Montaje automático de videos (Moovid)

Arquitectura e Integración de Sistemas Software

Grado de Ingeniería del Software

Curso 2º IS

Daniel Caro Olmedo (danielcaroolmedo2@gmail.com)
Nicolás De Ory Carmona (deorynicolas@gmail.com)
Antonio González Gómez (antoniogg696@gmail.com)
Ignacio Navarro Blázquez (inavarroblazquez@gmail.com)

Tutor: Javier Troya Castilla
Número de grupo:

Enlace de la aplicación: https://moovid-271019.appspot.com/
Enlace de proyecto en GitHub: https://github.com/nicolasdeory/moovid

HISTORIAL DE VERSIONES

Fecha	Versión	Detalles	Participantes
07/03/2020	1.0	- Incluye introducción, prototipos de las interfaces de usuario y diagramas UML de componentes y despliegue y secuencia de alto nivel.	Daniel Caro Nicolás De Ory Antonio González Ignacio Navarro
03/05/2020	2.0	- Añadidos los diagramas de clase y de secuencia de la arquitectura de la aplicación siguiendo las directrices del patrón MVC. - Añadida la documentación de la API REST en swagger.	Daniel Caro Nicolás De Ory Antonio González Ignacio Navarro
24/05/2020	3.0	-Añadido apartado de evolución de proyecto. -Descripciones añadidas a las vistas. -Diagramas actualizados. -Sección de pruebas añadida. -Mashup añadido.	Daniel Caro Nicolás De Ory Antonio González Ignacio Navarro

Índice

1 Introducción 4

1.1 Aplicaciones integradas 5

1.2 Evolución del proyecto 6

2 Prototipos de interfaz de usuario 7

2.1 Vista Inicio 7

2.2 Vista Creación Montaje 7

2.3 Vista Montaje Finalizado 8

3 Arquitectura 9

3.1 Diagrama de componentes 9

3.2 Diagrama de despliegue 9

3.3 Diagrama de secuencia de alto nivel 10

3.4 Diagrama de clases 10

3.5 Diagramas de secuencia 11

4 Implementación 12

5 Pruebas 18

6 Manual de usuario 23

6.1 Mashup 23

6.2 API REST 26

1 Introducción

Moovid es una aplicación web que pretende integrar Google Photos junto con Google Drive y Language Understanding Intelligent Service (LUIS – parte de los Cognitive Services de Microsoft).

El nombre es una fusión de Moo- (comúnmente usado para nombrar bots en internet) y coincidir con las primeras letras de "montaje", y -vid como abreviatura o corte de la palabra vídeo o video en inglés. Aprovechando esta última parte del nombre nace nuestro logo, una uva, puesto que es el fruto de la "Vid".

Su objetivo principal es crear de manera automática fotomontajes personalizados a partir de la biblioteca de fotos del usuario. Por medio del lenguaje natural, el usuario tiene la posibilidad de indicarle a la aplicación los parámetros que desea que tenga el montaje, desde la selección de fotos en función de su fecha y lugar de captura, hasta la selección en función del contenido de la propia imagen. Estos parámetros los indica el usuario a través de una interfaz que reconoce el lenguaje natural, gracias a LUIS.

El resultado final se almacena en la nube, dando al usuario la posibilidad de visualizarlo, descargarlo o compartirlo con total flexibilidad, gracias a la integración con Google Drive.

La aplicación tiene como objetivo facilitar en gran medida el trabajo de aquellos usuarios que quieran realizar un montaje con fotos de su biblioteca (abstrayéndoles de cualquier programa de edición), o incluso sorprenderle con recomendaciones personalizadas.

1.1 Aplicaciones integradas

LUIS. Recibe los mensajes del usuario expresados en lenguaje natural, y extrae la intencionalidad de estos (por ejemplo – confirmar o cancelar una acción, o petición para realizar un montaje), gracias a un modelo ML entrenado. Además de la intencionalidad del mensaje, LUIS extrae datos clave del mensaje como, cuando el usuario pide realizar un montaje, el rango de fechas que desea el usuario, y la temática del montaje. Asimismo, la API es capaz de discernir entre los mensajes que contienen información relevante y los que no.

Google Photos. Proporciona un análisis exhaustivo (Computer Vision) de la biblioteca de fotos del usuario, permitiendo clasificar cada foto en distintas categorías en función de su contenido, ubicación geográfica, y marca de tiempo. Se integra con LUIS para aplicar parámetros en función de lo que desee el usuario, y se utiliza además para ofrecer recomendaciones personalizadas.

Spotify. Encargado de, según las especificaciones del cliente, encontrar canciones adecuada para el montaje, esta lista de canciones será dada a la API de YouTube. Es capaz de identificar artistas o estilos de canciones (pausada, bailable, etc...) y devolver las canciones que concuerden con dichos criterios.

Youtube. Tras recibir la lista de canciones de Spotify se encargará de buscar las URLs de dichas canciones para que puedan ser añadidas al montaje final.

Nombre aplicación	URL documentación API
Google Photos	https://developers.google.com/photos
Youtube	https://developers.google.com/youtube/v3
LUIS (Language Understanding Intelligent Service)	https://www.luis.ai/home
Spotify	https://developer.spotify.com/documentation/web-api/

TABLA 1. APLICACIÓN INTEGRADAS

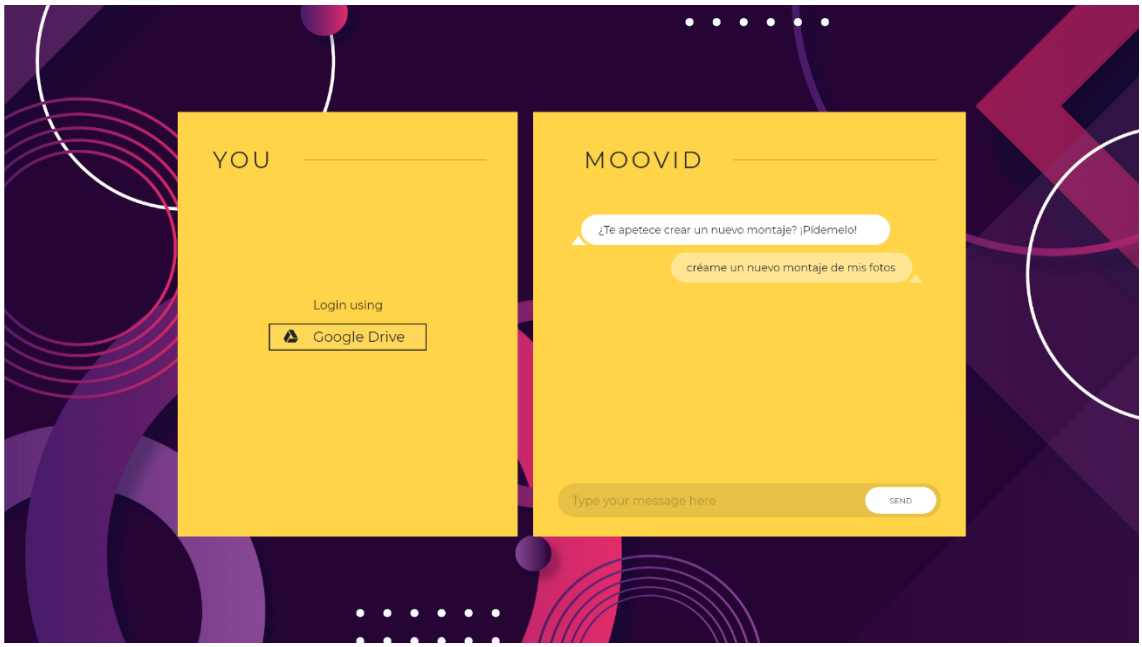
1.2 Evolución del proyecto

Nuestra idea inicial era implementar tres APIs, Google Drive, Photos y LUIS, sin embargo tras avanzar en el proyecto íbamos descubriendo que las posibilidades que nos brindaban las APIs elegidas a veces no era lo esperado.
Por ello uno de los principales cambios fue cambiar de APIs, empezando por eliminar Google Drive ya que Photos nos permitía almacenar y descargar las fotos necesarias para nuestra idea.
Pensamos que una buena API a añadir sería Spotify puesto que un montaje sin música no es un montaje, por ello decidimos implementarla, pero encontramos un problema y es que descargar música no es posible, por ello, añadimos YouTube, que junto a Spotify nos haría conseguir la canción que quisiéramos.
Para ello además usamos una librería externa donde obtenemos el enlace de YouTube y lo convertimos a mp3 desde cliente.
Para juntar todo, ha sido complicado puesto que un montaje en servidor nos era imposible, así que también decidimos hacerlo desde cliente, gracias a la librería FFMPEG , que una vez tenemos las fotos y la música se encarga de montar el vídeo.
Ha sido todo un reto hacer este mashup debido a la falta de ejemplos parecidos, y la novedad de una API para reconocer el lenguaje humano.

