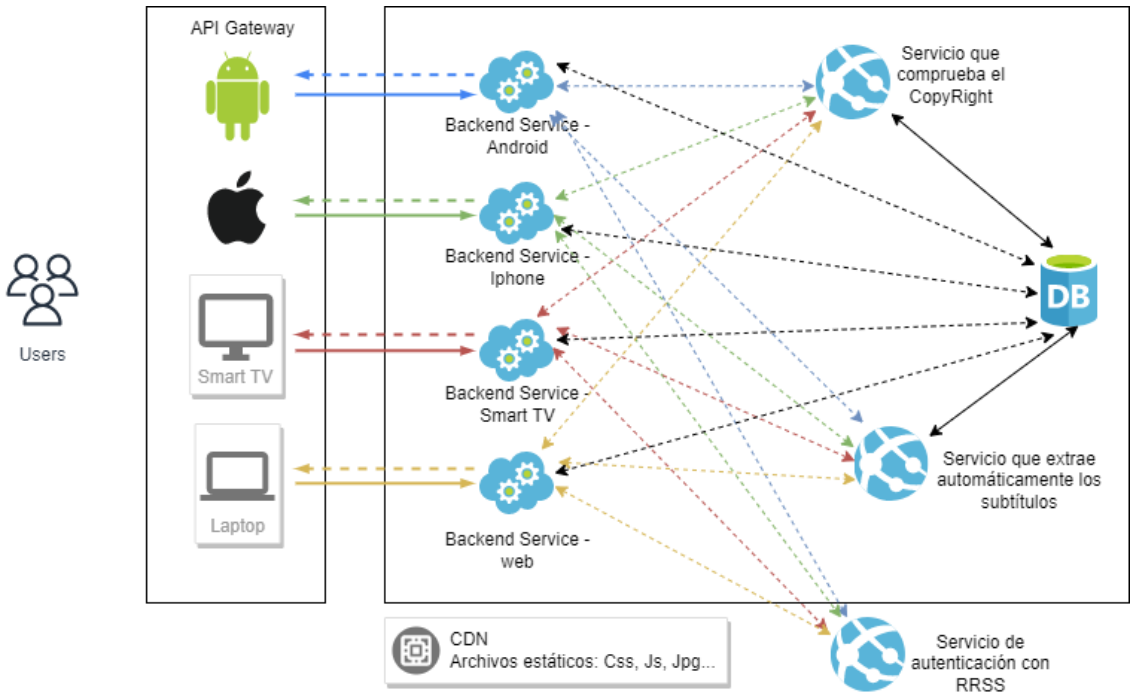
A continuación, se muestra un diagrama (de alto nivel) de como quedarían los distintos tipos de modelo de servicio cloud:



Diseño de la arquitectura del sistema

Para realizar la representación del diseño de alto nivel de todo el sistema nos hemos basado en tres patrones arquitectónicos: Back-end para Front-End (BFF), Microservicios y Patrón de identidad federada.

A continuación, mostraremos primero el diseño explicando cada uno de sus componentes principales y después veremos los tres patrones de arquitectura:



Como vemos en la imagen superior tenemos varios componentes principales. Por un lado, tenemos los distintos tipos de dispositivos que se pueden conectar a nuestra aplicación (estos serían cada uno de los dispositivos desde los que se puede acceder a nuestro servicio de distribución online), todos los back-end específicos para cada uno de los dispositivos (cada uno de ellos permite un menor acoplamiento y mejora la mantenibilidad, disponibilidad y mejora de los servicios), tenemos un servicio que nos comprobará el copyright de los videos que suban los usuarios a la plataforma, otro servicio que generará automáticamente los subtítulos de dichos contenidos y finalmente nuestra base de datos donde almacenaremos todos los datos necesarios para que funcione nuestra aplicación (los propios contenidos mostrados, datos de los usuarios, etc.).

