GRADO EN INGENIERÍA EN SISTEMAS AUDIOVISUALES Y MULTIMEDIA

Curso Académico 2014 / 2015

Trabajo Fin de Grado

AGRUPACIÓN DE SERVICIOS DE ALMACENAMIENTO VIRTUAL

Autor: Pablo Parejo Camacho

Tutor: Dr. Gregorio Robles

Proyecto Fin de Carrera

AGRUPACIÓN DE SERVICIOS DE ALMACENAMIENTO VIRTUAL

Autor : Pablo Parejo Camacho

Tutor : Dr. Gregorio Robles Martínez

La defensa del presente Proyecto Fin de Carrera se realizó el día de de 2015, siendo calificada por el siguiente tribunal:

Presidente:

Secretario:

Vocal:

y habiendo obtenido la siguiente calificación:

Calificación:

Fuenlabrada, a de de 2015

He fallado más de 9000 tiros en mi carrera.

He perdido casi 300 partidos.

26 veces han confiado en mí para lanzar el tiro que ganaba el partido y lo he fallado.

He fracasado una y otra vez en mi vida y eso es por lo que he tenido éxito

Michael Jordan.

A mis padres y mi hermano por su apoyo incondicional, gracias por estar siempre ahí cuando lo he necesitado. Sois una familia increible.

A Elena por tener siempre una sonrisa para mí y hacer pequeña la distancia.

A mis Hoolisamers por los momentos de estos 4 años, y los que quedan.

Resumen

En este proyecto se pretende realizar una aplicación web capaz de agrupar los servicios de almacenamiento en la nube, así como el despliegue en un servidor para realizar las pruebas de uso reales.

Esta integración será transparente de cara al usuario, para permitir un uso más sencillo de la herramienta y poder llegar a ser una alternativa a tener en cuenta respecto a los servicios actuales.

No hemos querido dejar de lado la experiencia de usuario y el diseño de una buena interfaz para la herramienta. No en vano, conforme avanzaba el proyecto, hemos ido cambiando varios aspectos de la misma.

Por otro lado, en este trabajo se opta por utilizar las tecnologías más punteras en cuanto a diseño de webapps; tanto en el desarrollo del servidor como en la parte cliente, haciendo uso de librerías y frameworks que nos permitirán abstraernos de problemas básicos y centrarnos en las características concretas de nuestra aplicación.

De esta manera, hemos realizado una investigación y valoración de las distintas herramientas disponibles para mejorar el desarrollo y la estructura de la aplicación. Además, para realizar la integración se han utilizado los recursos disponibles en los portales de desarrolladores de las plataformas de almacenamiento en línea, donde hemos indagado para completar los objetivos de este proyecto con éxito

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Motivación

Tras la explosión en el mercado de los *Smartphone* y *Tablets*, el aumento en el uso de servicios de almacenamiento online ha ido en aumento. En parte por la poca capacidad de estos productos; pero, sobre todo, por la necesidad del usuario de mantener sincronizados sus archivos entre todos sus dispositivos.

En el mercado, podemos encontrar tres soluciones con uso mayoritario: *Dropbox*, *Google Drive* y *OneDrive*.

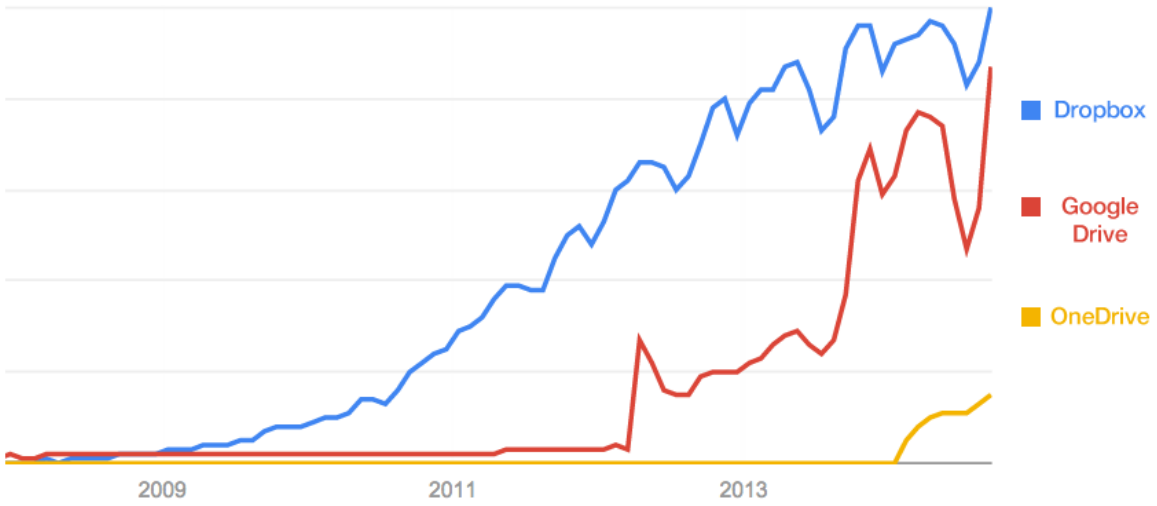


Figura 1.1: Análisis en Google Trends de los 3 principales servicios de almacenamiento. Elaboración propia

Dropbox se presenta como el dominante, superó la barrera de los 300 millones de usuarios en Mayo del 2014, además, según la revista Fortune[1], es el servicio más utilizado con diferencia. El segundo lugar pertenece a Google Drive, que en el mes Septiembre de 2014 llegó a los 240 millones de usuarios **activos**. Por último, OneDrive, la solución de Microsoft, alcanzó 250 registrados en Junio de 2014.

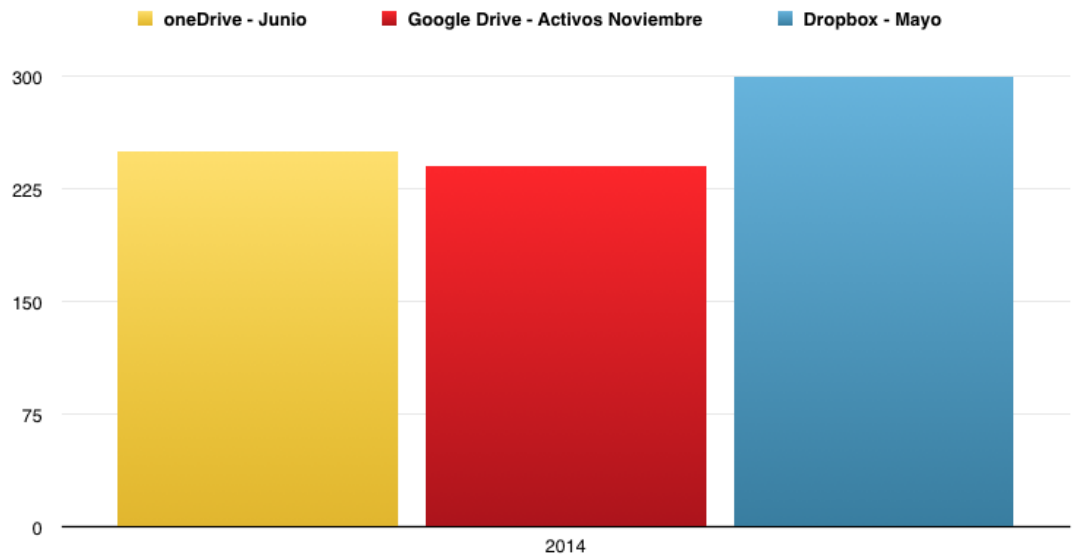


Figura 1.2: Usuarios registrados de OneDrive[2] , Dropbox[3] y activos de Google Drive[4]. Elaboración propia

Como se puede observar, la gran parte de los usuarios se distribuyen entre *Dropbox* y *Google Drive*, quienes se estima que tienen más del 70 % del mercado, teniendo en cuenta que los datos recopilados son de fechas distintas, relegando a oneDrive a una tercera posición bastante atrasada.

Con esta situación en cuanto al uso, y con un tiempo reducido en cuanto a desarrollo, centrándonos sólo en los dos servicios de uso mayoritario, tendremos un mercado potencial de más de 500 millones de usuarios. Por lo que hemos decidido dar soporte tanto a Dropbox como a Google Drive, dejando como opción futura la inclusión de cualquier otro.

[1]Who's winning the consumer cloud storage wars: Revista Fortune
[2]Fuente: winbeta.org
[3]Fuente: theNextWeb.com
[4]Fuente: theNextWeb.com

Como hemos comentado, estas aplicaciones tienen sentido en un entorno multidispositivo, donde el usuario está acostumbrado a utilizar una u otra pantalla indistintamente. Por lo tanto, hemos centrado nuestro desarrollo en crear una web que se adapte a las necesidades del usuario en cada uno de estos dispositivos y que haga uso de las ventajas de cada uno de ellos.

1.2. Objetivos

Objetivo principal

En consecuencia, este proyecto se plantea como objetivo principal la creación de una aplicación web donde poder gestionar de manera única todos los servicios de almacenamiento en línea que pueda tener un usuario, sean estos de pago o no. Creemos que la integración ha de ser transparente de cara al usuario, puesto que esto permitirá que el mismo pueda reemplazar las demás aplicaciones.

Objetivos secundarios

Como objetivos secundarios, hemos querido mantener el enfoque en:

1. **Uso de tecnologías web avanzadas:** En la actualidad, el desarrollo web está avanzando a mucha velocidad. Queríamos aprovechar esta oportunidad para adentrarnos en tecnologías web de última generación, como HTML5, CSS3, Django, AngularJS; Las cuales explicaremos más adelante.

2. **Metodología de trabajo:** Otro de los objetivos ha sido mantener una metodología de trabajo muy orientada a al entorno profesional, haciendo uso de Git y Gitflow.

3. **Diseño de interfaz y experiencia de usuario:** Un detalle que no hemos querido dejar de lado, por lo que hemos buscado que la curva de aprendizaje al usar nuestra aplicación sea bastante suave. Para ello, se ha dispuesto una interfaz bastante sencilla e intuitiva, teniendo en cuenta todos los tipos de dispositivos

1.3. Estructura de la memoria

Con el fin de facilitar la lectura de esta memoria, creemos necesario explicar brevemente su estructura, indicando los objetivos de cada uno de los capítulos que la componen:

1. **Introducción.** Tiene como objetivo enmarcar el contexto de este proyecto, así como los objetivos planteados desde el inicio.

2. **Estado de la ciencia.** En esta sección presentamos las principales tecnologías empleadas durante el proyecto y los conceptos para entender la estructura del proyecto.

3. **Diseño e implementación.** Trata de explicar y profundizar en los detalles del desarrollo de este proyecto, así como de su infraestructura y evolución.

4. **Resultados.** A modo de resumen, repasamos el estado final de nuestro proyecto.

5. **Conclusiones.** Capítulo final destinado a la evaluación del proyecto en general, así como los conceptos aprendidos durante el mismo.

Finalmente, podemos encontrar la bibliografía que ha sido consultada para la elaboración de esta memoria.

Capítulo 2

Estado de la Ciencia

En este capítulo hacemos un recorrido inicial de las tecnologías y herramientas que hemos tenido en cuenta para desarrollar este proyecto. Expondremos una visión general de las mismas para detallar en próximos capítulos su uso.

2.1. Introducción a La Web

La Web e Internet en general, han crecido en los últimos 20 años de manera exponencial. En 1991 fue creado el lenguaje HTML *(HyperText Markup Language)* por Tim Berners-Lee, quien en un primer boceto del mismo, incluyó 22 etiquetas con las que describir documentos de texto y jerarquizar su contenido[1]. Para la transferencia de estos documentos, se ideó HTTP *(HyperText Transfer Protocol)*, muy enfocado a transacciones, y sin estado.

Es en 1995, cuando se consigue publicar el primer estándar oficial, a pesar de su nombre, *HTML2.0*. Tras esta estandarización, sería el W3C (*WorldWideWeb Consortium*) el organismo encargado de publicar la recomendación referente a HTML, lanzando la versión 3.2 en el siguiente año. Ésta incluyó grandes mejoras como los applets de Java y el texto fluido alrededor de imágenes.