2 Prototipos de interfaz de usuario

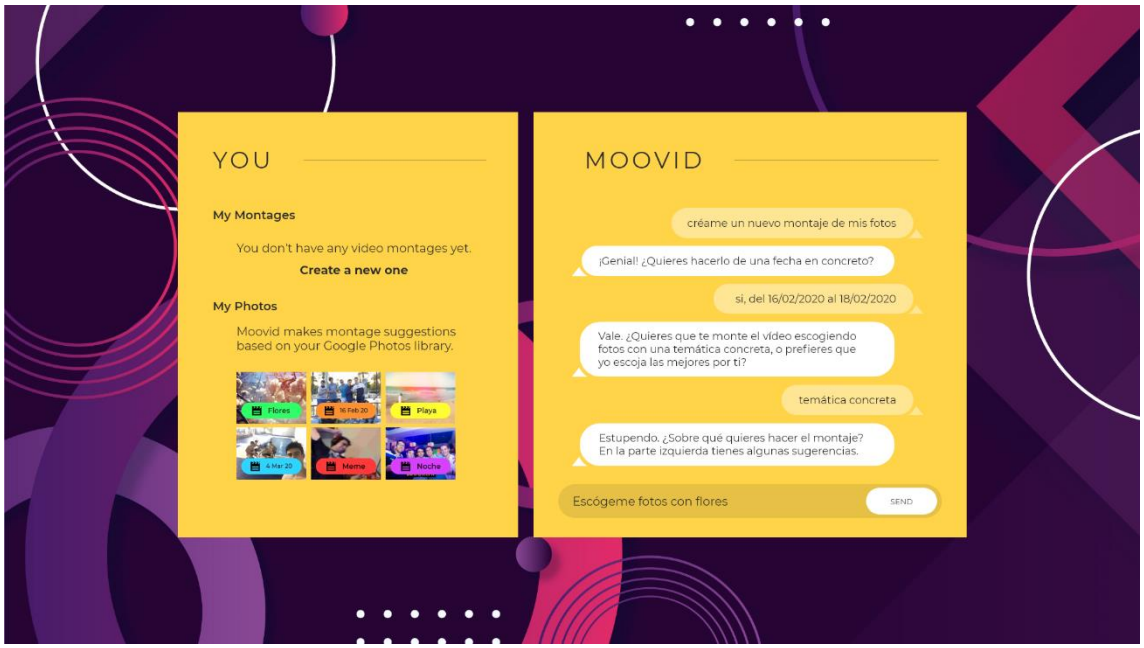
2.1 Vista Inicio

Esta vista esta disponible antes de hacer login con Google Photos, se muestra un icono para iniciar sesión y la ventana de conversación con LUIS donde se escribirán las diferentes instrucciones



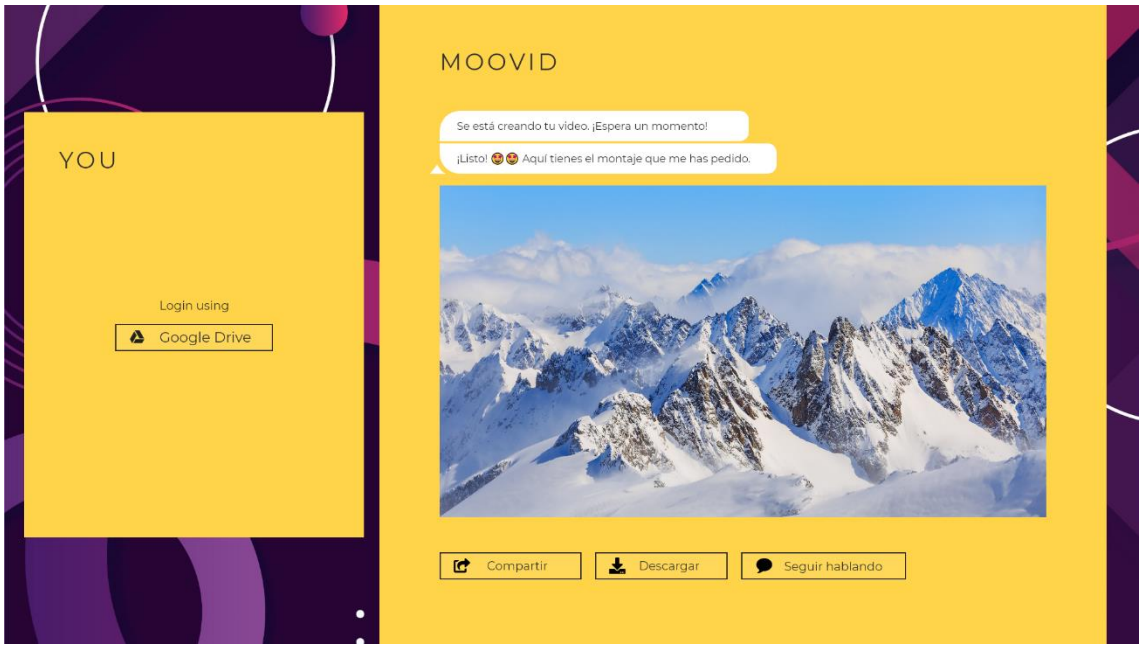
2.2 Vista Creación Montaje

Se accederá a esta vista una vez el usuario se haya registrado con Google, aparecerán los montajes que se han procesado y las fotos disponibles divididas en categorías. La conversación con LUIS seguirá estando disponible.