También, dispondremos de un proveedor de identidades externo (servicio externo de autenticación con RRSS) y nuestro CDN (Content Delivery Network) con los

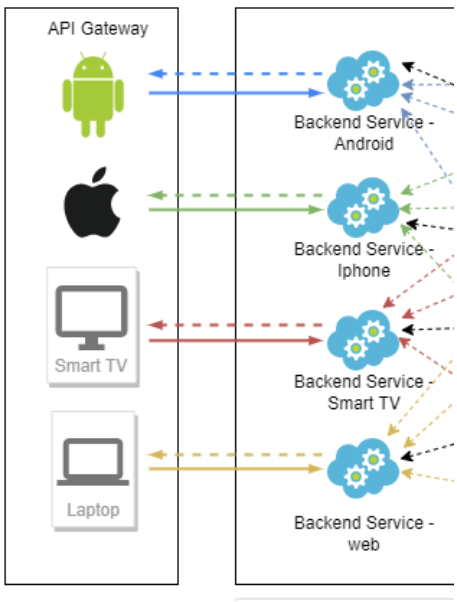
distintos archivos estáticos (CSS, JavaScript, png, jpg, etc.) que estarán disponibles cuando los usuarios los soliciten a través de los distintos dispositivos de visualización.

Teniendo una perspectiva general del conjunto del sistema y sus partes, pasamos a describir los patrones de arquitectura utilizados.

Uno de los requisitos funcionales que debe tener nuestro servicio de distribución online de contenido audiovisual, es poder buscar y reproducir series, películas y documentales en distintos dispositivos, ya sean móviles (Android, iPhone), navegador web, aplicación de escritorio o aplicación para Smart TV. Como vemos, esto nos plantea un problema, y es que, cada uno de estos dispositivos tiene sus propias particularidades: el tamaño en el que se va a visualizar el contenido es distinto, la usabilidad de un móvil no es igual que la de un navegador web en un portátil, los recursos computacionales que necesitan estos dispositivos son distintos y en cuanto al nivel de peticiones, hay estudios que indican que el 70% de las personas visualizan el contenido en una TV, aunque se loguen a través del ordenador o el móvil.

Para solventar dichos problemas, utilizaremos el patrón Back-end para Front-end. Desarrollaremos un back-end específico para cada uno de los posibles dispositivos en los que se pueda usar nuestro servicio de distribución. Con ello, podremos optimizar los servicios necesarios para cada uno de los dispositivos, haciendo que dichos servicios sean más sencillos de mantener y escalar. Con esto podemos eliminar el acoplamiento entre clientes diferentes con casos de uso específicos.

A continuación, se muestran los elementos más importantes de esta arquitectura:



Como podemos observar, tenemos los distintos dispositivos desde los que podemos acceder a nuestro servicio de contenido audiovisual (lado izquierdo) y los back-end específicos para cada uno de los dispositivos.

Al principio del epígrafe veíamos el diseño completo de nuestro servicio de distribución online de contenido audiovisual. En el podemos apreciar como nuestra aplicación está formada por pequeños servicios independientes que implementan cada una de las necesidades de nuestro proyecto (cada uno de los servicios de back-end para cada dispositivo, el servicio que comprueba el copyright de los vídeos, el servicio que genera de forma automática los subtítulos y el servicio que autentica a los usuarios en la aplicación con sus cuentas en RRSS). Esto es lo que se denomina Arquitectura basada en Microservicios.

Las partes más importantes de nuestra Arquitectura basada en microservicios serían: los microservicios citados anteriormente (independientes unos de otros, manejan y gestionan sus propios datos, como por ejemplo el tamaño óptimo del video que va a enviar al cliente, los recursos estáticos necesarios para dicha resolución, etc. y estado interno) y la puerta de enlace API (API Gateway) los usuarios establecen una conexión desde los distintos dispositivos a dicha puerta de enlace que es la que gestionará las conexiones con sus respectivos servicios de back-end.