[1]Primer boceto HTML: www.w3c.org

Una de las mayores evoluciones fue el salto a la versión 4.0, publicada en 1998, donde se incluyen, por primera vez, las hojas de estilo CSS y JavaScript; Lo que, a largo plazo, ha sido el mayor avance de HTML hasta la fecha junto a HTML5, puesto que ambos añadidos han propiciado el cambio de paradigma: HTML ya no representa un documento, si no que permite crear aplicaciones bastante parecidas a las de escritorio.

La última y más reciente versión es HTML5, que ha permitido numerosos avances y ha propiciado la explosión del propio lenguaje [7]. En ésta, se incluyen etiquetas con sentido semántico – Muy útiles para optimización en buscadores –, además de nuevas etiquetas nativas para el uso de audio y vídeo. Pero este lanzamiento, realmente, ha supuesto un avance importante por las nuevas opciones que se ponen a disposición del desarrollador. Ahora, existe la posibilidad de adaptar sus aplicaciones a las diferentes pantallas gracias a CSS3, que permite definir estilos dependiendo de las características físicas de las mismas; Además, haciendo uso de herramientas como *LocalStorage* se pueden almacenar datos en el propio navegador y acceder a recursos como la cámara, el micrófono o la localización.

Como se puede intuir, la evolución de HTML no está para nada enfocada a describir un documento, como era su idea inicial, si no a crear webapps que permiten el uso de los múltiples recursos del dispositivo donde se ejecutan, y acercar la experiencia de escritorio a la web.

Las ventajas de este enfoque, propuesto en HTML5, son claras:

- **No hay instalación:** No es necesario ocupar memoria en el dispositivo ni realizar un proceso de instalación.
- **Actualizaciones automáticas e inmediatas:** Como consecuencia de lo anterior, tras la publicación de una nueva versión, todos los usuarios la reciben y sin acción por parte de los mismos.
- **Accesible globalmente:** Una aplicación web puede ser accedida desde cualquier parte del mundo con acceso a internet y, virtualmente, desde cualquier dispositivo (si el desarrollador así lo quiere).
- **Compatibilidad en distintas platafomas:** A diferencia de la mayoría de los programas de escritorio, las *webapps* pueden ser ejecutadas indistintamente en cualquier ordenador sin importar el sistema operativo o arquitectura interna.

Con todas estas mejoras, hemos vivido en los últimos 5 años una explosión en cuanto a crecimiento de frameworks de desarrolo en JavaScript, en parte gracias a librerías como *jQuery* y, muy recientemente, *AngularJS*. Estas permiten que nos podamos abstraer, en parte, del desarrollo básico y repetitivo, para centrarnos en el avance de nuestra aplicación en concreto.

2.2. Aplicaciones cliente-servidor

La arquitectura cliente-servidor [1], es una de las más extendidas en redes de ordenadores en la actualidad. Ésta se basa en dos *roles* claramente diferenciados: El servidor, o proveedor de recursos; y el cliente o demandante de recursos.

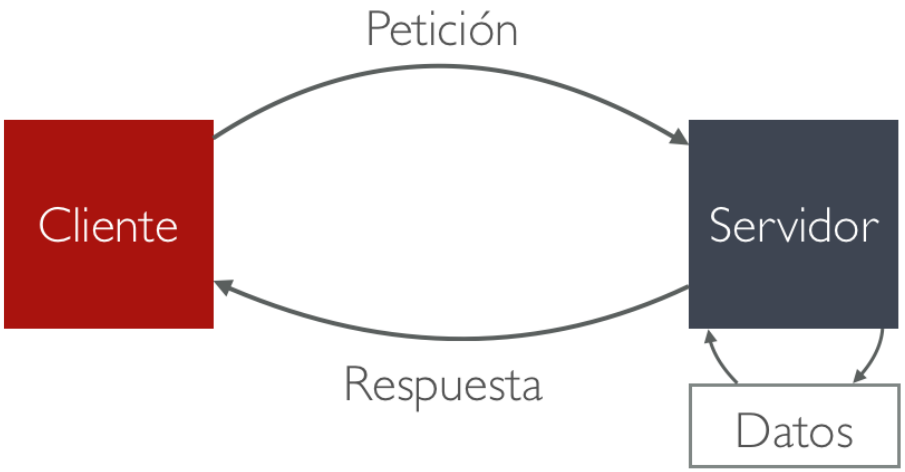


Figura 2.1: Estructura básica cliente-servidor

Es claramente un sistema centralizado, pues toda la información reside en los servidores, mientras que todos los clientes se encargan de acceder a la información allí alojada. Esto simplifica bastante el diseño y la estructura de la aplicación, puesto que la lógica más compleja normalmente reside en el servidor. Por tanto, el servidor será capaz de manejar de manera distinta las peticiones dependiendo de quién las realice. Además, aumenta la seguridad puesto que el lado cliente es mucho más sensible a ataques.

Nuestra aplicación se divide en una parte servidora y otra cliente. Por lo tanto, realizaremos una introducción a las tecnologías empleadas en cada caso. En siguientes capítulos, veremos como hemos conseguido conectar ambas partes, y la estructura final del proyecto.

2.2.1. Servidor

Si distinguimos por el modo de procesado de peticiones de los clientes, podemos diferenciar dos tipos de servidores:

- Servidores iterativos: Se atienden las peticiones de forma secuencial, por tanto se crea una cola de peticiones donde aumenta el tiempo de respuesta.
- Servidores concurrentes: Las peticiones son atendidas por un proceso, que a su vez lanza nuevos hilos al recibir una petición. Será este segundo hilo el que realice las tareas necesarias para generar la respuesta.

La ventaja de un servidor concurrente frente a uno iterativo en cuanto a tiempos de respuesta es muy amplia, ya que el servidor no mantiene en espera una petición mientras se procesa la otra.

En cambio, el coste computacional es mayor ya que cada petición genera un nuevo hilo y habrá que manejar las condiciones de carrera derivadas de la concurrencia. En general, este tipo de servidores son los más usados, ya que con una potencia computacional media, se pueden atender múltiples peticiones simultaneamente.

Cabe destacar, sin duda, la irrupción en los últimos años de servidores iterativos asíncronos, como es el caso de *NodeJS*. Este tipo de servidores realizan tareas de forma asíncrona y no-bloqueante, por lo que el tiempo de respuesta es, en muchos casos, menor que en servidores multihilo. Por estos motivos, en la actualidad, múltiples proyectos enfocados al tiempo real, utilizan este tipo de servidores asíncronos.

2.2.2. Cliente

El cliente será el encargado de realizar las peticiones al servidor dependiendo, normalmente, de las acciones del usuario. También es el encargado de presentar los datos de respuesta del servidor a cada una de estas peticiones.

El navegador web, en nuestro caso, cumplirá con este papel, y nos permitirá mostrar al usuario la información de forma clara. A través del mismo, irá accediendo a los distintos contenidos que el servidor remita.

Pero no sólo los navegadores web pueden desempeñar el papel de cliente. En la actualidad, la mayoría de teléfonos inteligentes y tabletas, hacen uso de esta arquitectura gracias a la implementación de las APIs REST.

2.3. Tecnologías de servidor

Históricamente, los servidores web atendían peticiones HTTP de manera estática: Al recibir una petición, buscaban el archivo html en su sistema de archivos y lo retornaban. Por lo tanto, las páginas tenían únicamente contenido fijado en sus archivos html. Si se quería añadir un apartado nuevo, había que modificar cada uno de esos archivos.

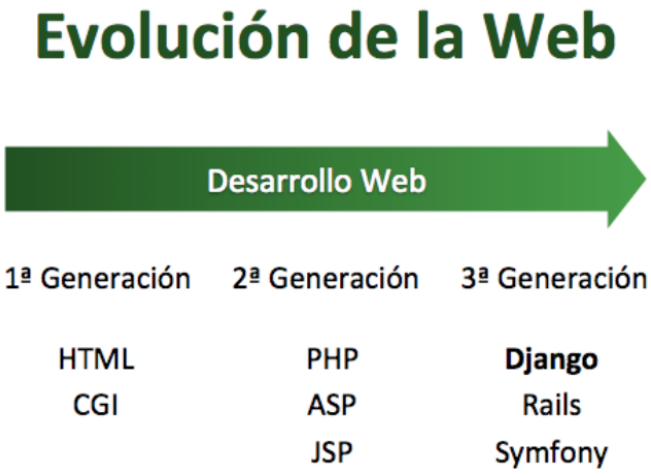


Figura 2.2: Evolución del desarrollo web en el servidor

Más adelante, surgieron otro tipo de servidores más avanzados, capaces de devolver el resultado de la ejecución de un programa o realizar consultas a una base de datos y renderizar html con los resultados de la misma. Al almacenar nuestros datos en una base de datos, podemos organizar el contenido, filtrarlo según necesitemos, e incluir nuevos datos sin construir un archivo html completo.

En nuestro caso, necesitaremos un servidor avanzado, ya que queremos consultar bases de datos y comunicarnos con otros sistemas a través de su API. Para ello, hemos optado por un servidor *Python* con el framework *Django*.

2.3.1. Python

El lenguaje *Python* fue desarrollado a finales de los años 80 por el holandés Guido Van Rossum como sucesor del lenguaje ABC. Se caracteriza por tener una sintaxis limpia y ordenada, haciendo mucho hincapié en la legibilidad del código, lo que permite un rápido aprendizaje por parte del desarrollador. A pesar de ello, tiene funcionalidades muy avanzadas, propias de lenguajes como C o C++.

Además, es multiparadigma, pues podemos hacer uso de programación imperativa, funcional u orientada a objetos. Es un lenguaje multiplataforma con una fácil portabilidad, aunque, al ser un lenguaje interpretado, está limitado a la existencia del mismo.

Por último, hay una gran comunidad de desarrolladores de *Python* ya que es uno de los lenguajes más utilizados en todo el mundo, y ha tenido un gran crecimiento en los últimos años gracias a su portabilidad, facilidad de uso y velocidad.

2.3.2. Django Framework

Django es un *framework* web de código abierto escrito en Python [8]. Fue desarrollado para gestionar páginas de noticias en la World Company Lawrence (EEUU). En 2005 se liberó su código bajo licencia BSD.

Al estar enfocado en sus inicios a la publicación de artículos, proporciona herramientas que facilitan mucho el desarrollo de páginas orientadas a contenidos. Está totalmente orientado al modelo MVC que, de forma básica, consiste en separar los datos de nuestra aplicación (modelo), la lógica (controlador) y la presentación (vista).Este *framework*, por tanto, permite la creación de sitios web complejos de manera rápida, como reza su lema '*The Web framework for perfectionists with deadlines*' o '*El framework Web para perfeccionistas con fechas de entrega*' en español.

El mismo está pensado para la creación de aplicaciones web partiendo de módulos independientes, para poder reutilizar al máximo nuestro código en los diferentes proyectos que afrontemos.

Por otro lado, una gran ventaja de Django es que nos permite una abstracción casi total de la base de datos, ya que nos provee una API para el acceso a nuestros modelos. Aunque, como muchos desarrolladores avanzados comentan[2], en ocasiones esto puede ser un problema si nos olvidamos por completo de lo que ocurre a bajo nivel.

2.3.3. PostgreSQL

PostgreSQL es un sistema de gestión de bases de datos objeto-relacional, distribuido bajo licencia BSD y con su código fuente disponible libremente. De los sitemas abiertos, es uno de los más potentes, incluso acercándose mucho al rendimiento de otras bases de datos comerciales.

PostgreSQL utiliza un modelo cliente-servidor, en este caso nuestro servidor *Django* actuaría de cliente de la base de datos. Además, hace uso de multiprocesos, en vez de multihilos, para garantizar la estabilidad del sistema: Un fallo en uno de los procesos no afectará el resto y el sistema continuará funcionando.

2.3.4. Nginx

Nginx es un servidor HTTP de alto rendimiento [12] usado en sitios como *GitHub*, *Hulu* o *Netflix*. Fue creado en 2008 para manejar las peticiones de varios sitios web de *Rambler*, un buscador ruso con unas 500 millones de peticiones al día. Este servidor es un proxy inverso, que cuenta con balanceo de carga y opciones de cacheado de información. Trabaja bastante bien con Django, ya que es capaz de comunicarse con los distintos hilos de ejecución de la aplicación web, como veremos, también gracias a Gunicorn.

