

AppVideo

Caso práctico de Tecnologías de Desarrollo Software



26 de junio de 2022

Grupo: 3.3

Raúl Luján Pascual con DNI 48746269T

raul.lujan@um.es

María Inmaculada Campillo Soto con DNI 49339676F

mariainmaculada.campillos@um.es

Git: https://github.com/RaulLujan/AppVideo.git

Contenido

[Introducción 1](#_Toc106992722)

[Diagrama de clases de dominio 2](#_Toc106992723)

[Diagrama interacción añadir video a lista 3](#_Toc106992724)

[Arquitectura aplicación 3](#_Toc106992725)

[Visual 3](#_Toc106992726)

[Lógica de la aplicación 4](#_Toc106992727)

[Persistencia 4](#_Toc106992728)

[Patrones diseño usados 4](#_Toc106992729)

[Componentes 4](#_Toc106992730)

[Test unitarios 4](#_Toc106992731)

[Manual de usuario 4](#_Toc106992732)

[Observaciones finales 4](#_Toc106992733)

# Introducción

AppVideo es una aplicación de escritorio hecha en Java para poder visualizar videos. Este documento contiene la explicación de cómo está estructurado la aplicación AppVideo y un manual de usuario para poder navegar.

Más concretamente, primero se hablará de la estructura de diseño de la aplicación para después centrarse en la del código y la aplicación en sí. Asimismo, se comentará los componentes y test unitarios.

Por otra parte, hemos añadido un manual de usuario para facilitar el uso de AppVideo.

# Diagrama de clases de dominio

Para empezar, mostramos en la Figura 1 el diagrama de clases del dominio.