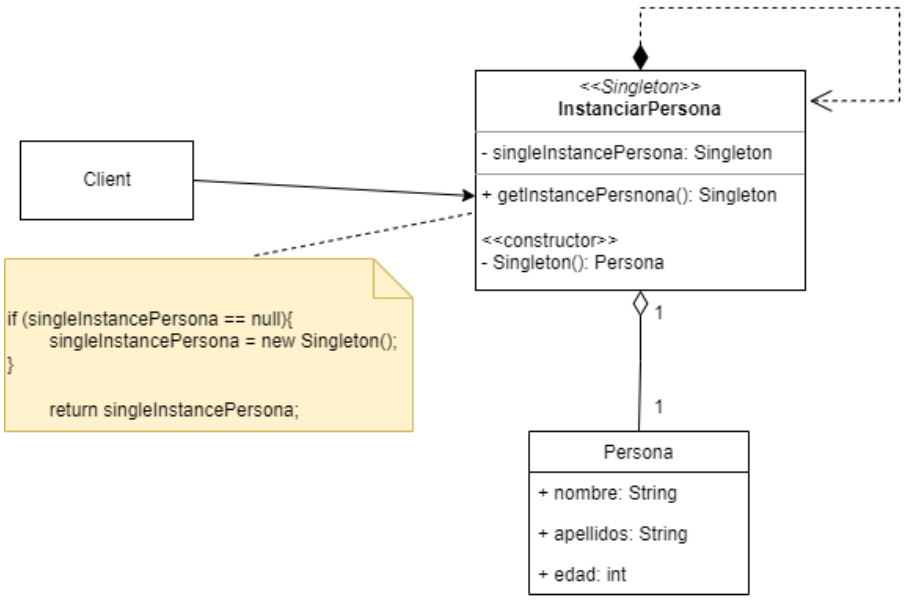
Otra de las especificaciones que tiene nuestra aplicación, es que los usuarios se pueden loguear sin necesidad de usar un usuario y contraseña específicos para nuestro servicio, mediante una cuenta suya de cualquier red social (Facebook, Twitter, etc.). Esto lo conseguimos con el Patrón de Identidad Federada.

Con este patrón, lo que haremos será delegar la función de autenticación en un proveedor de identidades externo (servicio de autenticación con RRSS) tal y como está indicado en la figura del esquema de nuestro sistema.

El usuario desde uno de los dispositivos intenta obtener servicio de su correspondiente servicio de back-end, pero previamente este tiene que establecer una conexión segura con el servicio de autenticación externo. Una vez que este back-end tiene establecida una relación de confianza con el servicio de autenticación, el cliente puede autenticarse y solicitar un token que le identifica y además proporciona sus datos (nombre, apellidos, edad, etc.). Esto se podría asemejar, a un SSO o Single Sign-On donde con un token de autenticación puedes acceder a varios sistemas y llevas tus datos personales.

Diseño de componentes y clases

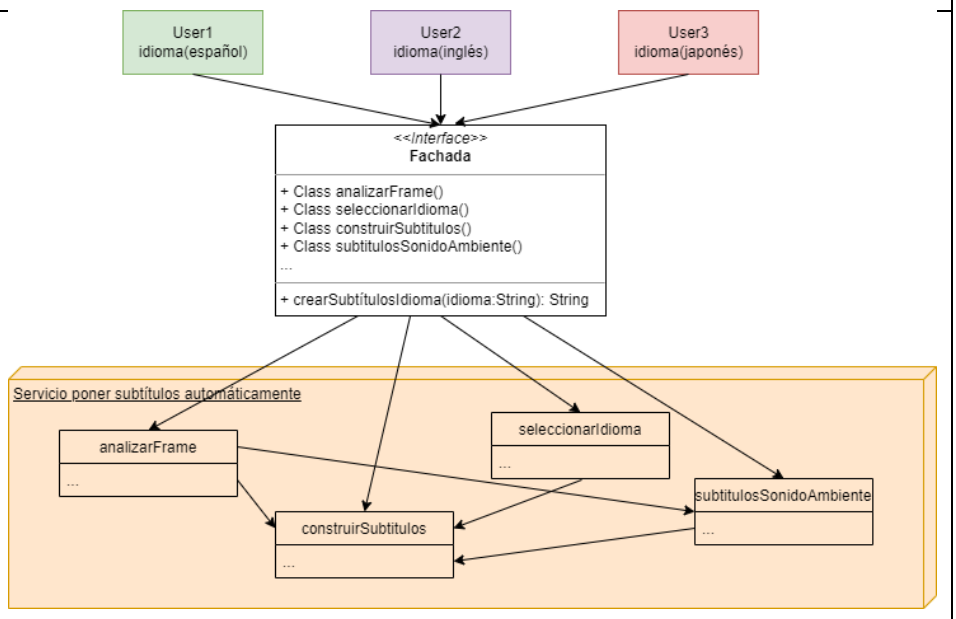
A continuación, se muestran las tablas por cada uno de los patrones seleccionados.