2.3.5. Gunicorn

Gunicorn es un servidor HTTP WSGI (Web Server Gateway Interface) para python [4]. Éste, permite la comunicación entre Nginx y Django, ya que es capaz de crear varias instancias o *workers* que corren la aplicación en paralelo y los mantiene a la espera. Cuando Nginx recibe una petición entrante, se comunicará con Gunicorn y pondrá en marcha uno de estos workers de Django para que la procesen.

[2]ORM is an anti-pattern: sheldo.com

2.3.6. Supervisor

Supervisor es un paquete que nos permite monitorizar y controlar una serie de procesos en sistemas UNIX. Parte de sus objetivos son idénticos a *Daemontools* o *Launchd*, la diferencia es que Supervisor vigila la ejecución de los procesos que configuremos, para que estén siempre activos. De esta manera podemos conseguir que el servidor web siempre esté ejecutándose y que, en caso de tener un cierre o error inesperado, los procesos de Django, Nginx y Gunicorn vuelvan a ejecutarse .

2.4. Tecnologías en el lado cliente

En el lado cliente, tenemos tecnologías relacionadas con HTML, JavaScript y CSS. En entornos profesionales, no se suelen utilizar lenguajes directamente, si no que se hace uso de *frameworks* que permiten un desarrollo más fluido y rápido. En nuestro caso, hemos utilizado varios de ellos, los cuales indicamos a continuación

2.4.1. HTML5

Como hemos indicado anteriormente, HTML5 es la versión más reciente de HTML [11]. El motivo principal de su aparicion es la gran evolución de la web en los últimos años y el cambio en la forma de interactuar por parte del el usuario, ya no se consultan webs sólo desde ordenadores, sino que hemos dado el salto a un entorno multidispositivo, donde gran parte del tráfico lo generan smartphones y tablets.

2.4.2. CSS3

CSS o *Cascading Style Sheet* (Hoja de estilos en cascada) es el lenguaje de diseño de la web. Su estandarización y especificación corre por parte del W3C, que lo incluyó a partir de la versión 4 de HTML.

Gracias a este lenguaje, podremos indicar dónde se colocan los elementos, su color, apariencia, etc. En esta última vesión se han incluido grandes mejoras, podemos hacer uso de transformaciones 2D y 3D así como animaciones aceleradas por GPU, lo que hace que sean mucho más suaves y vistosas que programándolas como hasta ahora, en JavaScript.

Al tratarse de *estilos en cascada*, los estilos que definamos en un elemento que contenga a otros, éstos podrán propagarse hacia abajo. Por ejemplo: Si cambiamos el tamaño de fuente al elemento <body>(Padre de todo el documento), todos los párrafos y links de la página tendrán ese tamaño de fuente a menos que indiquemos lo contrario.

Para realizar esta diferenciación de estilos, se suelen utilizar los selectores de id y clase. Si en el documento HTML definimos un elemento <p>con una clase, podremos indicarle un estilo diferente que a los demás. De esta aproximación, ha surgido una corriente llamada ***OOCSS*** (*Object Oriented CSS*) y varios Frameworks como *Foundation* y *Bootstrap*

2.4.3. OOCSS

Object Oriented CSS se basa en la inclusión de clases en HTML para organizar el diseño de nuestro documento. Esta solución se puede utilizar tanto de manera individual, donde nosotros mismos definimos las clases y las reutilizamos, o a través de un framework CSS.

Por ejemplo, si queremos que cinco textos distintos se alineen a la derecha, manteniendo todos los demás estilos aplicados, podemos añadirle a cada uno la clase en HTML 'text-right'. En cambio, sin tener en cuenta OOCSS, tendríamos que poner un identificador o clase mucho menos genérica y aplicar los estilos a cada uno de ellos por separado.

Por otra parte, el uso de OOCSS ha propiciado la aparición de frameworks de diseño CSS, como son *Bootstrap* o *Foundation*, que aceleran mucho la creación de un sitio web adaptado a todos los tipos de pantalla. Por contra, el uso de los mismos está tan masificado que podemos llegar a ver varios sitios web con exactamente los mismos botones y diseño casi idéntico. La causa de este problema, en cambio, no es directamente el uso de estas herramientas, sino la mala praxis de los desarrolladores, que no se preocupan por dotar a su aplicación de una personalidad propia. Por este motivo, entre otros que veremos más adelante, hemos decidido crear un diseño basado en Foundation para nuestra aplicación, ya que este framework nos provee de ciertas características que nos facilitan el desarrollo, pero de una manera muy genérica, lo cual nos hará más sencillo personalizar nuestra interfaz y tener una apariencia propio.

2.4.4. Stylus

Stylus es un precompilador CSS, que permite escribir código de manera más rápida y sencilla. Nosotros escribiremos sentencias y propiedades en su lenguaje, y éste será ejecutado para generar archivos CSS finales.

Una de sus ventajas es la sencillez de la sintaxis, ya que elimina varios aspectos tediosos, y tiene una filosofía similar a python. Por ejemplo, elimina los corchetes y anida los elementos dependiendo de su indentación. Otra de las ventajas de su uso es que podemos emplear variables y mixins (funciones), que permiten crear código reutilizable que devuelve una propiedad dependiendo de un parámetro de entrada.

Pero el uso de Stylus por sí solo no es un gran avance, simplemente es un cambio de sintaxis a una más simple y legible. El motivo principal de su uso, viene dado por otros añadidos que

podremos incluir. Por ejemplo, podemos hacer uso de la librería *nib* que permite escribir propiedades de CSS cuya sintaxis es distinta en cada navegador, como animaciones y degradados, una única vez y ésta lo traducirá para hacerlo compatible con todos ellos incluyendo los prefijos necesarios.

Figura 2.3: *Derecha*: CSS nativo. *Izq.*: Stylus con un mixin, **sin nib**

2.4.5. JavaScript

JavaScript es el lenguaje de programación para la web. Es un lenguaje interpretado, orientado a objetos y basado em prototipos. Una de sus características más importantes es que se define como un lenguaje asíncrono, por lo que es muy útil para reaccionar a eventos por parte del usuario.

Todos los navegadores modernos añaden un intérprete de JavaScript, que permite el acceso a una representación del documento, el *DOM* o *Document Object Model*. Este modelo, nos permite interactuar con los elementos HTML de la página web, interactuar con los eventos que se produzcan en ella y modificar el contenido de la misma.

También junto a la última versión de HTML se incluyen numerosas mejoras en JavaScript. Se ha abierto la posibilidad de acceder a recursos del dispositivo como la cámara y geolocalización entre otros. Además, podemos hacer uso de LocalStorage y SesionStorage, que permiten guardar información en el lado cliente, lo que usualmente se hacía, sin más remedio, mediante cookies.

Durante los últimos años, han surgido también varios frameworks para trabajar de una manera más ágil con JavaScript. Uno de los que más impacto ha tenido es jQuery, pero cabe destacar la gran explosión de AngularJS en los últimos meses.

2.4.6. jQuery

Esta librería es un conglomerado de funciones JavaScript que permiten hacer desde animaciones al documento HTML hasta llamadas a un servicio web [2]. Como indica su lema *write less do more*, su objetivo principal es proporcionar funciones repetitivas mediante *wrappers* o *envoltorios* para que sea más sencillo su uso y configuración.

Uno de los mayores problemas de jQuery reside en que no proporciona una buena abstracción a la hora de dividir nuestro códigopara una webapp. Es decir, otros frameworks MV* (Model-View-Whatever)[3] como *BackboneJS*, *EmberJS* o *AngularJS* permiten estructurar nuestra aplicación aplicando variantes del paradigma MVC, lo que nos permite tener un proyecto con partes mucho más independientes y reutilizables, como explicaremos más adelante.

El uso de esta librería ha sido muy intenso desde el año 2007, incluso usándose en proyectos donde no era la mejor solución, pero durante el último año la tendencia ha comenzado a cambiar y jQuery empieza a perder fuerza gracias a la aparición de los frameworks MV* nombrados anteriormente, que han tomado mucha más fuerza[4] para evitar el código spaguetti[5] y que permiten crear aplicaciones web de manera más sencilla.

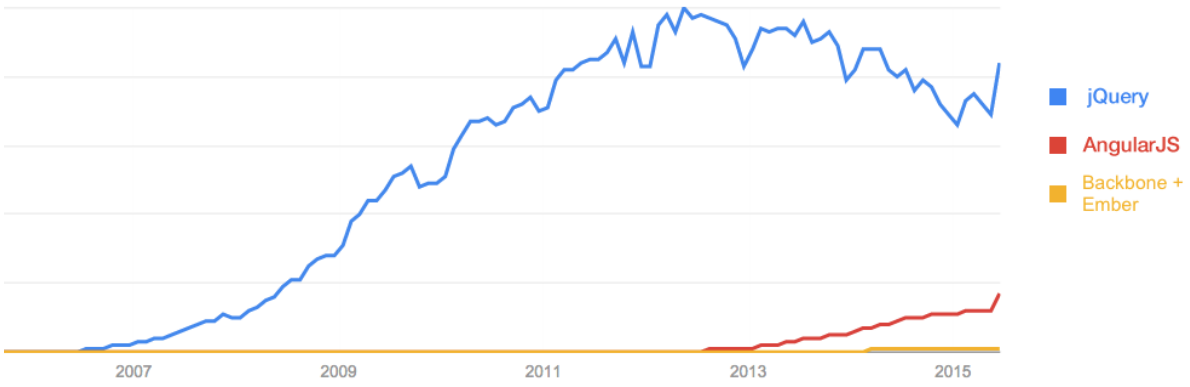


Figura 2.4: Interés en los principales frameworks JavaScript

[2]Model-View-Whatever: sitepoint.com
[3]Backbone, Ember y Angular: airpair.com
[5]jQuery, the good the bad and the ugly webdesignerdepot.com

2.4.7. AngularJS

Este framework comenzó a ser desarrollado por un empleado de Google, Miško Hevery, en el año 2009. Fue en 2012 cuando se lanzó su versión 1.0 aunque no tuvo gran repercusión hasta finales de 2013. Durante el año 2014, su uso se ha ido expandiendo hasta el punto de convertirse en uno de los frameworks más importantes en la actualidad [6].

AngularJS permite realizar múltiples tareas de manera sencilla. Nos permite crear nuestra aplicación de acuerdo al paradigma MVC, que explicaremos más adelante, pero también podemos *extender* las capacidades de HTML, creando etiquetas propias a partir de elementos HTML nativos.

Una de las características más impactantes de este framework es el *two-way data binding*, o enlazado de datos en doble dirección. Esta funcionalidad, enlaza un modelo de datos con su representación en la vista. Por lo tanto, cualquier cambio realizado en la vista se verá reflejado en el modelo y viceversa.

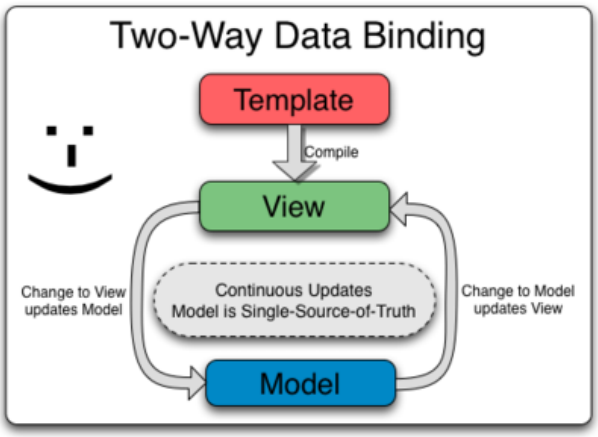


Figura 2.5: Diagrama explicativo del enlazado de datos en doble dirección

2.5. Otras tecnologías y metodología

Uno de los objetivos de este trabajo, era enfocar el desarrollo al entorno profesional, en esta sección veremos la forma de trabajar empleada y las herramientas utilizadas.