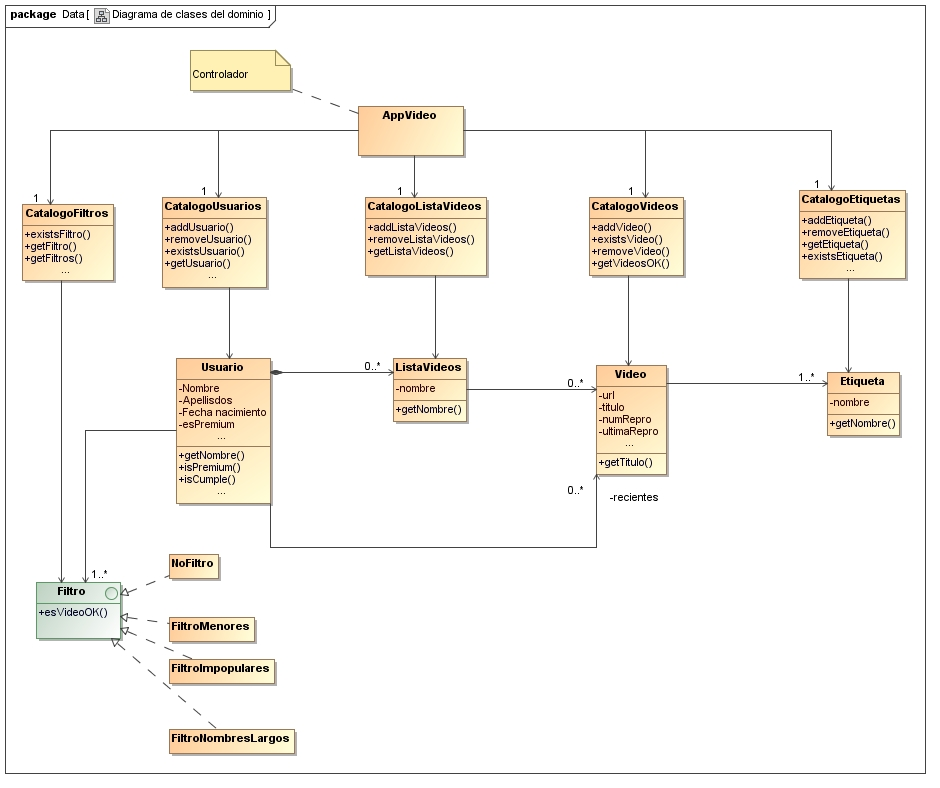
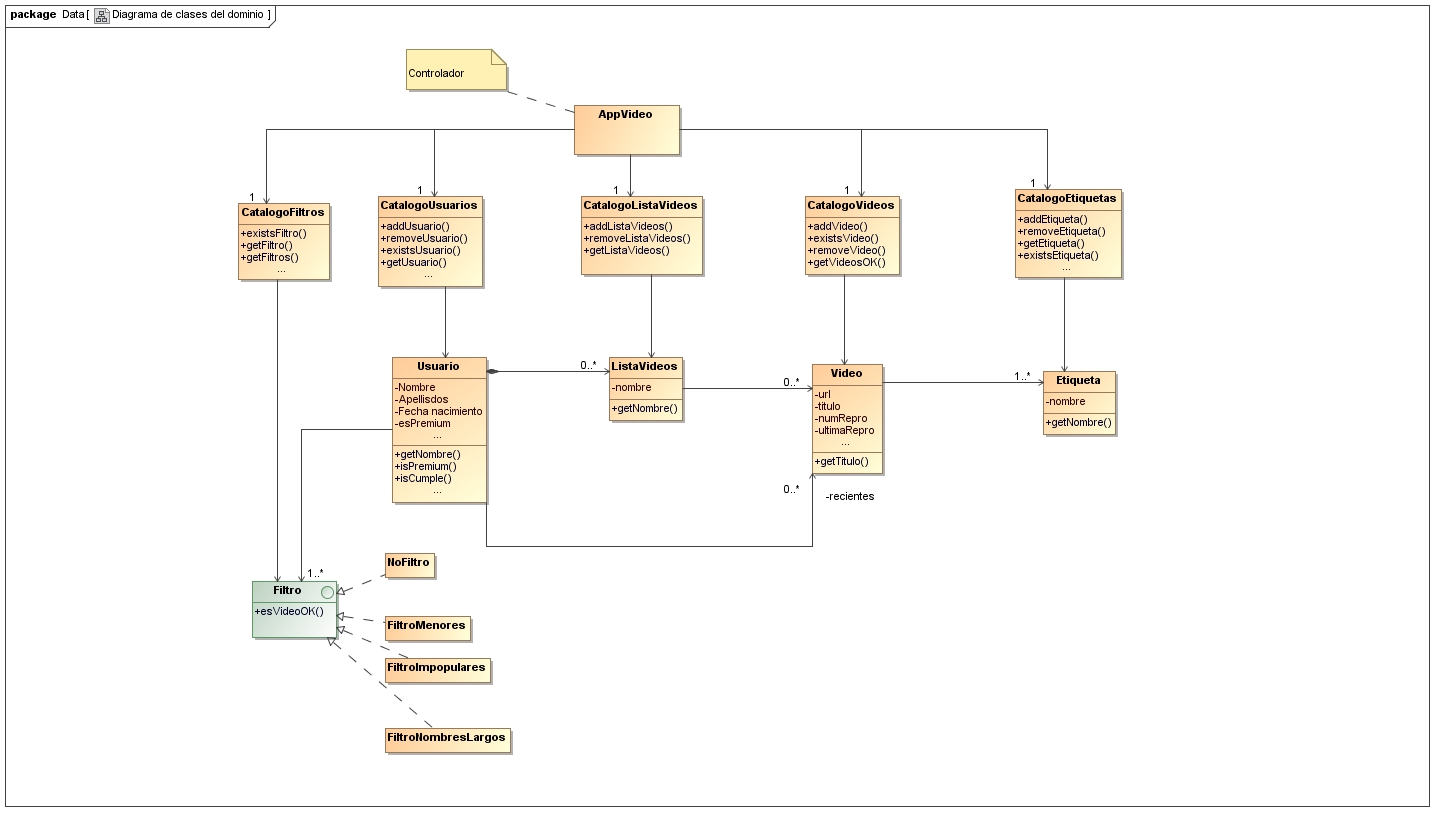


Figura . Diagrama de clases del dominio.



Como se puede observar, hemos puesto algunos atributos y métodos importantes de cada clase.

En cuanto al usuario Premium, en vez de tener una clase con el rol, hemos optado por que sea un atributo de la clase Usuario. Esto se debe a que la diferencia entre un usuario premium y otro que no lo es, es mínima. Asimismo, y haciendo uso del patrón *Null Object*, hemos implementado una clase NoFiltro para aquellos usuarios que no son premium. Esto nos ha llevado a tener también una jerarquía de filtros.

# Diagrama interacción añadir video a lista

En la Figura 2 mostramos la interacción que se hace dentro de la aplicación cuando el usuario añade un nuevo video a una lista seleccionada. Como se puede observar, es el controlador quien llama a las correspondientes instancias para añadir el video.