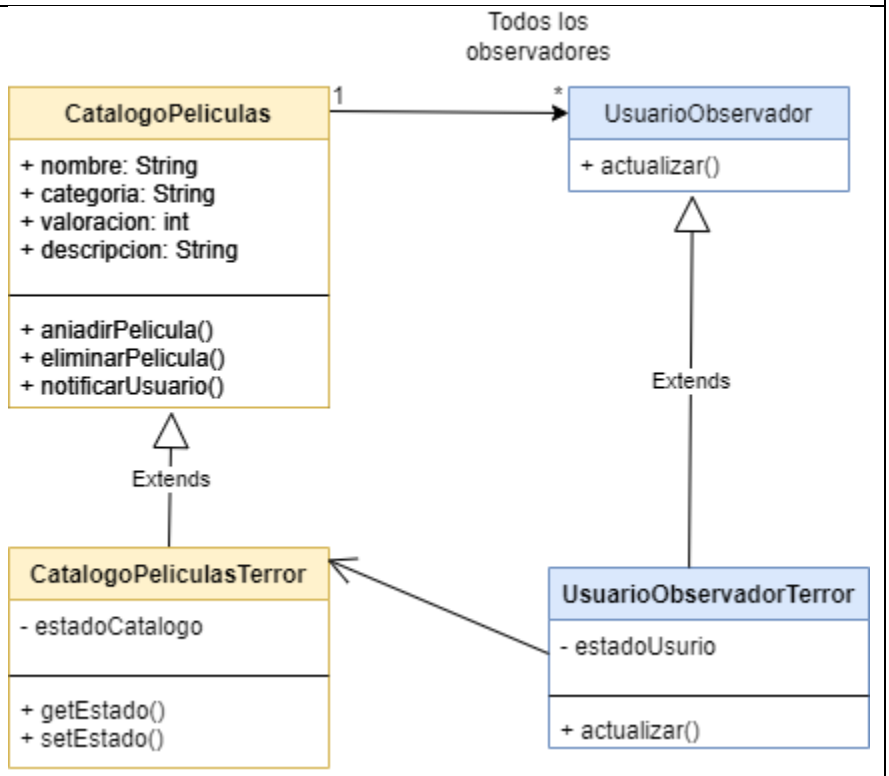
Problema identificado	El primer problema que tenemos en nuestra aplicación es que, una vez se han obtenido las credenciales mediante el proveedor de identidades, necesitamos crear el objeto Persona con sus datos solo una vez y acceder a este recurso desde cualquier punto de la aplicación. Por ejemplo, necesitamos el nombre para pintar arriba a la derecha "Hola Luis" ya que es la persona logueada, o cuando le vamos a recomendar una serie poner un título como "Estas son las nuevas recomendaciones para ti Luis". Accedemos a los atributos del objeto Persona (nombre, apellidos, tipo de usuario, edad, etc.) desde cualquier punto en cualquier momento y dichos atributos no van a cambiar durante toda la sesión.
Patrón que resuelve el problema	Este problema lo podemos resolver con el Patrón de Creación **Singleton**. La forma en que lo resuelve es, una vez que obtenemos el token de verificación del proveedor de identidades, extraemos los datos del usuario (nombre, apellidos, edad, etc.) y creamos un objeto de tipo Persona, el cual rellenaremos en el constructor privado (para evitar que otras clases puedan hacer un new de esta clase) de la clase Singleton previamente invocado por un método estático que actúa de constructor, y lo guarda en un campo estático.
Diagrama del diseño detallado del patrón utilizado	