2.5.1. API REST

Un API REST (**RE**presentational **S**tate **T**ransfer), se basa en utilizar los métodos HTTP para comunicar dispositivos o aplicaciones [10]. Es decir, una aplicación expone unos recursos a través de ciertas urls, sobre los que los clientes pueden realizar las distintas operaciones *CRUD* (Create, Recover, Update, Delete):

- Create — POST
- Recover — GET
- Update — PUT
- Delete — DELETE

Gracias a esta implementación, podemos trabajar con WebServices más ligeros y accesibles desde distintos dispositivos. Como respuesta, el servidor puede utiliza códigos HTTP y, en ocasiones, añadir contenido. Todos los recursos se exponen a través de una URL, y cada uno de los métodos soportados por la misma, será denominado *endpoint*. Por ejemplo, si queremos publicar un listado de archivos, podríamos crear lo siguiente:r

- URL del recurso: **/api/files/**
- Endpoints:
    - GET: Listado de archivos
    - POST: subida de un nuevo archivo

En este caso, sólo tendremos dos endpoints, puesto que si queremos realizar operaciones con un fichero concreto, debemos utilizar la url del propio recurso:

- URL del recurso: **/api/files/<id>**
- Endpoints:
    - GET: Detalle del archivo
    - PUT: Modificación del archivo
    - DELETE: Borrar el archivo

Las APIs REST son universalmente utilizadas en la actualidad y tanto Dropbox como Google Drive exponen sus recursos mediante las mismas, aunque en nuestro caso hemos utilizado el SDK que nos abstrae algo de la capa HTTP. Por otro lado, en nuestro servidor hemos hecho uso de REST para exponer de manera estandarizada los distintos servicios del usuario, aunque nuestra implementación no ha sido completa, ya que no realiza un diseño por endpoints sino que tendremos que indicar la acción en la propia URL (Ver sección lineas futuras).

2.5.2. JSON

JSON, acrónimo de JavaScript Object Notation [5], es un formato ligero para el intercambio de datos. JSON es un subconjunto de la notación literal de objetos de JavaScript que no requiere el uso de XML. JSON (JavaScript Object Notation) es un formato muy extendido para el intercambio de datos entre aplicaciones. Su estructura está basada en cómo se crea un objeto Javascript, formado por tuplas clave-valor para representar cada propiedad de una entidad o corchetes para representar arrays. En la actualidad, ha reemplazado el uso de XML para resolver esta comunicación ya que es mucho más ligero y legible por humanos. En este proyecto, hemos utilizado el parseador por defecto de Python para responder a las llamadas asíncronas desde Javascript.

2.5.3. Git

Git es un sistema de control de versiones de código libre y abierto, creado por Linus Torvalds para mantener el desarrollo del núcleo de Linux [3]. Por tanto, está diseñado para controlar cualquier tipo de proyectos independientemente de su magnitud.

Git mantiene todos los *commits* que hagamos de nuestro proyecto. Un commit es una *foto* de nuestros archivos en un determinado momento. Estos incluyen un identificador, todos los cambios respecto al commit anterior y una referencia al mismo. De esta manera, siempre que queramos, podremos retroceder hasta una versión anterior de nuestro código.

Por otro lado, permite tener varias versiones en pararelo o *ramas* de nuestros proyectos. Éstas son muy útiles para trabajos en equipo, ya que cada desarrollador puede implementar sus funcionalidades en una rama (branch) y luego unirse (merge) a las del resto. Al realizar un *merge*, se pueden ocasionar **conflictos** cuando en ambas ramas se ha modificado el mismo archivo en la misma línea. En estos casos, se puede optar por resolverlo con nuestro editor de texto, o con herramientas específicas. Hemos optado por la primera opción ya que las herramientas que hemos utilizado, como Kdiff3, no ofrecen una gran ventaja.

En nuestro caso, hemos hecho uso de ramas para avanzar dos versiones del proyecto de manera independiente. Por un lado haciendo uso de jQuery y, por otro, durante los últimos meses, añadiendo AngularJS. Para llevar esto a cabo, hemos optado por utilizar GitFlow como metodología de trabajo.

2.5.4. GitFlow

GitFlow es una metodología de trabajo muy útil para desarrollos software que hagan uso de Git. Pensada inicialmente por Vincent Driessen, se ha implementado como la metodología por defecto en muchos entornos de programación. Ésta consiste en crear, como mínimo, cinco ramas principales:

- **Feature**: Se creará una rama **feature/nombre-de-funcionalidad** cada vez que se quiera añadir una funcionalidad al proyecto. Una vez terminada ésta, debemos unirla, haciendo merge, a la rama **develop**

- **Develop**: Esta rama se destinará a llevar un seguimiento del desarrollo del código. Será donde cada desarrollador una sus funcionalidades terminadas. Una ver revisado el código y arreglados bugs y conflictos, esta rama se une a la de **release**

- **Release**: Tras completar el desarrollo de una versión, esta rama incluirá los cambios de la rama **develop**. Este código se comprueba, pudiéndo incorporar arreglos de bugs, que se aplicarán también en la rama develop.

- **Master**: Por último, una vez testeado el código, el proyecto pasa a la rama master, que suele estar alojada en el servidor público o de producción.
- **HotFix**: Si hubiera algún error en el proyecto mientras está en producción, se suele crear la rama **HotFix**, donde rápidamente se arregla y se aplican los cambios a las ramas **master** y **develop** para que los desarrolladores puedan tener la versión más estable del código.

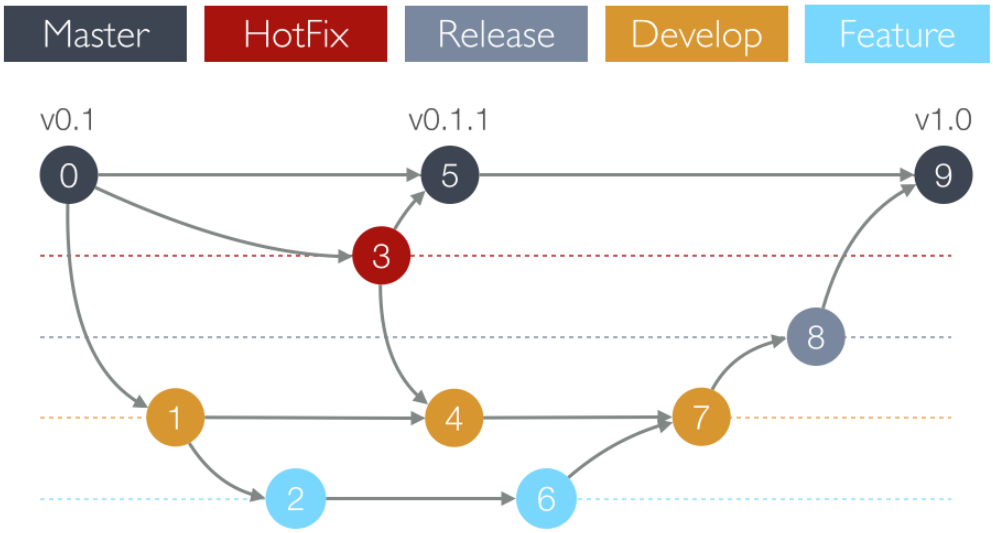


Figura 2.6: Diagrama del flujo de trabajo de GitFlow

Como vemos, puede ser algo tedioso de entender y de aplicar por parte de un gran equipo de desarrolladores. Tendríamos que ser muy disciplinados y que todos estuvieran correctamente coordinados y atentos a los cambios en las ramas del repositorio remoto (master y hotfix).

Para ayudarnos con esta metodología de trabajo, usamos *SourceTree*, que implementa una interfaz visual del repositorio y sus ramas. Además, nos permite llevar el control sobre el flujo de GitFlow.

2.5.5. SourceTree

SourceTree es un software desarrollado por Atlassian, una compañía con varios proyectos relacionados con Git y el desarrollo ágil como *BitBucket* o *Jira*. SourceTree ofrece una interfaz gráfica para el uso de Git, pero sobre todo es poderoso cuando hacemos uso de GitFlow.

Gracias a su interfaz, podremos ir creando las ramas que necesitemos de una forma mucho más intuitiva. Sólo tendremos que hacer click en la opción GitFlow y podremos crear un nuevo *feature*, *hotfix* o *release*, y nos hará las configuraciones necesarias.

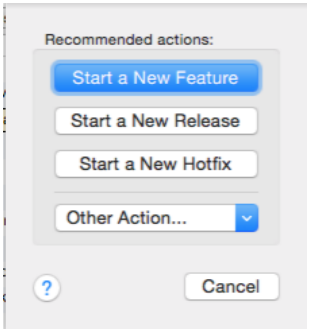


Figura 2.7: Opciones para comenzar un nuevo feature, release o hotfix

Una vez hayamos concluido con una de estas opciones, podremos cerrarla y hará la unión o *merge* con la rama que le corresponda, de forma automática y transparente al usuario. Si esta unión causara algún conflicto, deberemos resolverlo manualmente.

Capítulo 3

Diseño e Implementación

3.1. Introducción

En esta sección describiremos más a fondo el diseño y desarrollo realizados en el proyecto. Antes, comenzaremos con una breve introducción al mismo, para tener una visión más global y poder enterder más adelante las partes por separado.

3.1.1. Estructura básica

Para conseguir el objetivo principal del proyecto, hemos tenido que tomar ciertas decisiones clave en cuanto a desarrollo. En un principio pensamos en acceder a los servicios de almacenamiento del cliente desde el propio JavaScript, pero no nos parecía del todo seguro puesto que habría que exponer algunas claves secretas como las API Keys y tokens únicos de usuarios, quedando *al descubierto* datos críticos de la aplicación.

Por tanto, optamos por crear un servidor que sirviera de *proxy*, capaz de recibir las distintas peticiones y comunicarse con los servicios de terceros. De esta forma, aumentamos la seguridad y abrimos la puerta a futuros desarrollos, puesto que hemos creado una API que se puede consumir desde otros clientes como aplicaciones móviles o de escritorio (Pudiéndose sincronizar carpetas en tiempo real)

Como consecuencia, la estructura final de la aplicación tiene 3 divisiones principales:

- **Cliente:** Es la parte más externa y donde el usuario interactúa con la aplicación.
- **Servidor:** El bloque fundamental del proyecto reside en el servidor ya que realiza, de forma transparente, conexiones con cada una de las cuentas del usuario.
- **Servicios externos:** Son los servidores de terceros a los que tenemos que acceder para conseguir la información de las cuentas del usuario.

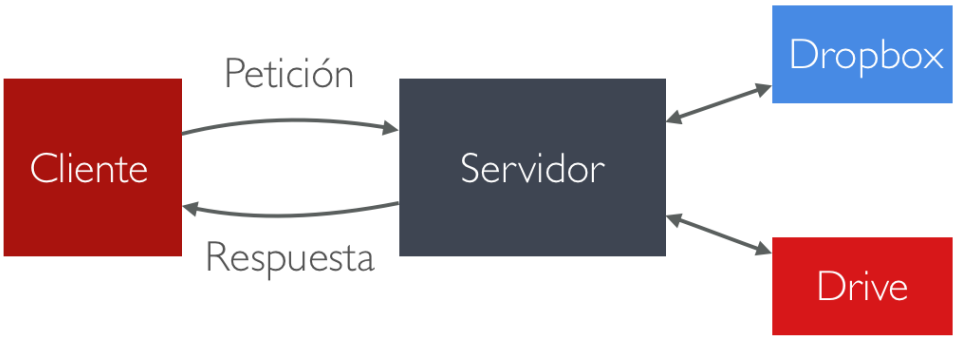


Figura 3.1: Diagrama básico de la aplicación. Elaboración propia

3.2. Desarrollo en el lado servidor

En el lado servidor, se ha utilizado Django para manejar la lógica de la aplicación y la comunicación con la base de datos. En nuestro caso, además de realizar esta comunicación, es necesaria una conexión con los servicios de terceros como son Dropbox y Google Drive.

Gracias al Patrón Modelo Vista Controlador, en el que se basa Django, hemos conseguido realizar una serie de bloques que se comunican entre sí para conseguir una aplicación mantenible y extensible.

3.2.1. Patrón MVC

MVC es un patrón que separa completamente los datos, la lógica de negocio y la interfaz de usuario de una aplicación. Esto ayuda a la reutilización de código y su mantenimiento. Veámos en qué se centra cada parte:

- El **modelo** es la información que maneja nuestra aplicación y su estructura.
- En el **controlador** es donde reside la lógica de negocio y, por tanto, será el encargado de procesar los datos y hacer operaciones con ellos. Por otro lado, también es el puente de comunicación entre los datos y las vistas.
- Las **vistas** suelen ser puras representaciones de los datos ofrecidos por el controlador.

3.2.2. Introducción a Django y MTV

Django está basado en el patrón MVC pero hace algunas modificaciones al mismo. En esta implementación, conocida como MTV (**M**odel, **T**emplate, **V**iew), hace uso de nuevos elementos enfocados a dar soporte a una aplicación web.