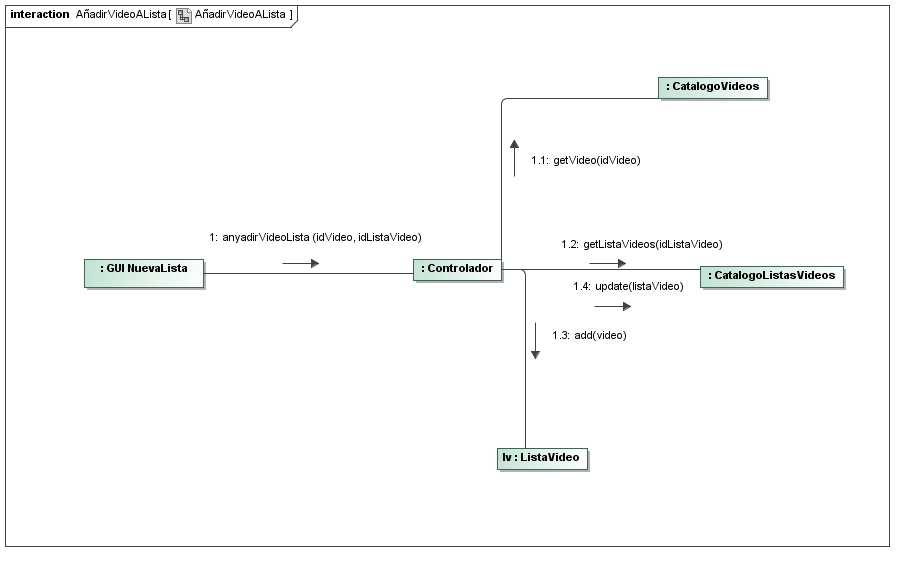


Figura . Diagrama de interacción.

# Arquitectura aplicación

El proyecto se divide en tres capas claramente diferenciadas: vista, modelo y persistencia. La visual y el modelo se comunican a través de un controlador y el modelo se comunica con la persistencia gracias al uso del patrón DAO. Si nos vamos al proyecto en sí, podemos ver los diferentes paquetes *vista*, *controlador*, *modelo* y *persistencia*, entre otros.

A continuación, vamos a explicar cada capa y sus decisiones a la hora de diseñarlas.

## Visual

La parte visual de AppVideo está implementada en Java Swing y con ayuda de algunos componentes como *JCalendar* del que hablaremos en el apartado de [Componentes](#_Componentes).

En la Figura 3 mostramos como es la estructura de esta parte, dado que hay muchas clases, solo hemos ilustrado las correspondientes a Login, Registro y Explorar. Sin embargo, todas siguen la misma estructura, en LaminaSuperior aparecen los botónes y en LaminaCentral se muestra su funcionalidad.