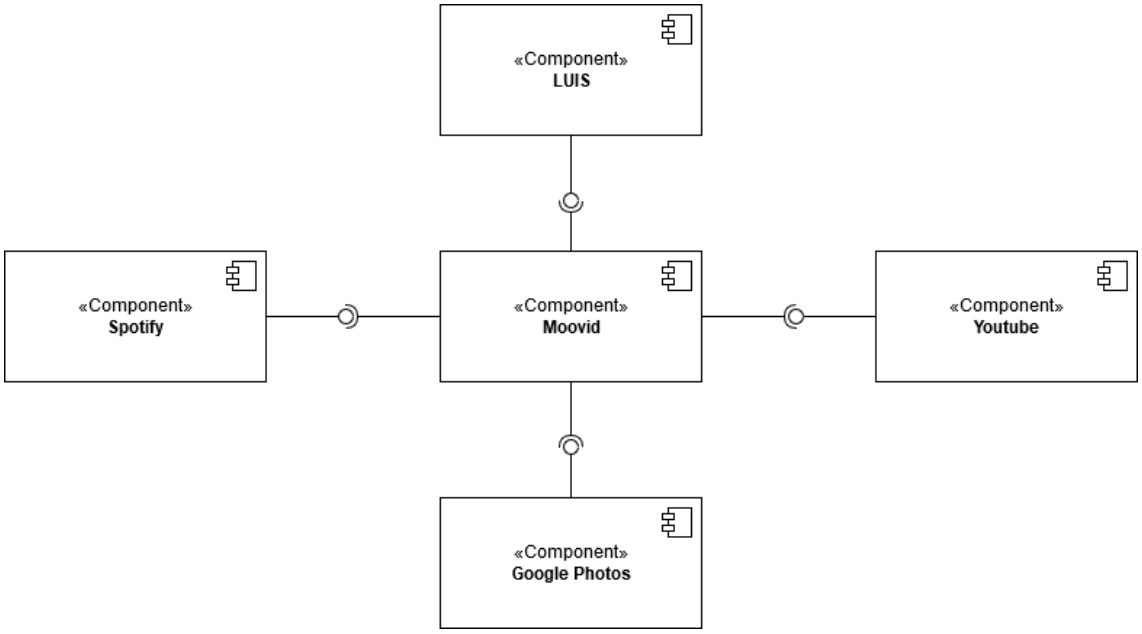
2.3 Vista Montaje Finalizado

Una vez se acabe de procesar un montaje LUIS devolverá un mensaje donde podremos ver una previsualización y algunos enlaces para descargarlo o compartirlo en distintas redes sociales.