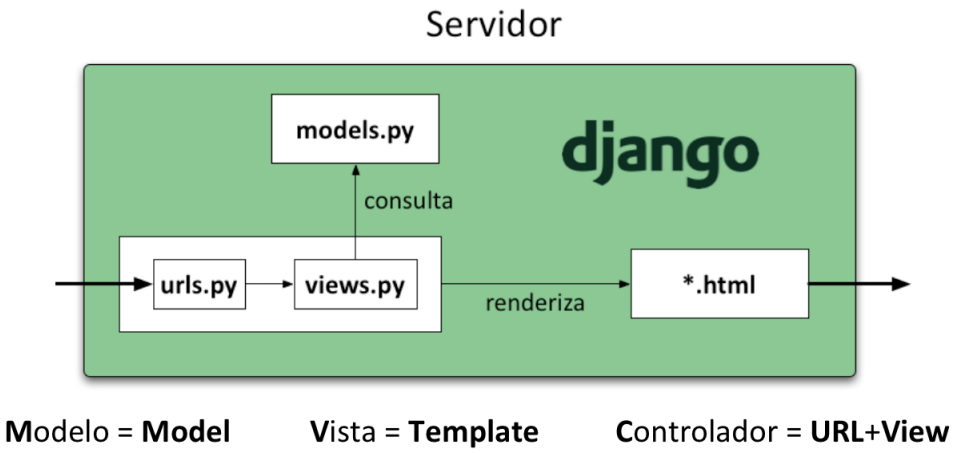


Figura 3.2: Estructura básica de Django MTV. Fuente: Jorge Bastida

Por tanto, un proyecto de Django se separa en varios archivos que cumplen con una funcionalidad distinta, obteniendo cuatro archivos principales:

- **urls.py** Se encarga de capturar las peticiones, y dirigirlas a la vista que indiquemos. Este enlazado se realiza mediante expresiones regulares que definimos en este mismo archivo
- **views.py** En este archivo definimos funciones en Python que recibirán como argumento los parámetros de las urls. Dependiendo de los mismos, normalmente, realizamos una serie de consultas a la base de datos, representada en el archivo models.py. Además, las vistas son las encargadas de renderizar las plantillas inyectando los datos recibidos del modelo y retornarlas.
- **models.py** El modelo representa un esquema de nuestros objetos en la base de datos. Gracias a Django, podremos abstraernos de las consultas e inserciones, así como de la creación de las tablas, que se realiza de manera automática. Los modelos, por tanto, son clases escritas en python que se almacenan en la base de datos gracias al ORM.
- **templates/*.html** Las plantillas son archivos HTML con variables y funciones de Django, que permiten manejar los resultados inyectados por las vistas.

Esta separación es muy útil, puesto que cada parte puede concentrarse en resolver su tarea de manera más sencilla, sin tener que recurrir a grandes cantidades de código. Por otro lado, si creamos una comunicación organizada entre cada bloque, un cambio interno en uno de ellos, normalmente, no afectaría al resto.

3.2.3. Django: Aplicaciones

Para conseguir una estructura de ficheros ordenada, Django nos permite dividir nuestro proyecto en bloques más pequeños, llamados aplicaciones. De esta forma, podemos añadir funcionalidades desarrolladas por otros programadores o mantener una estructura bien distribuida de nuestro propio proyecto. Cada aplicación puede contener sus propios archivos de plantillas, urls, vistas y modelos, por lo que pueden ser totalmente autocontenidas.

En nuestro caso, hemos definido cuatro aplicaciones que interactúan entre sí, pero que ofrecen una abstracción bastante alta de los procesamientos que cada una realiza internamente. Así pues, tenemos las siguientes aplicaciones:

- **Mwc_dropbox y Mwc_drive** Estos bloques se centran en resolver todo lo referente a los propios servicios de almacenamiento. También son los encargados de la autenticación del usuario en los servicios de terceros, donde el mismo nos dará acceso a sus datos. Una vez obtenido el acceso, podremos realizar las llamadas a los servidores remotos, cuyos datos son presentados a los dos bloques siguientes de una manera estandarizada.

- **Main:** Es el bloque de unión de todas las aplicaciones anteriores. Se encarga de renderizar las distintas plantillas dependiendo a qué sección se acceda y del manejo de usuarios (Login, registro, etc.). Es la única aplicación que devuelve archivos HTML procesados con los datos de la aplicación, puesto que las demás se encargan de APIs tanto de terceros como la propia.

- **Mwc_api:** En esta aplicación hemos añadido todo lo referente a la API propia de nuestro servicio. Las llamadas que se realizan de manera asíncrona desde el navegador son procesadas por este módulo. A su vez, éste se comunica con las aplicaciones Mwc_dropbox y Mwc_drive para procesar los datos y devolverlos en formato JSON.

3.2.4. Estructura de la base de datos

Esta aplicación web hace uso de una base de datos bastante sencilla, ya que en ningún momento se almacena la información de archivos ni carpetas de los servicios. Por lo tanto, hemos optado por el siguiente modelo de base de datos

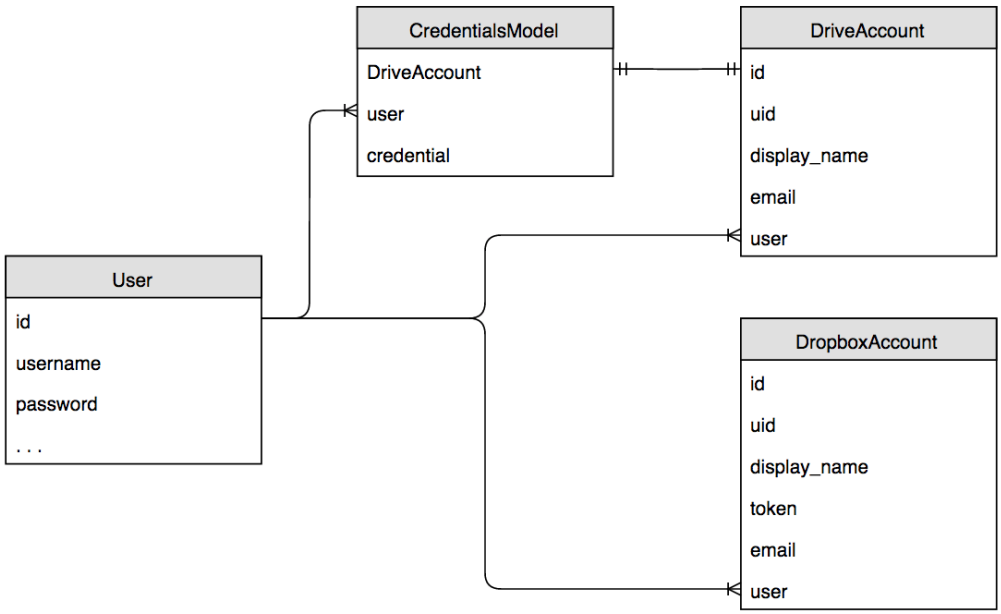


Figura 3.3: Diagrama de la base de datos. Elaboración propia

El diseño no es muy complejo, ya que la única información que necesitamos almacenar son las credenciales para acceder a cada servicio y la propia información del usuario, para no tener que pedir siempre estos datos.

Por otro lado, vemos que las tablas DropboxAccount y DriveAccount, tienen muchas similitudes: Ambas se relacionan con la tabla usuario, que podría tener tantas cuentas como quisiera, y guardan los datos de la información de cada cuenta con su identificador unico para cada servicio, uid. En cambio, almacenan de manera distinta las credenciales de los servicios de terceros: En el caso de Dropbox almacenamos directamente el token único, mientras que en Google Drive hemos creado otro modelo que será el encargado de manejar las credenciales del mismo. Esta decisión la hemos tomado para abstraernos de la complejidad *bajo nivel* y poder manejar todo lo referente a estas credenciales en esta clase; ya que, en el caso de Drive no obtenemos un token único como tal sino que los tokens pueden ser modificados y debemos tenerlo en cuenta.

3.2.5. Integración de servicios de terceros

En esta sección vamos a comentar la aproximación empleada para poder integrar las distintas cuentas que pueda tener el usuario y manejarlas dentro de nuestra aplicación. En este caso hemos optado por utilizar el SDK que proveen ambos servicios para trabajar con su API desde *Python*.

Un SDK o *Software Development Kit* es un conjunto de herramientas que facilitan al desarrollador la integración o programación de un sistema. Por ejemplo, Android pone su SDK al servicio de los desarrolladores para crear aplicaciones móviles. En nuestro caso, tanto Dropbox como Google Drive, nos entregan un SDK con el que podemos incorporar sus soluciones.

Manejar un API HTTP a bajo nivel suele ser bastante repetitivo y pesado: tendríamos que lidiar con los requerimientos de la misma consultando detalles de la documentación como tokens de autenticación y formato del cuerpo de las peticiones etcétera. En cambio, al utilizar un SDK obtenemos una interfaz de programación mucho más manejable[1], al estar escrita en el lenguaje que empleamos, en este caso Python, utilizarla es tan sencillo como realizar llamadas a las funciones que nos prevee con los parámetros indicados.

A continuación, pasaremos a comentar cómo hemos llevado a cabo esta integración para cada uno de los servicios pero primero, debemos obtener el permiso del usuario para acceder a sus datos. Para resolver este problema, hemos hecho uso de un estandar: OAuth2

OAuth2

Para comprender la necesidad de este estandar, vamos a proponer un caso práctico: Imaginemos que queremos añadir una cuenta de Dropbox nueva al servicio y éste, de manera totalmente inocente, nos solicita nuestro email y contraseña para acceder a nuestros datos. Sería bastante sospechoso, ¿no?. Al entregar nuestra contraseña a un tercero, nunca sabríamos dónde han ido a parar estas claves, cuántos servicios están accediendo a nuestros datos. Aún así, suponiendo que la intenciones de los desarrolladores son buenas, al realizar un cambio de nuestra contraseña, todas las aplicaciones de terceros conectadas perderían su acceso.

Ahora, se nos presenta otro escenario bastante diferente: Cuando queramos incluir una nueva cuenta de Dropbox en el servicio, éste nos redirigirá a la página de Dropbox donde podremos

[1] Diferencias entre API y SDK: stackoverflow.com

acceder como habitualmente lo hacemos, sin entregar la contraseña a un tercero, ya que en todo momento nos estamos autenticando en el servicio como tal.

Para conseguir el segundo caso, donde nuestra seguridad y datos personales no puedan quedar comprometidos, necesitamos hacer uso de OAuth2. Este estandar es mundialmente utilizado para realizar este tipo de autenticación y el conocido *social login*. Su especificación[2] está centrada en mantener la interoperabilidad y seguridad, por lo que consta de varios pasos pero cada uno de ellos es bastante sencillo.

Es necesaria una configuración inicial donde el desarrollador tiene que registrar su aplicación en el servicio correspondiente, que nos otorgará unas claves - Normalmente *APP Key* y *Secret Key* - que serán nuestras *contraseñas* para poder comenzar el proceso. Una vez tenemos estas credenciales, cuando un usuario quiera hacer *login* utilizando una cuenta de terceros, por ejemplo Google Drive, se iniciará el proceso de la siguiente manera:

1. Nuestro servidor comenzará una *conversación* con un servidor de Google para obtener la URL de autenticación, durante la misma, le indicaremos dónde redirigir al usuario cuando el proceso esté completado.

2. Una vez obtenida la URL de autenticación, simplemente conduciremos al usuario a la misma.

3. El usuario introducirá los datos necesarios para la autenticación en Google Drive, tras esto, se generará una clave aleatoria que identificará inequívocamente a ese usuario y a nuestra aplicación al mismo tiempo, normalmente llamado token o credenciales.

4. Tras completar el proceso anterior, serán los servidores de Google los que redirigirán al usuario a nuestra URL de retorno, haciéndonos llegar este token, que debemos almacenar para hacer uso de él próximamente.

Cabe destacar que el usuario en cualquier momento puede revocar el token de acceso a su información eliminándolo en el servicio correspondiente, por lo que debemos hacer las comprobaciones pertinentes antes de utilizarlo.