A continuación, adjunto una captura de la clase implementada para aclarar el diagrama:

```
public InstanciarPersona class Singleton
{
    private static Singleton singleInstancePersona = null;
    private static Persona persona = new Persona();

    private Singleton() {
        persona.setNombre("Ernesto");
        persona.setApellidos("González Pradas");
        persona.setEdad(29);
    }

    public static Singleton getInstancePersona()
    {
        if(singleInstancePersona == null){
            singleInstancePersona = new Singleton();
        }

        return singleInstancePersona;
    }
}
```

Como podemos observar en el código, tenemos nuestro constructor privado para que ninguna otra clase pueda hacer un new de esta, donde rellenamos los datos a partir del token obtenido del proveedor de identidades (por simplificar en el código se ha puesto a fuego los atributos del objeto Persona). Cuando el cliente llama al método getInstancePersona() si la instancia ya existe simplemente devuelve el objeto y si no lo crea con el new.

Problema identificado	Tenemos que traducir a varios idiomas los vídeos que tenemos en la plataforma.
Patrón que resuelve el problema	Este problema lo podemos resolver con el Patrón de Estructura **Fachada** (Facade). Utilizaremos un servicio que es el que tiene todas las clases que implementan los subtítulos en diferentes idiomas y para ello pondremos una interfaz única que dependiendo desde que país se conecte el usuario pondremos los subtítulos en ese idioma. Esta implementación también nos serviría para cuando el usuario quiera poner dichos subtítulos en un idioma diferente.
Diagrama del diseño detallado del patrón utilizado	

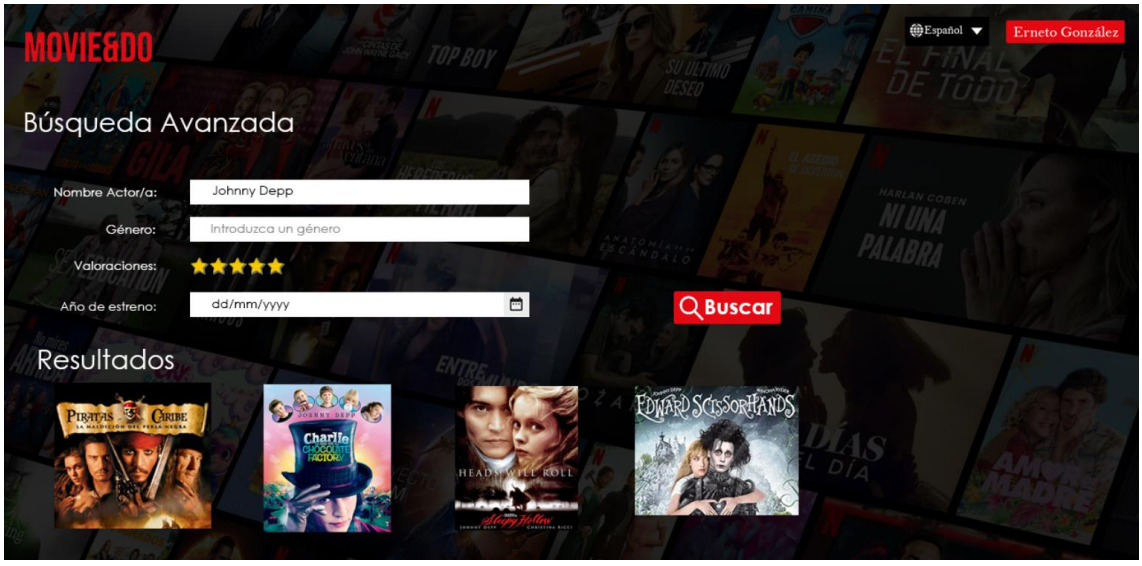
Problema identificado	Tenemos la necesidad de que los usuarios quieren ser avisados de las nuevas películas/series/documentales que se suben a la plataforma, de una categoría concreta (por ejemplo: series de terror).
Patrón que resuelve el problema	Este problema lo podemos resolver con el Patrón de Comportamiento **Observer** (Observador). Este patrón lo resuelve de tal forma que tenemos una clase sujeto (por ejemplo: Catálogo de películas) donde se producen cambios cuando se sube contenido nuevo y varias clases observador (las clases de usuario) que serán avisadas de las actualizaciones en la clase sujeto. Así cuando se añada una película del género de terror se le mandará una notificación a todos aquellos usuarios que estén suscritos a las películas de terror.
Diagrama del diseño detallado del patrón utilizado	

Diseño de la interfaz del usuario

Vamos a incluir dos patrones de diseño de interfaz.