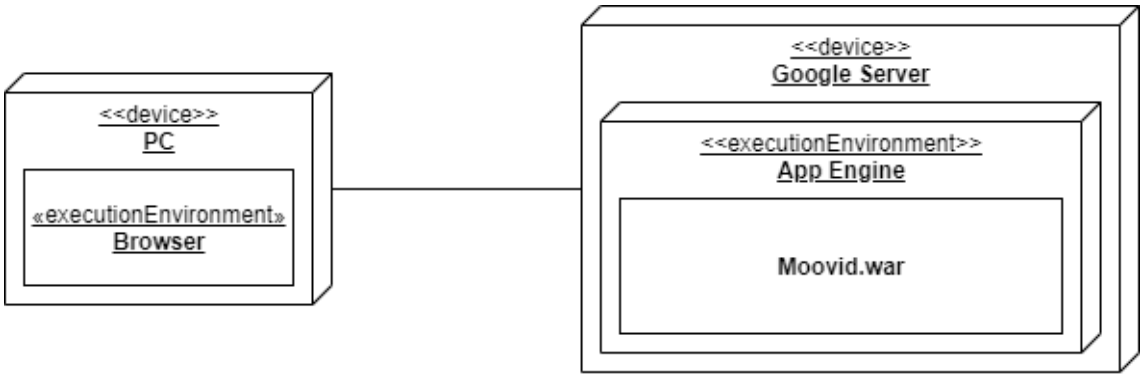


3 Arquitectura

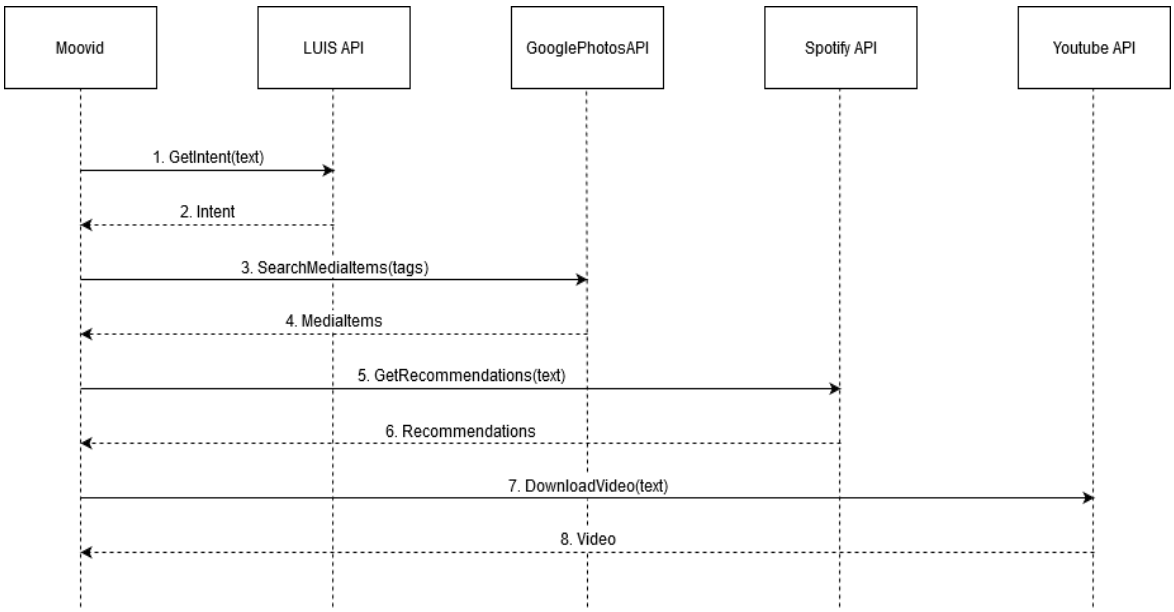
3.1 Diagrama de componentes



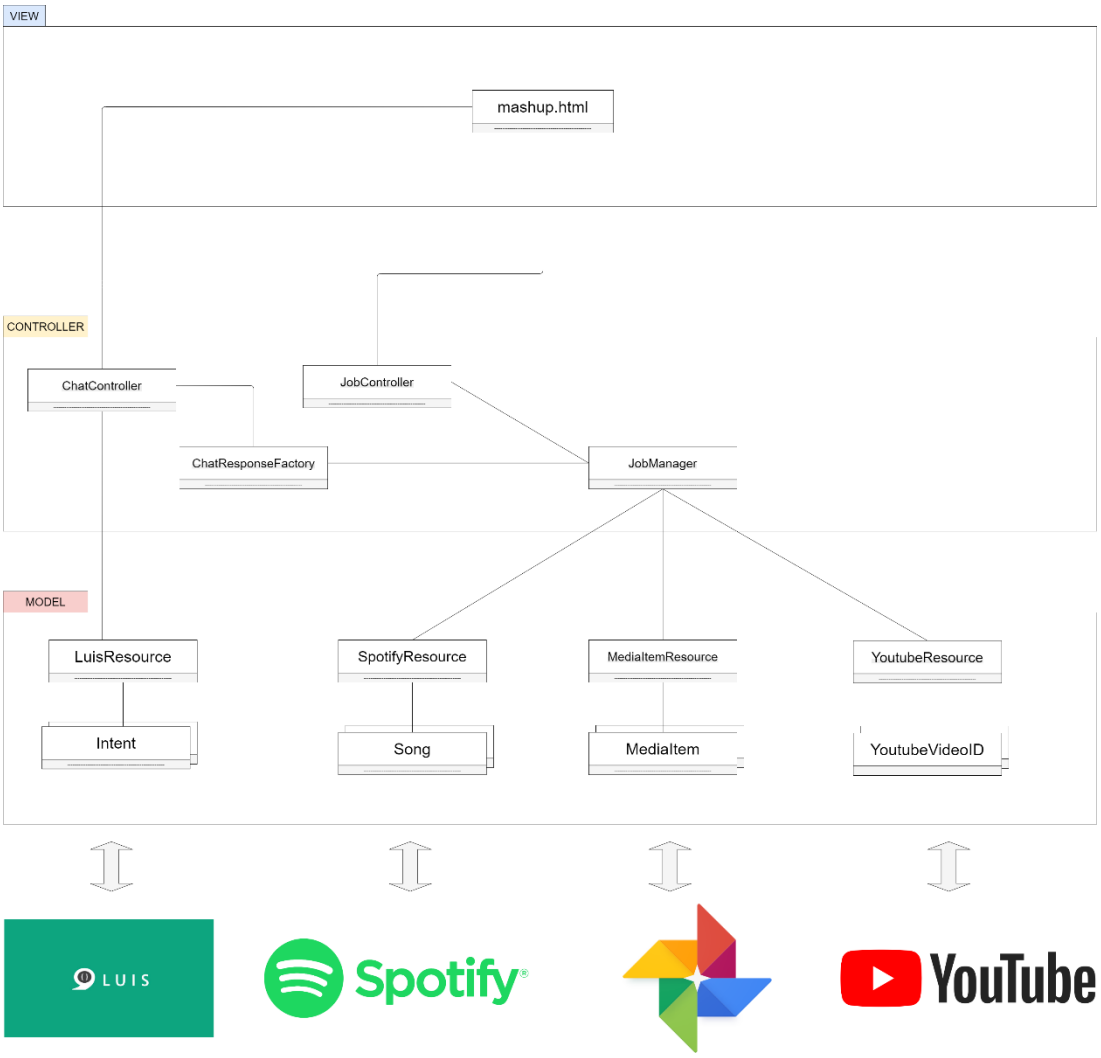
3.2 Diagrama de despliegue



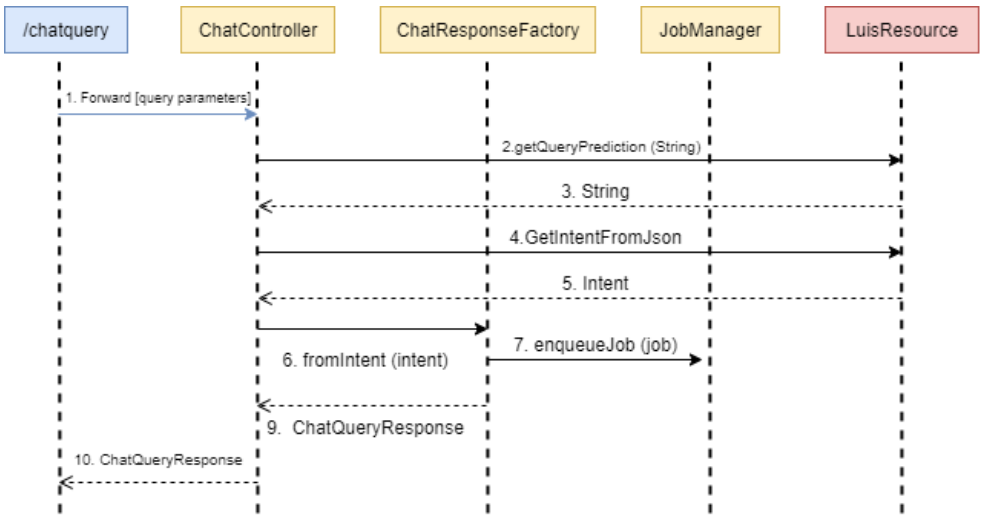
3.3 Diagrama de secuencia de alto nivel

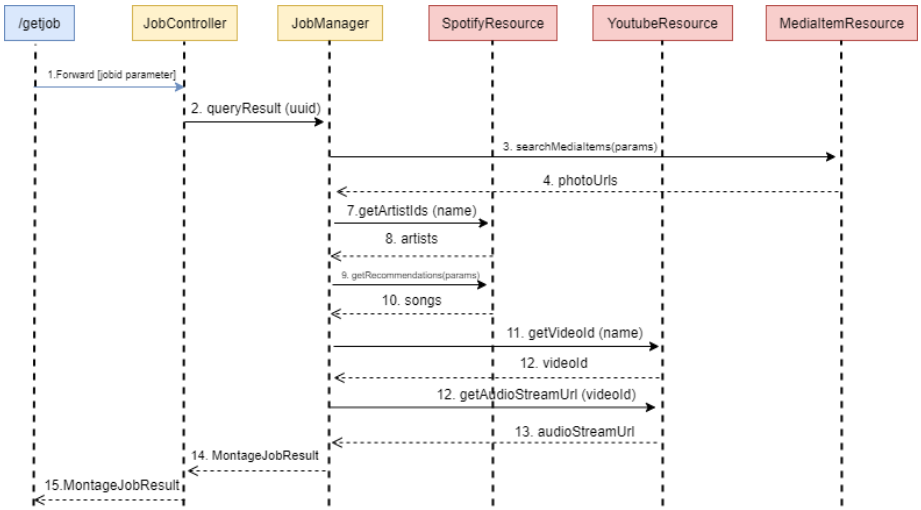


3.4 Diagrama de clases



3.5 Diagramas de secuencia





4 Implementación

La implementación del proyecto está dividida en varias secciones significativas, como se detalla en el diagrama de clases.

La mayoría de la interacción del usuario con la aplicación se realiza a través de una interfaz de chat que permite la comunicación con lenguaje natural.

Gracias a Cognitive Services, podemos extraer la intencionalidad del usuario y extraer entidades de interés, como fechas, temas sobre lo que hacer el montaje, el género musical y artistas, entre otros.

Sin embargo, este servicio no nos permite tener constancia del contexto de la conversación con el usuario, el cual es imprescindible para poder mantener una conversación fluida con el chatbot.

Para ello, el servidor procesa el contexto de la conversación y produce una respuesta acorde a él, por medio de *ChatResponseFactory*. El nuevo contexto de la conversación se almacena en una variable de sesión de corta duración, por si el usuario actualiza momentáneamente la página. Este contexto permite que frases como “decide por mí” sean interpretadas correctamente y la aplicación entienda que el usuario se refiere a elegir el tema del montaje, o bien la música de este. Asimismo, el contexto permite a Moovid no olvidar las fechas y datos que se le hayan dicho con anterioridad, y los tenga en cuenta a la hora de hacer finalmente el montaje.

Respecto a las respuestas, la clase que se describe en el párrafo anterior se encarga de buscar la respuesta más apropiada para la situación. Una vez la encuentre, la transforma a una frase natural en el idioma correspondiente. Esto se realiza llamando a *ChatResponseSupplier*, que es una clase que mantiene en memoria un diccionario que tiene como claves el identificador de respuesta y como valores una lista de posibles respuestas. Las respuestas son inicialmente cargadas desde un archivo XML con el idioma especificado. La clase, a continuación, escoge una aleatoria para aportar variedad a la conversación. Cabe mencionar que una respuesta se puede componer de uno o varios mensajes, aunque esto es algo puramente visual para estructurar de manera más natural las contestaciones.

Una vez que se ha llegado al punto de la conversación en el que el servidor tiene todos los datos que considera necesarios para hacer el montaje (incluso si el usuario ha omitido algunos), envía una respuesta especial al cliente indicándole que se ha creado una carga de trabajo para recabar todos los elementos que se necesitan para montar el vídeo.

El sistema de cargas de trabajo se centraliza en la clase *JobManager*, y es quien se encarga de hacer todas las llamadas al modelo para obtener los datos que considere oportunos.

Inicialmente, la intención era procesar todas las cargas de trabajo que iba recibiendo de distintos usuarios en un hilo dedicado (*background thread*), de manera que el cliente podía ir haciendo *polling* cada poco tiempo para comprobar si el servidor había terminado procesando su petición.

Sin embargo, la modalidad de AppEngine de Google Cloud no permite hilos en la modalidad estándar de coste gratuito.

Por lo tanto, se optó por ejecutar la carga de trabajo en el mismo hilo que la petición del usuario. Evidentemente, como observamos que esto podía ralentizar mucho el mensaje de respuesta de "Por favor, espera mientras te realizamos el montaje…", optamos por enviar una primera respuesta con la carga de trabajo creada, como se indica en el párrafo anterior, pero sin procesar nada.

El cliente, por medio de *JobController*, solicita a Moovid que se procese la carga de trabajo. Como esta petición sí puede durar más de lo habitual, se realiza asíncronamente en cliente. No es un enfoque ideal ya que puede incurrir en problemas de timeout si la carga de trabajo tarda en completarse más de lo esperado, pero es la única opción ante la imposibilidad de utilizar otros hilos en la infraestructura de despliegue.