[2]Especificación de OAuth2: www.oauth.net

Dropbox

Uno de los servicios de almacenamiento online más conocidos es Dropbox, cuenta con más de 300 millones de usuarios y es, según la revista Fortune[3] el servicio más utilizado con diferencia. De cara a los desarrolladores, provee un SDK muy bien documentado, donde podemos consultar cómo realizar la integración con el servicio y las distintas funciones que podemos utilizar. En general el API es muy práctico ya que, por ejemplo, podemos consultar las carpetas del usuario por su ruta, como si se tratara de una navegación y subir ficheros a una ruta concreta.

Como hemos comentado anteriormente, hemos utilizado el SDK desarrollado por Dropbox para Python. Lo podemos encontrar como cualquier paquete de python, instalándolo mediante el comando:

```
>> pip install dropbox
```

Podríamos destacar que este paquete puede ser usado desde cualquier aplicación basada en Python, por lo que podríamos incluso crear una versión de escritorio reutilizando parte del código empleado para Django.

Para poder comenzar las peticiones, como hemos comentado en la sección de OAuth2, debemos registrar nuestra aplicación en el portal de desarrolladores de Dropbox y obtener nuestro *APP Key* y *APP Secret*. Una vez obtenido el acceso, debemos utilizar esas credenciales para conseguir los tokens de usuario. Para esto, utilizamos el método *start* de clase *DropboxOAuth2Flow*, que nos permite comenzar un flujo OAuth2 asignando una URL de regreso y nos genera la ruta de login para redirigir al usuario.

Tras el ingreso del usuario, Dropbox lo redirigirá a la URL que hayamos indicado, enviándonos el token de acceso en la propia petición HTTP. Utilizando ahora el método *finish* de la misma clase y enviando como parámetros la sesión y los parámetros GET que recibimos en la petición, obtendremos finalmente el token del usuario.

Con este token ya podremos acceder a los recursos del usuario. Para poder interactuar con estos, haremos uso de la clase del SDK *DropboxClient*, que recibe como parámetro en el inicializador el token de acceso. Una vez tengamos la instancia de *DropboxClient*, podremos hacer uso de los distintos métodos, veámos algunos de los que hemos empleado:

[3]Who's winning the consumer cloud storage wars: Revista Fortune

- ***DropboxClient.account_info*** Este método, nos da acceso a los datos del usuario, como su identificador único, datos personales y su email. Estos datos los guardamos para añadirlos a nuestra base de datos y poder mostrar al usuario los datos de sus distintas cuentas. Retorna un objeto JSON con la estructura:

```
{
  "uid": 12345678,
  "display_name": "John User",
  "name_details": {...},
  "referral_link": "https://www.dropbox.com/referrals/r1a2n3d4m5s6t7",
  "country": "US",
  "locale": "en",
  "is_paired": false,
  "team": {
      "name": "Acme Inc.",
      "team_id": "dbtid:1234abcd"
  },
  ...
}
```

- ***DropboxClient.metadata(path)*** nos permite obtener información de una ruta. Una gran ventaja frente al API de Google Drive, es que podemos ir avanzando entre las distintas carpetas del usuario, siguiendo la navegación. Como respuesta, obtenemos otro JSON:

```
  {
      "modified": "Wed, 27 Apr 2011 22:18:51 +0000",
      "path": "/Photos",
      "is_dir": true,
      "root": "dropbox",
      "contents": [
          {
              "size": "2.3 MB",
              "modified": "Mon, 07 Apr 2014 23:13:16 +0000",
              "path": "/Photos/flower.jpg",
              "is_dir": false,
              "root": "dropbox",
```

```
            "mime_type": "image/jpeg",
            ...
        }
    ],
    ...
}
```

- ***DropboxClient.put_file(file_path, file, overwrite=False)*** Este método lo utilizamos para cargar un fichero en la ruta *file_path* indicada. Por defecto no sobreescribe el fichero, pero podemos modificar este comportamiento indicando *overwrite=True*. Nos devovlerá un JSON con los datos del nuevo fichero:

```
{
    "size": "225.4KB",
    "modified": "Tue, 19 Jul 2011 21:55:38 +0000",
    "path": "/Getting_Started.pdf",
    "is_dir": false,
    "icon": "page_white_acrobat",
    "root": "dropbox",
    "mime_type": "application/pdf",
    "revision": 220823,
    ...
}
```

Gracias a estos métodos, hemos conseguido consultar y gestionar la información del usuario así como la subida de nuevos archivos. Toda la documentación la hemos obtenido en el portal de desarrolladores de Dropbox[4], donde podemos ver más detalles sobre las peticiones y las respuestas HTTP.

El propio SDK de Dropbox es el encargado de realizar la conversión de JSON a Python, generando un diccionario con el que podremos acceder a cada uno de los campos de la respuesta a través de su clave.

[4]Documentación del SDK para Python: www.dropbox.com/developers

Google Drive

Como hemos comentado en la introducción, Google Drive es el segundo servicio de almacenamiento en la nube más utilizado. En cuanto a las herramientas ofrecidas por Google, encontramos un API desde la que podemos hacer todo tipo de operaciones con Drive, aunque la documentación e implementación no es tan clara como en el caso de Dropbox.

Al igual que con Dropbox, hemos instalado el SDK como un paquete Python:

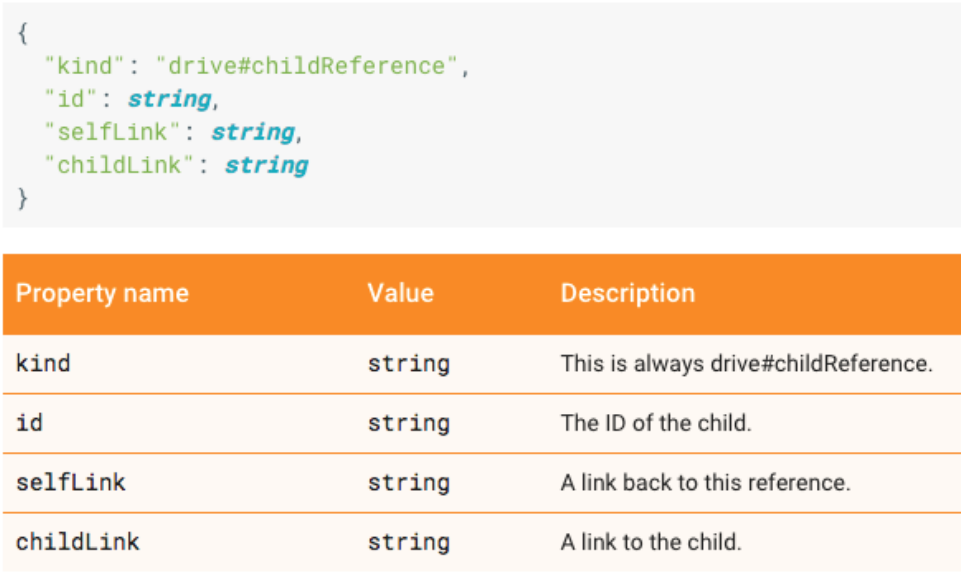
```
>> pip install google-api-python-client
```

En este caso, tenemos varios módulos, entre los que destacan ***OAuth2client*** y ***Apiclient***. El primero es el encargado de manejar la autenticación con OAuth y el almacenamiento en la base de datos de las credenciales mediante la clase ***CredentialsField***. El segundo, nos permite realizar peticiones a la API mediante la función *apiclient.discover.build*. Aquí encontramos uno de los mayores problemas, este cliente puede manejar varias de las APIs de Google, así que no está especializado. Por tanto, para realizar una simple consulta de, por ejemplo, el espacio disponible de un usuario, tendremos que realizar los siguientes pasos:

```
# Obtenemos las credenciales de la base de datos
credentials_model = CredentialsModel.objects.get(drive_account=self.pk)
credentials = credentials_model.credential

# Construimos una petición http y la autorizamos con las credenciales
http = httplib2.Http()
http = credentials.authorize(http)

# Creamos una petición a la api de 'drive' versión 2
drive_api = build('drive', 'v2', http=http)

# Realizamos una petición GET al endpoint del api 'about'
quota_info = drive_api.about().get().execute()
```

Como vemos, el estilo es mucho menos amigable que en el caso de Dropbox, donde podíamos llamar a métodos mucho más claros. Si indagamos en la documentación, encontramos

varios métodos que podrían ser de utilidad, como pueden ser los endpoints *Children* y *Parents*, pero ambos devuelven una estructura sin mucho sentido, ya que no podemos obtener información detallada de esos elementos, sino que recibimos identificadores de estos recursos, por lo que para poder ver el nombre, deberíamos realizar una petición por cada recurso devuelto.



```
{
  "kind": "drive#childReference",
  "id": string,
  "selfLink": string,
  "childLink": string
}
```

Property name	Value	Description
kind	string	This is always drive#childReference.
id	string	The ID of the child.
selfLink	string	A link back to this reference.
childLink	string	A link to the child.

Figura 3.4: Documentación para el endpoint *Children*. Google Developers

Así pués, hemos tenido que emplear otras llamadas a la API para poder filtrar los elementos por carpetas. Veámos algunos de estos métodos:

- ***drive_api.about().get()*** Al igual que en Dropbox, con este método obtenemos información sobre la cuenta del usuario, su identificador y otros datos personales. Éstos, los almacenamos para añadirlos a nuestra base de datos y poder mostrar al usuario los datos de sus distintas cuentas. Retorna un objeto JSON con la estructura:

```
{
  "kind": "drive#about",
  "selfLink": string,
  "name": string,
  "user": {
    "kind": "drive#user",
    "displayName": string,
    "picture": {
      "url": string
```

```
    },
    "isAuthenticatedUser": boolean,
    "permissionId": string,
    "emailAddress": string
  },
  "quotaBytesTotal": long,
  "quotaBytesUsed": long,
  ...
}
```

- ***drive_api.service.files().list(maxResults=1000).execute()*** Una de las funciónes más utilizadas, ya que la empleamos para obtener los ficheros del usuario. Como hemos comentado anteriormente, no tenemos opción de navegar por carpetas, por lo que siempre solicitamos el listado de todos los ficheros del usuario y los ordenamos en nuestro servidor para ofrecer al cliente una interfaz más cómoda, permitiendo esta navegación. Como respuesta, en el JSON, recibiremos un listado con objetos del tipo *drive#file*:

```
{
  "kind": "drive#file",
  "id": string,
  "title": string,
  "mimeType": string,
  "description": string,
  "createdDate": datetime,
  "modifiedDate": datetime,
  "modifiedByMeDate": datetime,
  "parents": [
    parents Resource
  ],
  "downloadUrl": string,
  "permissions": [
    permissions Resource
  ],
  "md5Checksum": string,
  "fileSize": long,
  "owners": [{...}, ...],
```

```
    ...
}
```

Al igual que con Dropbox, como referencia hemos empleado el portal de desarrolladores de Google Drive[5], donde podemos ver más información sobre los endpoints HTTP, que nos servirán para extrapolar al SDK, ya que no tiene documentación propia.

Además, el SDK de Google también ofrece la conversión de objeto JSON a diccionario de Python, por lo que en este sentido sí que ofrece una opción más cómoda para el desarrollador.

3.2.6. Creación del API

De cara a los distintos clientes que la aplicación pudiera tener, hemos decidido crear un API que permita el acceso a los datos del usuario a través de una interfaz común. De esta manera, el desarrollo en el lado cliente será siempre el mismo, sin tener en cuenta a qué servicio estamos accediendo, además, se podría añadir compatibilidad con más servicios de terceros, sin tener que realizar cambios en el código de cliente.

Por tanto, para contruir esta interfaz común nos hemos basado en los principios REST explicados anteriormente, aunque no hemos seguido completamente el estándar. En nuestro caso, añadimos la acción en la url, por tanto veámos qué endpoints hemos expuesto en nuestra API.

Endpoints

- **api/path/**

  Este endpoint es el encargado de mostrar la carpeta raíz del usuario. Aquí, hemos decidido incluir todos los archivos y carpetas que el usuario tiene en la raíz de cada servicio.

- **api/path/(service_class)/(account_id)/(path)**

  Al igual que el anterior, esta acción retorna los archivos y carpetas, pero recibe de qué cuenta se trata y en qué ruta queremos buscar.

- **api/download/(service_class)/(account_id)/(path)**

  En esta ocasión, permitimos al cliente descargar los archivos en esta url, que descarga el archivo del servicio y lo devuelve como respuesta.