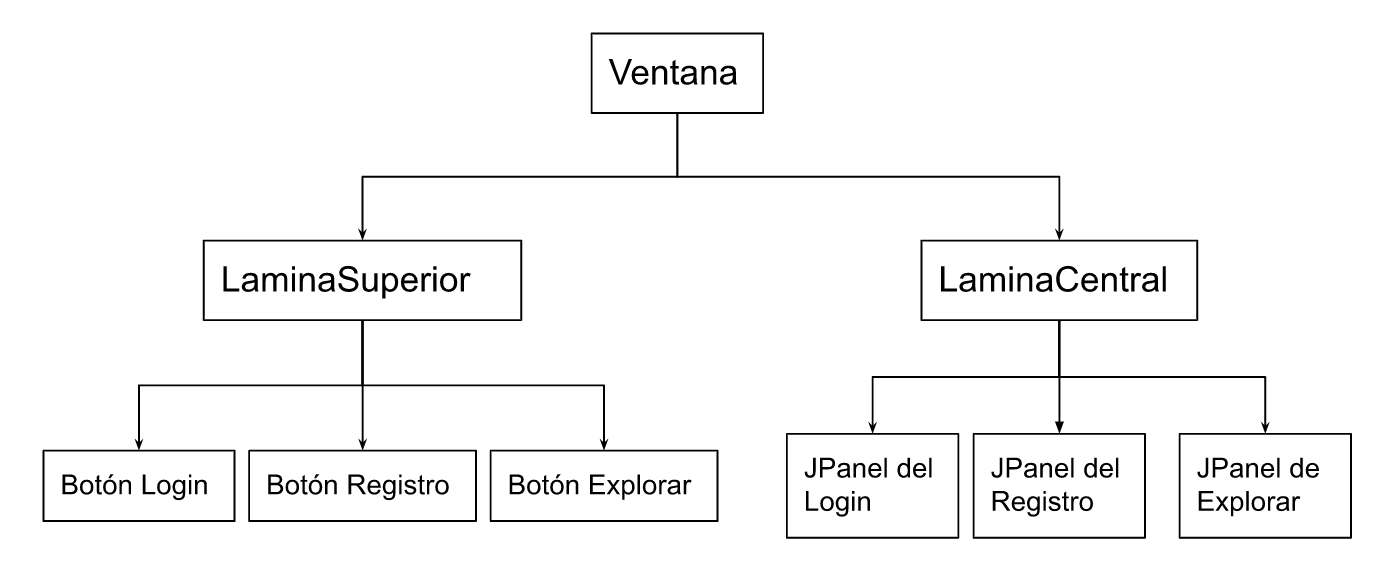


Figura . Estructura de la capa visual.

Inicialmente, el usuario verá una ventana con las opciones de iniciar sesión, registrarse o cargar videos de un XML. Esencialmente, dicha ventana está dividida en por dos paneles o laminas como hemos adelantado antes: *LaminaSuperior* (con los correspondientes botones) y *LaminaCentral* (con la funcionalidad correspondiente a cada botón). Es esta última la que se encarga, por medio de introspección, de crear las instancias de aquellos paneles necesarios para llevar a cabo la funcionalidad la primera lamina.

El usuario, una vez que entra en la aplicación, le aparece diferentes botones como Explorar o Generar PDF si es un usuario premium. Todas estas nuevas funcionalidades funcionan igual que el registro o el login. Son *JPanel* que se van añadiendo o eliminando de *LaminaCentral*.

Dichos *JPanels* están mayoritariamente puestos con el layout *GridBagLayout*, es uno de los más potentes en Java y, por tanto, difícil de entender. Sin embargo, permite dividir el panel en celdas y distribuir los componentes con ayuda de *GridBagConstraints*.

Esta capa se comunica con la lógica gracias a un controlador. Como se mostró en la Figura 2 del apartado [Diagrama de interacción](#_Diagrama_interacción_añadir), todo elemento de la visual, si necesita trabajar con datos, debe “hablar” con el controlador.

## Lógica de la aplicación

En esta parte de aplicación, es donde aparecen las clases que se han representado en la Figura 1, tanto la de *Usuario*, *Video* y *Filtro*, entre otras, como los correspondientes catálogos. Estos son utilizados por el controlador.

Las clases como *Usuario*, *Video*, *Etiqueta* y *Filtro* tienen métodos básicos, también llamadas clases *POJO* (Plain Old Java Object), aunque también tienen atributos calculados como Usuario tiene *isCumple*. Por otro lado, los catálogos son los encargados de añadir, modificar y eliminar, a través de un adaptador, los diferentes registros. Sin embardo, está *CatalogoVideos* que tiene más funcionalidad como obtener los videos según las etiquetas que tengan o los videos más vistos.

Por último y como se ha comentado esta capa se comunica con la base de datos gracias a los adaptadores de aquellas entidades que se almacenan.

## Persistencia

Inicialmente se había elegido MySQL como servicio de persistencia. Sin embargo, decidimos cambiar a H2. Dado que estamos usando los adaptadores con el patrón *DAO* (en otras palabras, la suma de patrones como *Adaptador*, *Factoría abstracta* y *Singleton*), el cambio entre un tipo u otro ha fácil de llevar a cabo.

En nuestra base de datos guardamos las etiquetas, lista de videos, videos y los usuarios. Por tanto, necesitamos cuatro interfaces y sus correspondientes clases que hacen uso del servidor de persistencia. Tenemos las interfaces *AdaptadorEtiquetaDAO*, *AdaptadorListaVideosDAO*, *AdaptadorUsuarioDAO* y *AdaptadorVideoDAO* y las correspondientes clases llamadas igual, pero terminando con TDS, en vez de DAO. Además, en cada una de ellas hacemos uso de *ServicioPersistencia*.

# Patrones diseño usados

A lo largo de este documento hemos hablado de los diferentes patrones usados, pero en esta sección vamos a definir con más detalle cada uno de ellos.