La carga de trabajo, en esencia, consiste en hacer llamadas a las APIs necesarias para recabar los datos necesarios para el montaje.

En primer lugar, se invoca a *Google Photos* para hacer una búsqueda de fotos en la librería de la nube del usuario. Esta búsqueda se realiza acorde a parámetros que se han especificado durante la conversación, como fechas o temáticas.

En principio, se procesan todas las fotos que se encuentren, incluso si son más que el límite de 100 fotos que hemos considerado para hacer un montaje de una duración apropiada y no excesivamente largo. Si se encuentran más, se realizará una criba en cliente.

A continuación, se hacen varias llamadas a Spotify para obtener las recomendaciones de canciones acorde a los parámetros del usuario, que pueden variar desde el género u autor de la canción, hasta la velocidad, el estado de ánimo, la presencia o no de letras, o incluso si es una canción es “bailable”.

El resultado es una lista de canciones que Spotify recomienda para el montaje, de las cuales se escogen tres (en vez de una, por un motivo que se explica abajo).

Sobre las canciones que se obtienen, se buscan por título en YouTube para obtener un enlace de vídeo. Una vez que se recupera el identificador de vídeo, se utiliza una librería Java que extrae el enlace del stream de audio del vídeo (nos interesa únicamente la pista de audio, de cara al montaje).

Una vez termina esta petición, se almacena el resultado de la carga de trabajo en caché y se devuelve al cliente, que durante todo este tiempo había estado esperando.

La respuesta contiene las URLs de las fotografías y de las pistas de audio.

Por último, queda el procesamiento final del montaje en cliente. Durante todo este tiempo, el usuario es informado de cómo va avanzando el proceso por medio de una barra de progreso.

El cliente, a partir de las URLs, debe recibir los blobs correspondientes a las imágenes y pistas de audio. Se puede hacer a partir de un *fetch* o cualquier sistema de peticiones asíncronas en Javascript. Sin embargo, existe un problema a la hora de recibir los blobs de audio.

Dado que extraer los streams de audio de un vídeo de YouTube está en contra de los términos del servicio, la manera de hacerlo no es oficial, ni está documentada, por lo que está sujeta a cambios y medidas que puedan evitar obtener de manera programática el stream. Una de las trabas que se encuentran es que, al tratarse de una petición cross-domain que se realiza desde el cliente, por medio de Javascript, el navegador bloquea en condiciones normales la respuesta, por motivos de seguridad, ya que carece de cabeceras CORS.

La manera ideal de descargar los blobs sería descargarlos y almacenarlos en servidor, de manera que el cliente solo tendría que hacer peticiones en el mismo dominio. Sin embargo, realizar las peticiones desde servidor es muy lento, dada la modalidad gratuita de la infraestructura de despliegue.

Por lo tanto, el método por el que hemos optado es hacer uso de un CORS proxy, para eludir las restricciones de seguridad inherentes a los navegadores. Hemos utilizado *CORS Anywhere*, proxy conocido de código abierto.

Este proxy incluye cabeceras en la respuesta para que el navegador pueda aceptar las respuestas sin problema, aunque supone una penalización de rendimiento.

Sin embargo, aún con estas medidas, la solicitud a la URL del stream de audio de YouTube puede fallar en algunas ocasiones, ya sea por el proxy o porque YouTube bloquee la petición.

Por tanto, como se menciona anteriormente, se pasan direcciones de varios streams de audio, para que haya alternativa en caso de fallo.

Una vez ya se hayan descargado todos los blobs, se puede realizar el montaje.

Inicialmente, se iba a realizar el montaje en servidor para ahorrar recursos al cliente y evitar que se tuviese que descargar todas las fotos previamente a montar el vídeo (a pesar de que están optimizadas para que tengan un consumo bajo de datos).

La solución más completa e intuitiva es utilizar un ejecutable de código abierto llamado FFMPEG. Permite transcodificar una inmensa variedad de formatos de vídeo, imagen y audio y aporta una gran libertad para generar vídeos. Es una utilidad de línea de comandos, pero existen wrappers en Java que facilitan la interacción. Sin embargo, esto no ha sido factible debido a que AppEngine, por naturaleza, no soporta la ejecución de código nativo o ejecutables en su entorno.

Por lo tanto, la única opción sería utilizar una librería de transcodificación que esté escrita en Java puro. Dado que un caso de uso muy poco común, las librerías que existen para ello están muy poco documentadas, son poco intuitivas, el rendimiento es nefasto y la codificación de vídeo es de muy baja calidad.

La mejor opción que hemos encontrado, por lo tanto, es hacer el procesamiento del montaje en cliente. Existe un proyecto de código abierto que, utilizando *Emscripten*, compila FFMPEG a código Javascript que puede ser ejecutado por el navegador. Esto permite ejecutar el programa desde cualquier plataforma que soporte Javascript, incluido móvil.

FFMPEG.js utiliza el estándar de *Web Workers* para poder ejecutarse de manera asíncrona y comunicarse por un sistema de mensajes. Con estos mensajes, le pasamos los datos binarios de todos los archivos que queremos utilizar, y los monta en un sistema de archivos virtual *WORKERFS* que, a ojos de FFMPEG, es como si se ejecutase en una máquina con un sistema de archivos montado que contiene los archivos a incluir en el montaje.

Cuando en Javascript tenemos un blob de una imagen, para el programa es como si la imagen estuviese en */img/imagen1.jpg*.

Como recordamos, FFMPEG en su medio original es una utilidad de línea de comandos, por lo que los archivos que debemos utilizar se pasan por argumentos.

La librería tiene una vasta documentación con todos los argumentos posibles que se pueden incluir, permitiendo una flexibilidad ilimitada a la hora de codificar vídeo, imagen o audio. Tiene un pipeline de filtros llamado *Filtergraph* que permite encadenar filtros y ajustar los parámetros de trozos determinados de vídeo. Existen tantos filtros, que incluirlos todos haría que el tamaño del ejecutable fuese excesivamente grande (lo cual, al pasarlo a Javascript, causaría que el tiempo de evaluación fuese muy elevado). Para ello, se compila FFMPEG (FFMPEG.js en nuestro caso) con los filtros que queramos incluir para nuestro caso de uso. Por lo tanto, hemos compilado un FFMPEG.js que se ajusta a las necesidades de Moovid.

Hasta el momento, toda la infraestructura funciona correctamente. Sin embargo, existen varios problemas al utilizar la librería compilada en Javascript, sobre los que hemos tenido que trabajar.

En primer lugar, el rendimiento no es ideal. FFMPEG se aprovecha enormemente del multi-threading, pero esto no es posible con Javascript, ya que el motor de evaluación (V8) se ejecuta en un solo hilo.

En segundo lugar, este método poco ortodoxo para compilar LLVM a Javascript, si bien fue revolucionario hace unos años, ha quedado un poco anticuado y no es muy soportado por los navegadores. Los programas compilados con *Emscripten*, además, tiene unos límites de memoria extremadamente reducidos. Hemos encontrado que FFMPEG, que no está muy optimizado para reducir al máximo el uso de memoria (lo cual es lógico, porque está diseñado en ser ejecutado en una máquina relativamente potente), interrumpe inesperadamente el montaje si se queda sin memoria.

En total, FFMPEG tiene como máximo 64MB para funcionar. Por lo tanto, si las imágenes superan una determinada resolución o tamaño, FFMPEG se queda sin memoria y es incapaz de codificar el vídeo.

Por lo tanto, ha habido que hacer un esfuerzo muy elevado para optimizar al máximo el proceso de montaje, utilizando filtros muy simplificados, estrategias de codificación que consuman muy poca memoria (son más rápidas pero el resultado no es de muy alta calidad, pero es apropiado para nuestro caso de uso), y encontrando un equilibrio entre bitrate y calidad de vídeo y audio. Asimismo, se ha ajustado al máximo el límite y resolución de las imágenes, teniendo en cuenta que la pista de audio es el archivo que más espacio ocupa.