[5]Portal de desarrolladores de Google Drive: https://developers.google.com/

- **api/upload/**

  Al crear una carpeta raíz *virtual* con todos los servicios del usuario, también permitimos que el usuario cargue archivos en la misma. Nuestro enfoque ha sido que nuestra aplicación sea inteligente y, dependiendo de la capacidad disponible en cada servicio, cargue el archivo en la que más espacio tenga.

- **api/upload/(service_class)/(account_id)/(path)**

  Este endpoint, como el anterior, maneja la carga de archivos, pero sí que indicamos el servicio y la carpeta a la que queremos cargar.

- **api/delete_account/(service_class)/(account_id)/**

  En esta URL, permitimos eliminar una cuenta. Este borrado se realiza tanto en la base de datos como revocando las claves tanto en Dropbox como en Google Drive.

3.2.7. Estructura completa del servidor

Tras ver todo el desarrollo en la parte servidora, podemos hacer un resumen de la estructura mucho más detallada del mismo.

Infraestructura

Como hemos podido ver en la sección de tecnologías empleadas en el servidor, para poner nuestra aplicación en producción haremos uso de Nginx[6] para recibir las peticiones, Gunicorn[7] se encargará de realizar la gestión de *web workers* de Django y Supervisor[8] mantendrá estos procesos siempre activos.

[6]Nginx: Subsección 2.3.4

[7]Gunicorn: Subsección 2.3.5

[8]Supervisor: Subsección 2.3.6

Aplicación web

En el servidor de producción, tendremos varias instancias de Django activas, los *web workers*, que normalmente están a la espera de recibir una petición por parte de Gunicorn. Cuando recibimos una petición, Django mediante la configuración del archivo urls.py encamina la misma a una función que hayamos designado. Si ponemos como ejemplo una petición a la API, ésta será encaminada hacia una función del archivo **mwc_api/views.py**, que se comunicará con los modelos definidos en mwc_dropbox o mwc_drive, donde, normalmente, utilizaríamos la API de estos servicios para conseguir la información solicitada.

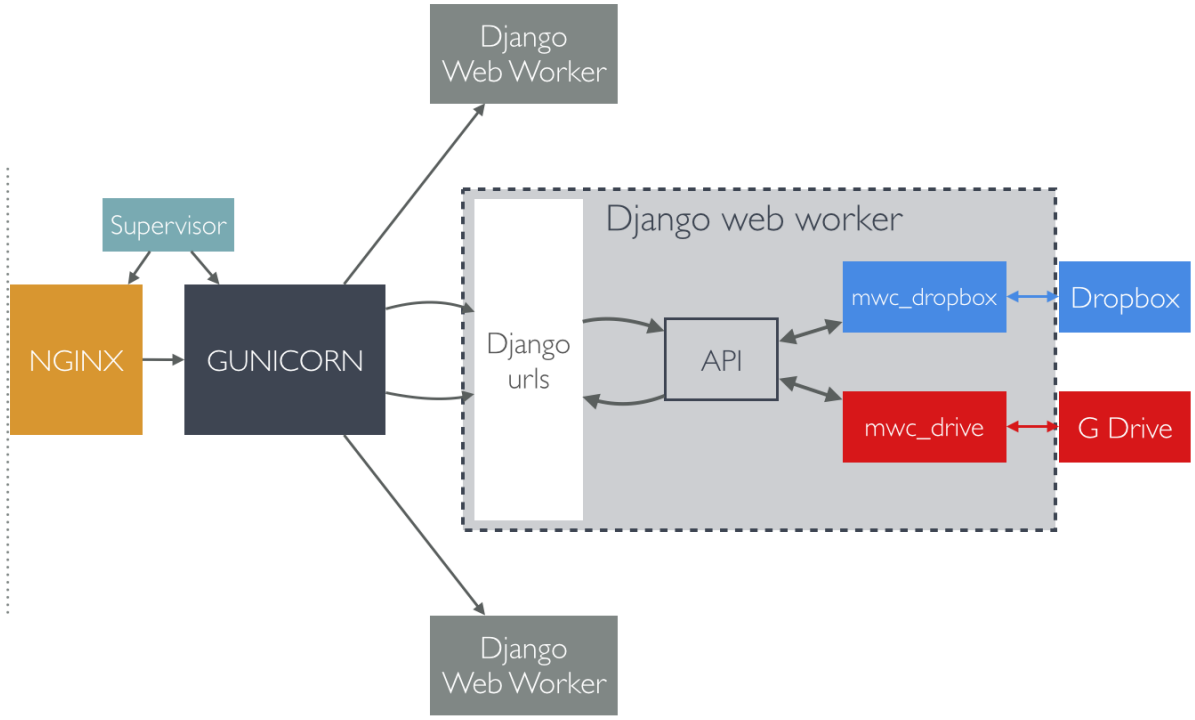


Figura 3.5: Estructura final del servidor. Elaboración propia

3.3. Desarrollo en el lado cliente

En el lado cliente, como cabe esperar, el desarrollo se ha realizado con HTML, CSS y JavaScript. Nuestra aplicación cliente ha sufrido varios cambios a lo largo de la implementación. Tanto la interfaz en sí como el código de cliente han sufrido cambios bastante drásticos, ya que también hemos querido ofrecer un diseño y rendimiento a la altura de una aplicación profesional.

Así pués, en las próximas secciones vamos a profundizar más en estos cambios y haremos un recorrido por todas las etapas sobre las que hemos ido iterando hasta dar con la opción final.

3.3.1. HTML5 y Django templates

Comencemos por la estructura de nuestras páginas. En este caso, hemos utilizado la sintaxis de HTML5 y el sistema de plantillas de Django.

El sistema de plantillas permite realizar iteraciones sobre variables, condicionales, mostrarlas, etc. tras haverlas generado en nuestras *vistas* de Django. Por ejemplo: para mostrar todas las cuentas de un usuario utilizamos un loop que recore todos los tipos de servicio y, dentro, recorremos cada una de las cuentas mostrando la información de la misma.

3.3.2. CSS3, mobile first y responsive design

En cuanto al diseño de la aplicación, comenzamos desarrollando nuestro propio CSS desde cero, sin tener en cuenta dispositivos móviles ni el tamaño en general de la pantalla.

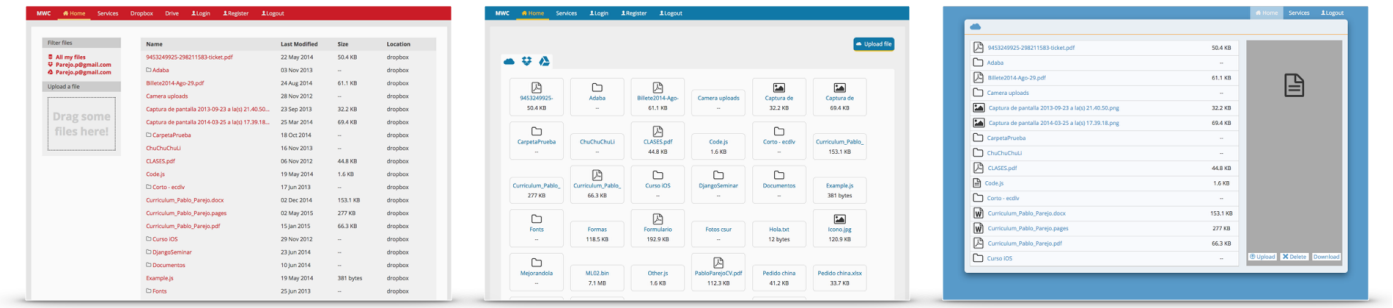


Figura 3.6: Primeras iteraciones en escritorio

Figura 3.7: Primeras iteraciones en móvil

Para solucionar este problema, utilizamos los media queries de CSS3, gracias a los cuales, podemos emplear un diseño u otro según el ancho del dispositivo. Este concepto es conocido como *Responsive Design*: tras la explosión de los smartphones y las smartTVs, casi todas las páginas web hoy día son visitadas desde pantallas con resoluciones completamente distintas: Móviles, ordenadores y televisiones. Por tanto, siguiendo con nuestro objetivo de crear una aplicación útil en cualquier dispositivo, nos centramos en esta idea.

El responsive design intenta que en todos los dispositivos la página tenga una funcionalidad plena y responda a las necesidades del usuario. El primer paso para conseguir una web responsiva fué generar un diseño elástico: Debe ajustarse al ancho del dispositivo y no superar los bordes de la pantalla. Si sólo tenemos un diseño elástico, gran parte de la web estará descolocada o demasiado encogida dependiendo del tamaño. Por lo tanto, nos interesa reorganizar la información dependiendo del dispositivo. En nuestro caso, además, hemos seguido la tendencia *mobile first*, basada en crear primero el diseño para el dispositivo móvil, es decir, con el ancho mínimo que queramos mostrar. Al no tener espacio de sobra, sólo añadimos lo realmente importante, lo que nos ayuda a hacer diseños más simples y útiles para el usuario.

3.3.3. Foundation, Stylus y utility classes

Tras intentar crear nuestro propio CSS desde la base, decidimos investigar para ver posibles opciones a la hora de trabajar con CSS que nos facilitaran el trabajo y permitieran un desarrollo más ágil. Por un lado, queríamos usar un framework, lo que nos llevó a la tesitura de decidir entre Bootstrap y Foundation. Ambos frameworks facilitan el uso de HTML gracias a sus clases predefinidas para trabajar con media queries y elementos repetitivos como botones o tablas. En nuestro caso nos inclinamos por Foundation porque creemos que ofrece herramientas muy útiles pero bastante genéricas, sobre las cuales podemos incluir nuestro diseño. En el caso de Bootstrap, la mayoría de los elementos tienen características fácilmente reconocibles, por lo que es relativamente sencillo aterrizar en dos páginas con un aspecto muy similar.

Por lo tanto, tras comenzar a utilizar estos frameworks, realizamos dos iteraciones más para conseguir nuestro diseño final. Queríamos un diseño simple e intuitivo, que mostrara la información relevante para el usuario de manera clara.

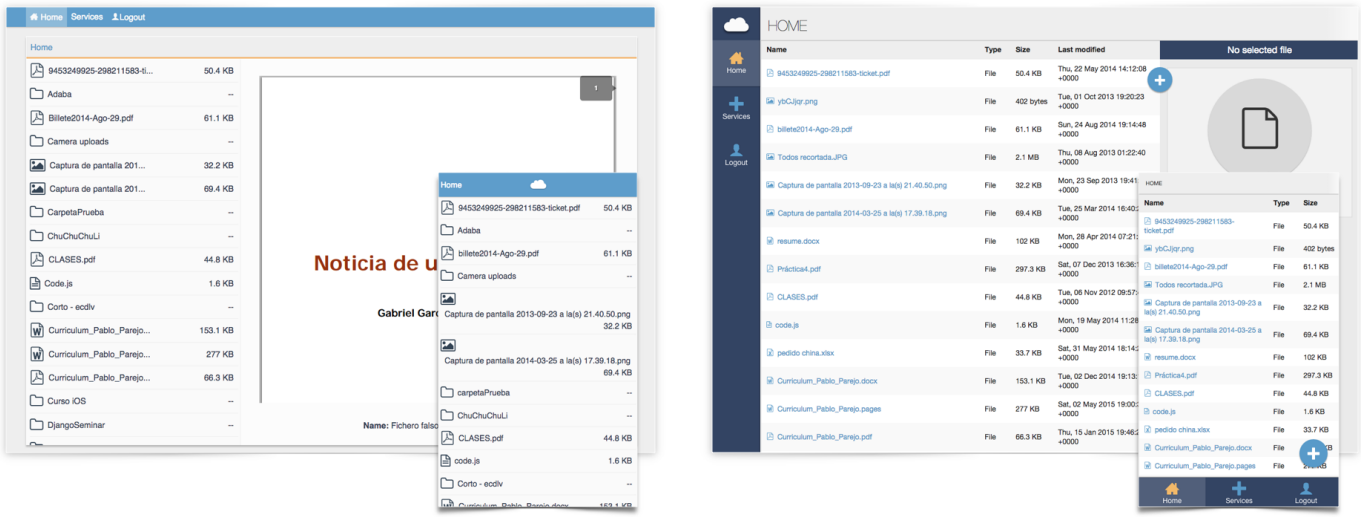


Figura 3.8: Diseños finales aplicando mobile first, Foundation y Stylus

Por otro lado, para escribir nuestro propio CSS decidimos emplear un precompilador. Hay varios disponibles, como son Less, Saas o Stylus, pero la verdad es que no encontramos grandes diferencias entre ellos, por lo que decidimos utilizar Stylus por tener una sintaxis similar a Python.