En primer lugar, en la lógica de la aplicación tenemos la clase Filtro en la cual llegamos a usar dos patrones. El primero es el patrón *Strategy*, el cual permite definir algoritmos de la misma familia y colocar cada uno en clases distintas. La ventaja de este patrón es que se puede intercambiar las clases sin provocar ningún daño. El otro patrón que se usa es *Null Object* para evitar valores nulos, en concreto, esto lo utilizamos con la clase *NoFiltro*.

public abstract class Filtro {

private String descripcion = "";

public String getDescripcion() {

return descripcion;

}

protected Filtro(String descripcion) {

super();

this.descripcion = descripcion;

}

public abstract boolean esVideoOK(Video video, Usuario usuario);

}

Cuadro de texto . La clase Filtro es la clase padre para NoFiltro y FiltroImpopulares, entre otros.

// Clase NoFiltro

public boolean esVideoOK(Video video, Usuario usuario) {

return true;

}

// Clase FiltroImpopulares

public boolean esVideoOK(Video video, Usuario usuario) {

return video.getNumRepro() >= 5;

}

// Clase FiltroMenores

public boolean esVideoOK(Video video, Usuario usuario) {

CatalogoEtiquetas catalogo = CatalogoEtiquetas.getInstancia();

if (usuario.getEdad() >= 18 || !catalogo.existsEtiqueta("Adultos"))

return true;

return video.containsEtiqueta(catalogo.getEtiqueta("Adultos"));

}

Cuadro de texto . Implementaciones de las clases que heredan de Filtro

Asimismo, para comunicarse la lógica con la persistencia, usamos el patrón *Adapter*, el cual es un patrón estructural que facilita la colaboración entre interfaces. Esto nos lleva a las clases/interfaces *AdaptadorEtiqueta*, por ejemplo, que hace uso de *Abstract Factory*, un patrón creacional para crear familias de objetos. Esto es útil si queremos añadir más tipos de adaptadores. Además, usamos *Singleton* para obtener siempre la misma instancia del adaptador en cuestión. El uso de este último patrón creacional permite también tener una instancia única de los catálogos.

// Catalogo de Usuarios

private static CatalogoUsuarios instancia = new CatalogoUsuarios();

public static CatalogoUsuarios getInstancia() {

return instancia;

}

private Map<Integer, Usuario> mapaPorID;

private Map<String, Usuario> mapaPorLogin;

private AdaptadorUsuarioDAO adaptador;

private CatalogoUsuarios() {

try {

FactoriaDAO factoria = FactoriaDAO.getInstancia(FactoriaDAO.TDS\_DAO);

adaptador = factoria.getUsuarioDAO();

mapaPorID = new HashMap<Integer, Usuario>();

mapaPorLogin = new HashMap<String, Usuario>();

cargarCatalogo();

} catch (Exception e) {

e.printStackTrace();

}

}

Cuadro de texto . Uso de diversos patrones en los catálogos, concretamente, se muestra el de Usuarios

Aun así, estos son patrones que hemos usado directamente, pero también hemos usado el patrón *Experto*, que se trata de delegar tareas a otras clases, que se utiliza junto con el patrón *Controlador*. Ambos son patrones GRAPS que, en realidad, son una serie de buenas prácticas en el desarrollo software.

# Componentes

A lo largo de todo el proyecto, hemos usado componentes, partes independientes más pequeñas ya implementadas por nosotros o por otras personas.

En primer lugar, tenemos el *JCalendar* para seleccionar la fecha de nacimiento cuando hacemos el registro en AppVideo. Asimismo, hemos añadido *VideoWeb* que nos ha ayudado a reproducir los videos.

Relacionado con los videos, tenemos *Luz*, un pulsador en la parte superior de para importar los videos desde un XML. Cuando este componente es pulsado, se abre una ventana *JFileChooser* para elegir el correspondiente archivo con los videos. Después, únicamente llama al método *setArchivoVideos()* de *IBuscadorVideos* (una interfaz) pasándole como argumento el archivo seleccionado. Por último, el componente *CargadorVideos* es el encargado de guardar dichos videos y el que actualiza los catálogos.

Por otra parte, está la API de *iText* para generar el archivo PDF que contiene la información de las listas de videos en el directorio del proyecto donde se ejecuta *AppVideo*. Al contrario que el resto, hemos añadido este componente como una dependencia de Maven.