Respecto a las imágenes, desde el servidor garantizamos que las imágenes que se obtienen desde *Google Photos* tienen una dimensión concreta y no superan un determinado tamaño.

Para evitar estas limitaciones tan ajustadas de memoria, la solución ideal sería compilar FFMPEG a WebAssembly, un estándar relativamente nuevo en navegadores que les permite ejecutar código nativo, a rendimiento casi nativo, sin pasar por el evaluador de Javascript, y permitiendo una flexibilidad absoluta. Los navegadores están incluyendo muchas optimizaciones para aumentar la velocidad a la que se ejecuta WebAssembly, por lo que esto aumentaría enormemente el rendimiento de FFMPEG, ya que este podría hacer uso de multi-threading sin necesidad de usar núcleos virtuales. Además, no habría tantas limitaciones de memoria (aunque no sería infinita, una pestaña de navegador tiene una cantidad finita de memoria que puede tomar), por lo que no habría que sacrificar calidad para optimizar la ejecución.

Esta migración de FFMPEG.js a WebAssembly es un proyecto que está en curso y está activo en el repositorio del proyecto, pero no está preparada para entornos de producción y no hace uso de multi-threading.

Como conclusión, Moovid es el resultado de analizar todos los problemas y complicaciones que surgen a partir de una idea, y de explorar una gran variedad de opciones y los nuevos estándares actuales en diversas áreas.

Hemos tenido que realizar un aprendizaje de Voice UX, un tipo de experiencia de usuario que ha aparecido hace muy poco tiempo con sistemas como Alexa o Google Home.

También hemos aprendido mucho de las limitaciones de los despliegues en Google App Engine, y las limitaciones de la ejecución de código en Java.

Por último, Moovid nos ha aportado una gran perspectiva sobre el proceso de codificación de vídeos y los avances más actuales en materia de ejecución de código nativo en navegadores.

5 Pruebas

Hemos usado integración descendente en profundidad integrando cada rama por separado debido a que cada una tiene una funcionalidad especifica. Por ejemplo, LUIS no depende de Spotify. Esto nos ha sido una gran ventaja debido a que podíamos probar puntos de control específicos sin necesitar que el mashup estuviera 100% funcional, adicionalmente podíamos probar la funcionalidad completa de cada API sin necesitar de las otras.

Resumen	
Número total de pruebas realizadas	14
Número de pruebas automatizadas	11 (78.5%)

ID	**Generador de String filtro**
Descripción	Prueba de conversión en Photos de parámetros a filtro en forma de string.
Entrada	Entran Strings que simbolizan los parámetros pertinentes en el filtro.
Salida esperada	Se deberán parsear los parámetros individualmente y después devolver un String preparado para ser parseado como Filters.
Resultado	**EXITO**
Automatizada	Sí

ID	**Generador de Filtro**
Descripción	Prueba de conversión de parámetros a filtro en Photos
Entrada	Entran Strings que simbolizan los parámetros pertinentes en el filtro.
Salida esperada	Tras parsear los parámetros se obtendrá el filtro en formato string y usando dicho string se formara un objeto Filters.
Resultado	**EXITO**
Automatizada	Sí

ID	**Generador de Filtro Vacio**
Descripción	Creación de filtro en Photos en caso de que no se pase ningún parámetro
Entrada	Los parámetros serán Strings vacíos.
Salida esperada	Se generará un objeto tipo Filters con sus valores mínimos
Resultado	**EXITO**
Automatizada	Sí

ID	**testGetVideoID**
Descripción	Prueba de obtención correcta de la id de un video de Youtube dada una query.
Entrada	Parámetro query que identifica el video.
Salida esperada	Id del video seleccionado.
Resultado	**EXITO**
Automatizada	Sí

ID	**testDownloadVideo**
Descripción	Comprueba que se obtiene un URL de descarga válido a partir de la id de un video.
Entrada	Id que identifica el vídeo.
Salida esperada	URL de descarga.
Resultado	**EXITO**
Automatizada	Sí

ID	**testGetArtistsID**
Descripción	Prueba de la obtención de artistas dentro de la API de Spotify.
Entrada	Se proporcionan las querys asociadas a los artistas que se testean.
Salida esperada	Se devuelve la lista de los artistas (objeto Artist) solicitada.
Resultado	**EXITO**
Automatizada	Sí

ID	**testGetRecommendations**
Descripción	Obtiene una lista de canciones recomendadas a partir de ciertos parámetros de búsqueda.
Entrada	Una lista de artistas (Artist), otra de géneros (String), y una serie de propiedades (String) como el tempo de la canción o si es instrumental.
Salida esperada	Una lista de canciones (Song) recomendadas.
Resultado	**EXITO**
Automatizada	Sí

ID	**testGetBPM**
Descripción	Obtiene el BPM (Beats Per Minute) de una canción.
Entrada	Id de Spotify asociado a una canción.
Salida esperada	El BPM de la canción (Double).
Resultado	**EXITO**
Automatizada	Sí

ID	**testGetRandomMessage**
Descripción	Comprobación de obtención de mensaje.
Entrada	Se pide un mensaje de despedida a la API LUIS.
Salida esperada	Se muestra en pantalla el mensaje devuelto.
Resultado	**EXITO**
Automatizada	Sí

ID	**testParseXMLDoc**
Descripción	Comprobar el parseo de un documento XML a algo legible por el componente de conversacional.
Entrada	Documento en formato XML.
Salida esperada	Se muestra en pantalla el parseo del archivo en el formato deseado
Resultado	**EXITO**
Automatizada	Sí

ID	**testIntentFromQuery**
Descripción	Comprobación de obtención correcta de un objeto Intent a partir de un mensaje enviado a la API LUIS.
Entrada	Mensajes escritos en String.
Salida esperada	Objetos Intent con las propiedades especificadas en los mensajes.
Resultado	**EXITO**
Automatizada	Sí

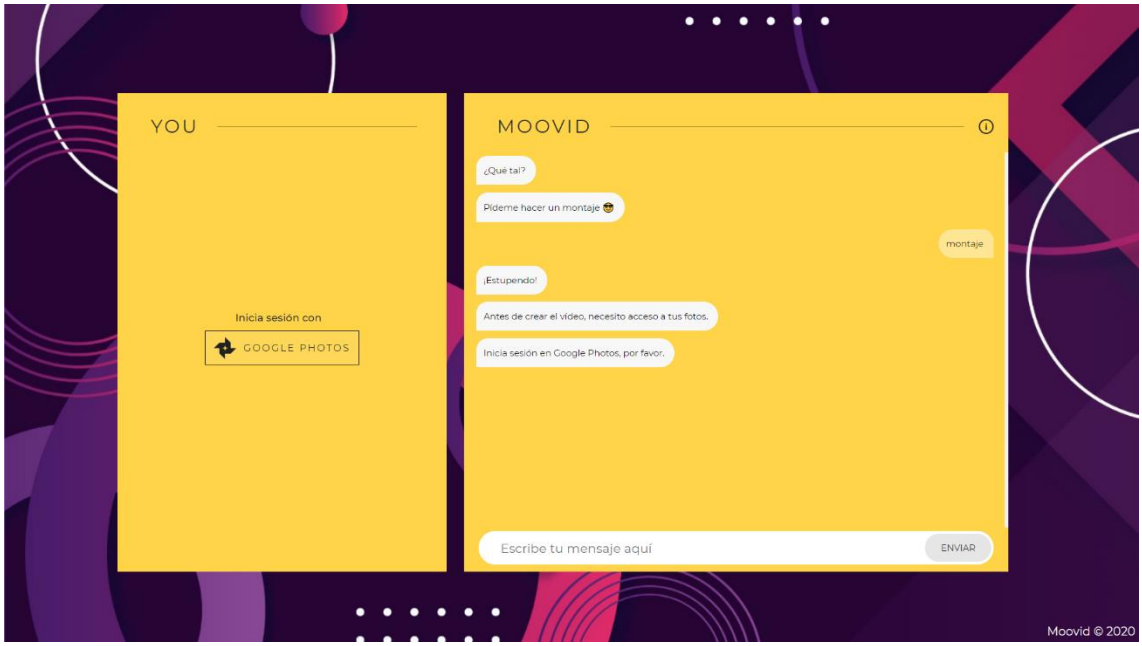
ID	**pruebaIntegración1**
Tipo	Integración
Descripción	Comprobación de que Chat Controller utiliza bien los datos recibidos por el LUIS.
Entrada	Query Montaje, ContextType, MontageTheme.
Salida esperada	.CreateMontage, Response.
Resultado	**EXITO**
Automatizada	No