El primero está orientado a la búsqueda de contenido avanzado, es decir, cuando el usuario quiere buscar todas las series/películas/documentales en las que sale un actor en concreto utiliza una serie de filtros para realizar la búsqueda. El patrón en concreto se llama búsqueda avanzada (Avanced Search) y está extraído del catálogo *Patterns in Interaction Design*. Con este patrón le daremos al usuario la funcionalidad de buscar un contenido audiovisual no solo por el nombre, si no por los actores que intervienen, año de estreno, género, valoraciones, etc.

A continuación, se muestra un mockup realizado con Photoshop de como quedaría la implementación de este diseño en nuestro sistema. Como se puede ver el usuario ya logueado, puede realizar búsquedas avanzadas por nombre de actor, género, valoraciones o año de estreno. En el ejemplo se ha querido mostrar todas las películas en las que ha participado Johnny Depp con una valoración media de los usuarios de cinco estrellas:



Para el segundo patrón, queremos mostrar al usuario todos los contenidos que tiene a medias en cuanto a reproducción, para que pueda hacer clic en uno de ellos y seguir por donde estaba. Para resolver este problema utilizaremos el patrón Carrousel, extraído del catálogo *Patterns in Interaction Design*.

Con este patrón le mostraremos al usuario ya logueado un carrousel con las imágenes de las series/películas/documentales que estaba viendo y podrá hacer clic en una de ellas para seguir reproduciendo el contenido por el minuto en el que estaba.

A continuación, se muestra un mockup realizado con Photoshop de como quedaría la implementación de este diseño en nuestro sistema:

Como se puede observar en la imagen, el usuario acaba de entrar en su sesión y lo primero que le aparece es un carrousel con las películas y series que ha estado visualizando, para que pueda seguir por donde lo ha dejado.

Conclusión

Como hemos podido ver a lo largo de la actividad, hemos usado diferentes tipos de patrones (patrones de arquitectura, de diseño, cloud, etc.) en cada una de las fases de desarrollo de un proyecto. Aunque hemos utilizado patrones como microservicios, Back-end para Front-end… Es verdad que se podrían usar otros patrones como, por ejemplo: cliente-servidor (en nuestro caso de cliente ligero ya que toda la lógica de negocio y las operaciones con los datos se realizan en el back, aunque si utilizásemos alguna tecnología como angular, sí que sería de cliente pesado), Factoría abstracta, Builder, Proxy, etc.

No solo podemos utilizar distintos patrones para resolver los mismos problemas, sino que podemos apreciar como muchos de estos patrones se combinan entre sí casi de manera natural, como por ejemplo microservicios y back-end para front-end o con el patrón fachada.

Bibliografía

Chakray. (s.f.). *Chakray*. Obtenido de https://www.chakray.com/what-is-single-sign-on-sso-definition-characteristics-and-advantages/

Gomara, L. P. (1 de Mayo de 2022). Tema 05: Patrones de arquitectura. Madrid, Madrid, España.

Gomara, L. P. (1 de Mayo de 2022). Tema 06: Patrones de diseño. Madrid, Madrid, España.

Gomara, L. P. (1 de Mayo de 2022). Tema 07: Arquitectura en La Nube.

Kafka, P. (7 de Marzo de 2018). *recode*. Obtenido de https://www.vox.com/2018/3/7/17094610/netflix-70-percent-tv-viewing-statistics

refactoring.guru. (s.f.). *Observer - refactoring.guru*. Obtenido de https://refactoring.guru/es/design-patterns/observer

refactoring.guru. (s.f.). *Singleton - refactoring.guru*. Obtenido de https://refactoring.guru/es/design-patterns/singleton