# Test unitarios

Todo programa software necesita ser probado, existe diferentes tipos de pruebas y se debería hacer en el siguiente orden: las unitarias, integradas, de sistema y aceptación. Asimismo, podemos añadir las pruebas de regresión que son aquellas que tienen la finalidad de descubrir errores.

Para las pruebas unitarias tenemos *JUnit*, un framework para automatizar las pruebas (unitarias e integradas). En nuestro caso, hemos creado una Suite de pruebas llamada *AppTest* que engloba a las clases *FiltroTest* y *UsuarioTest*. Para declarar esta suite de pruebas, debemos añadir las siguientes anotaciones:

@RunWith(Suite.class)

@SuiteClasses({UsuarioTest.class, FiltroTest.class})

public class AppTest

Estas dos son las que implementan las pruebas unitarias para las clases del dominio *Filtro* y *Usuario*.

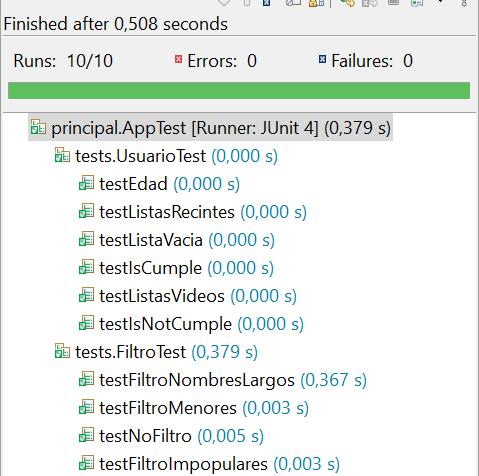


Figura . Estructura de prueba unitarias para AppVideo

En primer lugar, tenemos *Usuario*, hemos probado los diferentes métodos como *IsCumple()* con dos pruebas (una que de verdadero y otra falso) cambiando la fecha de nacimiento del usuario y *getEdad().* Por otra parte, están las pruebas relacionadas con las listas que puede tener el usuario relacionadas.

Después están las pruebas con *Filtro*, dado que es una jerarquía y hay varias clases hijas que implementan el método *esVideoOK(),* debemos probar cada una de ellas.

En conjunto, las clases *FiltroTest* y *UsuarioTest* tienen un método *init()* con la anotación *@Before* para que se ejecute antes de las pruebas unitarias, aquellas marcadas con la anotación *@Test*.

# Manual de usuario

Inicialmente cuando ejecutamos la aplicación de escritorio *AppVideo*, nos aparece una ventana con tres botones: Login, Registro y el pulsador, tal y como se muestra en la Figura 3. Para poder entrar a la funcionalidad de la aplicación en sí, debemos o bien registrarnos o logearnos si ya tenemos usuario. Por otro lado, tenemos el pulsador cuya finalidad es cargar los videos de un XML sin tener que entrar.

[FIGURA 3]

Para el caso de importar videos, se debe introducir un archivo XML con el título, la URL y las posibles etiquetas. A continuación, mostramos un ejemplo:

<?xml version=**"1.0"** ?>

<videos xmlns=**"http://www.tds.es/videos"** xmlns:xs=**"http://www.w3.org/2001/XMLSchema-instance"**

xs:schemaLocation=**"http://www.tds.es/videos videos.xsd"**>

<video titulo=**"Invasion"**>

<URL><![CDATA[https://www.youtube.com/watch?v=DlaHxL3mHAU]]></URL>

<etiqueta>**Series**</etiqueta>

<etiqueta>**AppleTV**</etiqueta>

</video>

</videos>

Cuadro de texto . Estructura del XML para subir videos a AppVideo.

En el caso de registrarse, hay que introducir diferentes datos en el formulario. Entre ellos los obligatorios son los que aparecen en junto con un \*, tal y como se muestra en la Figura 4.

[FIGURA 4]

Una vez entrado, nos aparece directamente en la ventana de Recientes, donde veremos los videos que hemos visto recientemente, si hay alguno. Sin embargo, AppVideo contiene más funcionalidad. En la Figura 5 se muestra cómo podemos buscar un video por título y/o etiquetas.

[FIGURA 5]

Además, podemos crear una nueva lista de reproducción, Figura 6.a o ver las listas que ya tenemos guardadas Figura 6.b.

[FIGURA 6a/b]

En el caso de que el usuario decida ser premium. Puede darle al botón rojo que aparece en la parte superior derecha de la pantalla y le aparecerá nuevas funcionalidades como el botón de generar un PDF con la información de sus listas de videos.

# Observaciones finales