ID	**pruebaIntegración2**
Tipo	Integración
Descripción	Prueba de JobManager y MediaItemResource.
Entrada	Se simula una llamada con los parámetros necesarios. Query Personas.
Salida esperada	Se espera mensaje afirmativa y pase a MontageMusic.
Resultado	**EXITO**
Automatizada	No

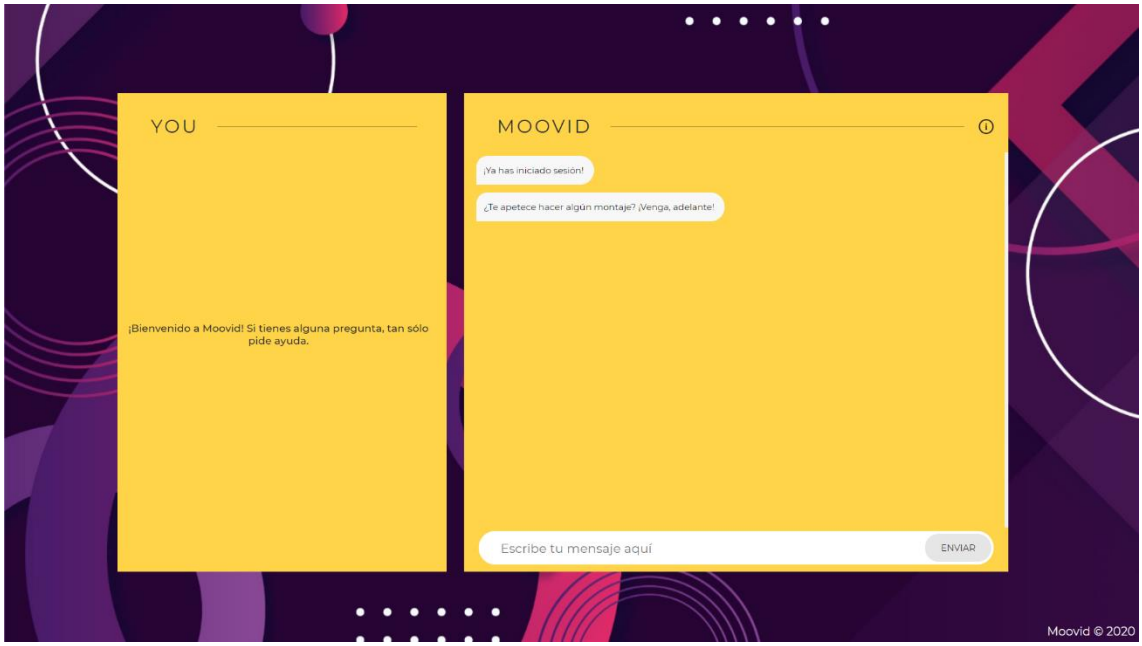
ID	**pruebaIntegración3**
Tipo	Integración
Descripción	Probamos de nuevo JobManager con Youtube Resource y Spotify Resource comprobando que devuelve una canción de manera correcta.
Entrada	Query Canción de Avicci.
Salida esperada	JobId, mensajes de montaje de video y llamada a VideoGeneration.
Resultado	**EXITO**
Automatizada	No

6 Manual de usuario

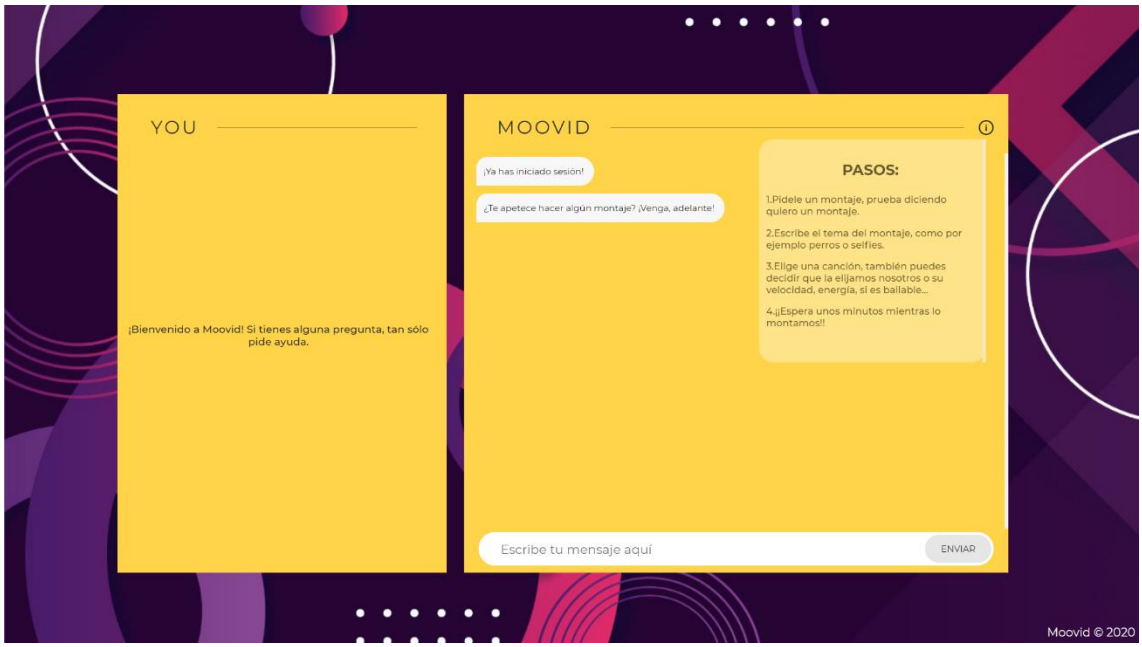
6.1 Mashup



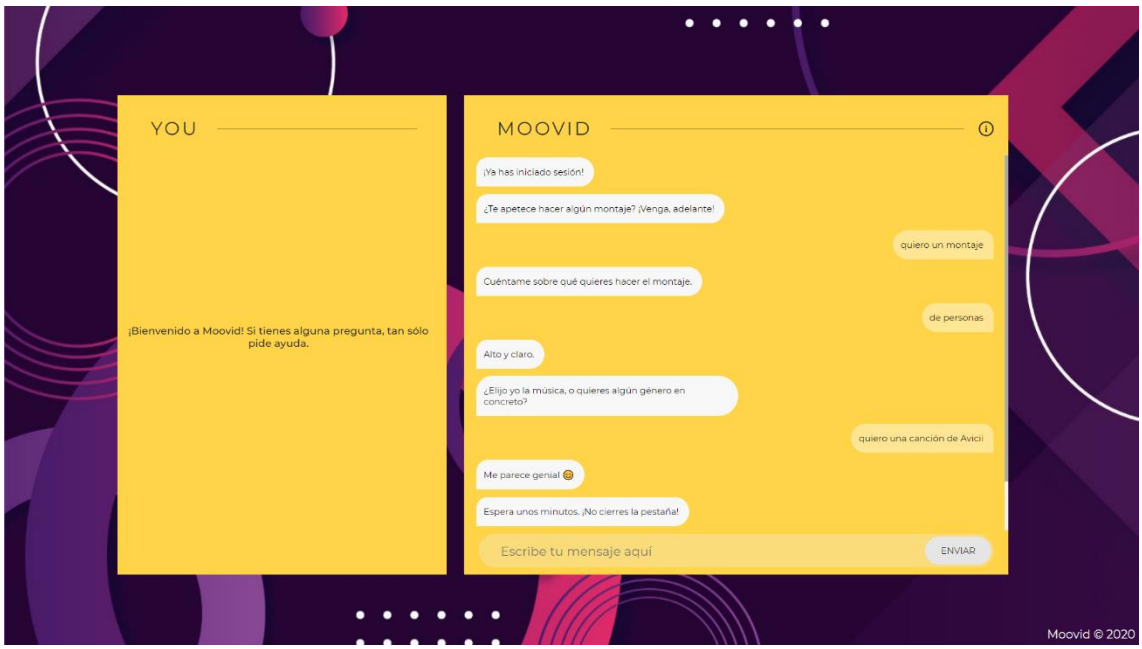
En la primera vista se nos mostrará el chat con LUIS y el acceso a Google Photos, por lo que para poder empezar a hacer un montaje necesitaremos antes acceder a nuestra cuenta de Google.



Una vez hayamos accedido LUIS nos avisará con un mensaje que todo esta listo.

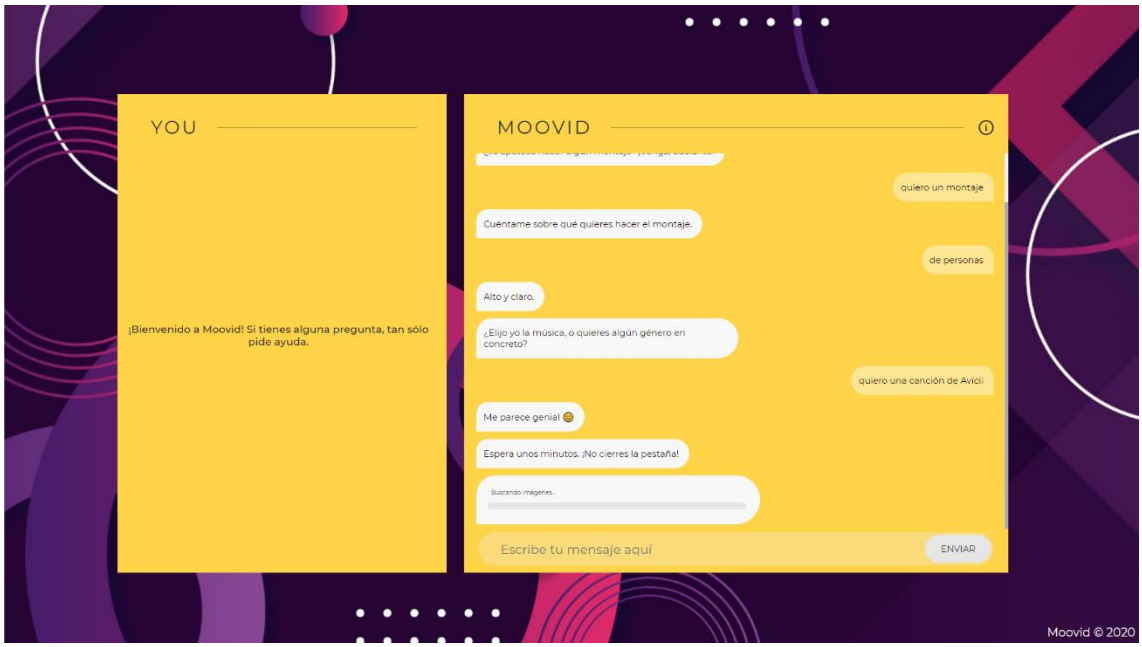


También contamos con un botón de información arriba a la derecha, presionando en el saldrá un cuadro emergente detallando los pasos a los usuarios.



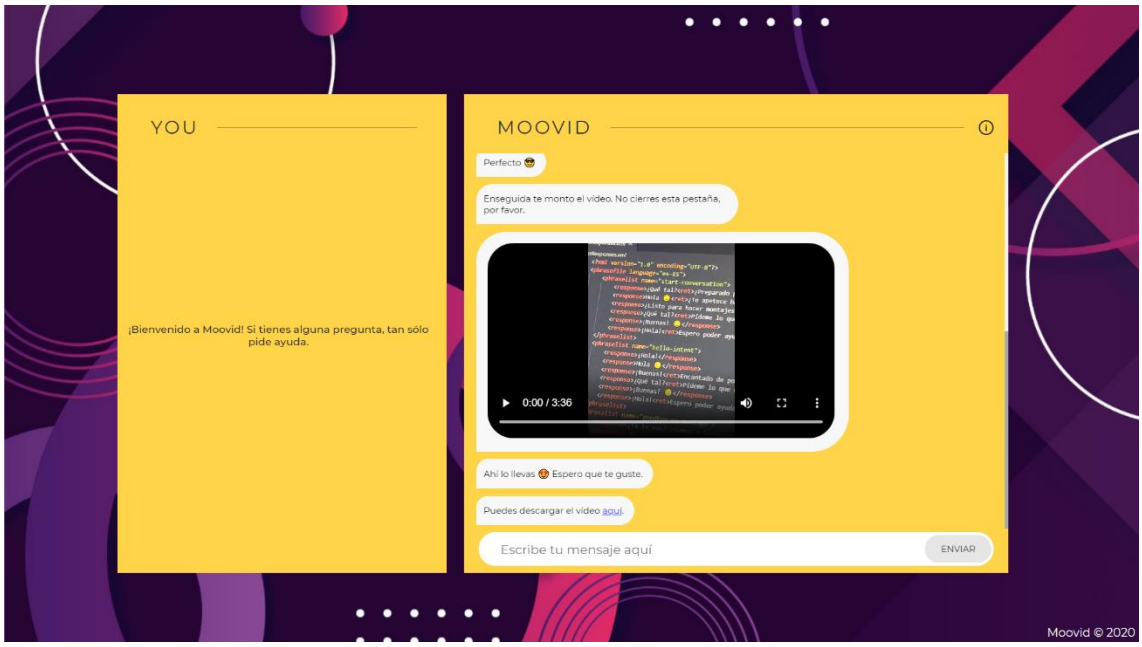
Una vez iniciado la sesión con Google photos, podremos indicarle que nos haga un montaje.
Nos preguntara que queremos incluir en el montaje y podremos escoger el tema que queramos, algunos posibles son personas, fotos, comidas, viajes, paisajes, y un largo etcétera, gracias a LUIS, que previamente ha sido entrenado en una variedad de temas.

También nos preguntará si queremos alguna música en específico en cuyo caso podemos decirle un estilo musical o un artista para que obtenga recomendaciones basadas en tu petición o dejar que decida él.



Tras la petición de música, el vídeo comenzará a montarse y el usuario no podrá enviar mensajes.
Primero buscará las imágenes sobre el tema pedido, tras ello comenzará a descargarlas al igual que con la música.

Una vez tenga todo, comenzará el montaje que aproximadamente durará entre uno y tres minutos, dependiendo del número de fotos que haya cogido.



Una vez la barra ha cargado y no ha ocurrido ningún error, te devuelve un mensaje con el vídeo, para poder verlo en web y si es de tu agrado, pulsando el mensaje de abajo puedes descargarlo directamente al ordenador o dispositivo móvil.

6.2 API REST

Documentación interactiva: https://moovid-271019.appspot.com/docs/index.html

Fichero YAML: https://app.swaggerhub.com/apis/IgnaxL/Moovid/2.0.0#/