Cuando empleamos un precompilador, pasamos de escribir CSS a escribir pseudo-código.

En el caso de Stylus podemos crear funciones que reciben parámetros y generan código CSS; También es interesante la forma que tiene de anidar elementos, veamos un ejemplo:

```
.floating-action-button
  bottom 5rem
  border-radius 100%
  position fixed
  right 1.5rem
  z-index 99
  &:hover
    background mainColorHover
    bottom-shadow(3,3,0.35)
    color white
```

```
.floating-action-button {
  bottom: 5rem;
  border-radius: 100%;
  position: fixed;
  right: 1.5rem;
  z-index: 99;
}
.floating-action-button:hover {
  background: \#357cb2;
  -webkit-box-shadow: 0px 3px 3px rgba(0,0,0,0.35);
  box-shadow: 0px 3px 3px rgba(0,0,0,0.35);
  color: #fff;
}
```

Figura 3.9: Código Stylus y salida CSS. Elaboración propia

En este caso, hemos empleado una función *bottom-shadow*, que recibe los parámetros de la sombra y genera el código CSS compatible para todos los navegadores. Además, podemos ver que las relaciones de los elementos se realizan mediante el indentado. Además, gracias al precompilador, podemos tener varios archivos con código stylus y/o CSS, que se agruparán ofreciendo un solo archivo CSS para utilizar en producción y reducir el número de peticiones a realizar por el cliente web.

Basándonos en Foundation y sus *utility classes*[9], hemos decidido crearnos nuestro propio *framework* para reutilizar en futuros desarrollos. Así pués, nuestro código Stylus utiliza variables para los colores primarios y secundarios. Además hemos generado clases con una función muy específica, como las encargadas de añadir un margen a un elemento, añadirle padding, etc.; Por otro lado, hemos diseñado elementos reutilizables como el *floating action button*[10].

[9]Foundation Utility Classes: http://foundation.zurb.com/docs/

Nuestro diseño final está inspirado en las especificaciones de Material Design, cuyo creador es Google, y lo recomienda para todo tipo de aplicaciones ya que, según defienden, facilita al usuario la interacción ya que se basa en principios del mundo físico[11] para crear interfaces más naturales.

3.3.4. JavaScript y single-page applications

Como explicamos anteriormente, JavaScript es el lenguaje de programación de la web. Para esta aplicación, donde necesitamos una interacción contínua por parte del usuario y una respuesta inmediata a sus acciones, tenemos que emplear JavaScript, un lenguaje asíncrono y basado totalmente en eventos, para dar cabida a esta funcionalidad.

Este tipo de aplicaciones web, se basan en realizar llamadas asíncronas, donde se pedirá parte de la interfaz a intercambiar o datos de un API (como es nuestro caso). Al realizar este tipo de llamadas, la carga es mucho más rápida ya que tan solo estamos recibiendo los datos que necesitamos y no la página completa con los elementos estáticos. Últimamente, gran parte de las páginas web realizan peticiones asíncronas[12], ya sea para cargar datos nuevos, envío de formularios, etc. Estas aplicaciones, se denominan *Single-Page Application*[13]

Al igual que a la hora de elegir framework para el lado servidor, nos decantamos por un framework que resolviera los problemas repetitivos y de bajo nivel como es Django, en este caso, tomamos la misma decisión. El gran problema es que en el mundo de JavaScript tenemos gran cantidad a nuestra disposición. Frameworks como EmberJS, ReactJS, Backbone, AngularJS o jQuery son algunos de los más conocidos. Nuestra primera elección fue jQuery, ya que era el único que nos era familiar, desconocíamos AngularJS, y los demás parecián bastante complejos.

[10] Material Design – Floating action button http://www.google.com/design/

[11] Material Design – Material Properties http://www.google.com/design/

[12] Tecnimedios: Single Page Application (SPA) una tendencia creciente http://tecnimedios.com

[13] Wikipedia – Single Page Application es.wikipedia.org

3.3.5. JQuery

Nuestra primera opción fue jQuery, una librería bastante sencilla y con muchas opciones que nos pueden servir. Sobre todo nos facilita el trabajo y manejo de los elementos del DOM. La curva de aprendizaje es bastante suave, ya que la documentación es bastante buena y podemos encontrar muchísimos ejemplos en la web.

El gran problema de esta librería es que no ofrece herramientas definidas para separar nuestra lógica de la aplicación. Durante el desarrollo con la misma, hemos creado un código poco ordenado y con bastantes errores. Fue en este punto cuando decidimos apostar por un framework más avanzado y potente, lo que nos llevó a comenzar desde cero con AngularJS.

3.3.6. AngularJS

Tras nuestra experiencia con jQuery, decidimos investigar para encontrar otros fameworks que se adaptaran más a nuestras necesidades y nos permitieran tener un código más ordenado y mantenible.

Así pués, encontramos artículos[14], entradas en blogs[15] e incluso preguntas en StackOverflow[16], parecía que era un buen cambio y nos permitiría volver a la dirección correcta.

AngularJS permite un diseño mucho más empaquetado y reutilizable, ya que podemos crear disintos módulos con nuestro código. De esta forma, hemos separado nuestra aplicación siguiendo el principio MVC, donde tneemos unos servicios (Modelo) que son los encargados de interactuar con el servidor, los controladores y las plantillas (html y directivas) de Angular.

Plantillas y directivas

En este framework, tenemos un sistema de plantillas muy similar a Django, ya que tienen un lenguaje bastante parecido. De esta forma, podremos utilizar valores de variables que estén definidas en el *Scope*. El *Scope* es un elemento intermedio entre nuestras vistas y controladores que se encarga de comunicar ambos elementos.

[14] Building Rich Web Apps: jQuery & MVC vs AngularJS, John Culviner http://superdevelopment.com/

[15] ¿Qué es AngularJS? Primeros pasos para aprenderlo, Carlos Azaustre https://carlosazaustre.es

[16] What does AngularJS do better than jQuery? http://stackoverflow.com

Por otro lado, tenemos las directivas, estos bloques pueden ser utilizados para construir bloques de HTML y JavaScript encapsulados[17], que tengan una funcionalidad definida. Además, Angular pone a nuestra disposición directivas predefinidas, utilizables en cualquier elemento HTML, veamos algunas de ellas:

- ***ng-click:*** Una de las directivas más utilizadas, permite ejecutar una función definida en el Scope al realizar click en un elemento

```
<p ng-click="didClick()" > Click me!</p>
```

- ***ng-repeat:*** Como se intuye por su nombre, nos permite iterar por un array u objeto de JavaScript, con una sintaxis similar a los bucles de Python.

```
<li ng-repeat="{{element in array}}">{{element}}</li>
```

- ***ng-if:*** Permite generar o no código HTML dependiendo del valor de una variable.

```
<div ng-if="showDiv">
  [...]
</div>
```

Podemos encontrar gran cantidad de ellas en la documentación de AngularJS[18], donde además hay publicados recursos para aprender a utilizar este framework.

[17]Cómo pasar variables como atributos en directivas de AngularJS, Carlos Azaustre https://carlosazaustre.es
[18]Documentación de AgularJS https://docs.angularjs.org/api

Servicios

Como hemos comentado anteriormente, los servicios son los encargados de la conexión con el servidor, y actuarían como nuestro modelo. Por lo tanto, serán nuestro nexo de unión entre las APIs definidas en Django y nuestra aplicación cliente.

Los servicios suelen trabajar con *Promesas*. Es un concepto algo difícil de explicar, por lo que hemos decidido citar a Carlos Azaustre, que en una entrada[19] de su blog lo deja bastante claro:

> *Las promesas son objetos de JavaScript que sustituyen de alguna manera a los callbacks. Se utilizan cuando no se puede retornar el valor de una función porque aún no se conoce, pero no podemos dejar que bloqueen la función esperando a que llegue. El objeto promesa en un futuro contendrá el valor que esperamos retornar. Es complicado de entender en un principio, pero si conocemos como funciona, en ocasiones nos puede resultar útil.*

Como vemos, las promesas pueden servirnos para realizar llamadas asíncronas a funciones y recibir una respuesta cuando la ejecución haya terminado. Por esta característica, se suelen utilizar en los servicios, ya que son muy útiles para obtener datos del servidor y ejecutar código tras conseguir la respuesta.

Veamos un ejemplo de una de nuestras funciones para obtener la información de una carpeta

```
function listPath (path){
  /* Creamos un objeto deferred para trabajar con promesas */
  var deferred = $q.defer();
  $http.get(path)
    .success(function(data){
      deferred.resolve(data)
    })
    .error(function(err){
      deferred.reject(err)
    })
```

[19] Uso de promesas en AngularJS, Carlos Azaustre https://carlosazaustre.es/blog/

```
    /*
      Devolvemos una promesa pendiente.
      Este código se ejecuta antes que .success y .error,
      por lo que no bloqueamos la ejecución
    */
    return deferred.promise
  }
```

Como podemos observar, en el código anterior hacemos una petición a *path*, y configuramos dos callbacks: Una en caso de éxito y otra en caso de error. Estas funciones permanecen a la espera de una respuesta que las ejecute, por lo que no son bloqueantes. Por último, devolvemos un objeto promise.

Cuando recibamos respuesta del servidor, en caso de éxito, se ejecutará la fución que hemos implementado en *.success*, que simplemente resolverá la promesa y entregará los datos solicitados. En caso de fallo, devolveremos un error y la promesa será rechazada. Ambos casos serán manejados en el controlador.

Controladores

Los controladores de AngularJS trabajan muy estrechamente con servicios y el Scope. Normalmente, son los encargados de iniciar una petición a los servicios para obtener datos, además, se encargan de responder a los eventos producidos en las vistas y de incluir las variables correspondientes en el Scope para que las vistas las muestren.

La interacción con los servicios suele basarse en promesas que, como ya hemos visto, facilitan las peticiones asíncronas a un servidor. Por ejemplo, teniendo un servicio "service", podríamos llamar a la función anterior y manejar los datos de respuesta:

```
Service.listPath($scope.path)
  .then(function (data) {
    console.log(data)
    $scope.loading = false
    $scope.account = data
  })
  .catch(function(err){
    alert("Error:" + err)
  })
```

La función *.then* será ejecutada en caso de éxito, por lo que podríamos realizar las operaciones oportunas pero, en caso de error, la función que se ejecutará será *.catch*, donde recibimos el error.

Una característica muy interesante de Angular, aparte del two-way data binding que ya hemos comentado[20], es la posibilidad de observar una variable y ejecutar código cuando cambie su valor. En nuestro caso lo hemos utilizado para realizar una petición al servidor y actualizar el listado de archivos cuando cambie el valor de *$scope.path*

[20] AngularJS Subsección 2.4.7

3.4. Resumen y estructura final de la aplicación

Finalmente, vamos a explicar cómo interactúan entre sí ambas partes. Para conseguir una comunicación eficiente, cliente y servidor utilizan el API que hemos definido anteriormente. De esta forma, podríamos resumir el proceso de la siguiente manera:

1. **Acción por parte del usuario.** Cuando el usuario realiza una acción, ésta es capturada por los controladores que, típicamente, realizará una llamada a un servicio de AngularJS

2. **Petición HTTP:** El servicio será el encargado de realizar la petición al servidor y esperar una respuesta.

3. **Recepción de la petición:** Nuestro servidor de Django será el encargado de procesar la petición y realizar la acción solicitada. Normalmente, esta desencadenará otra petición a un servicio de terceros.

4. **Servicios externos:** Ahora serán los servidores externos los que procesen nuestra petición y nos devuelvan la información solicitada.

5. **Respuesta HTTP:** Tras recibir esta información, construiremos nuestra respuesta formateada – estandarizando ambos servicios – y la devolveremos a nuestro cliente

6. **Recepción en cliente:** Será el propio servicio AngularJS el que reciba los datos y los entregue al controlador

7. **Actualización de la vista:** El controlador, al cambiar los datos del *Scope*, estará modificando la vista y, por lo tanto, el usuario verá el resultado de su acción por pantalla

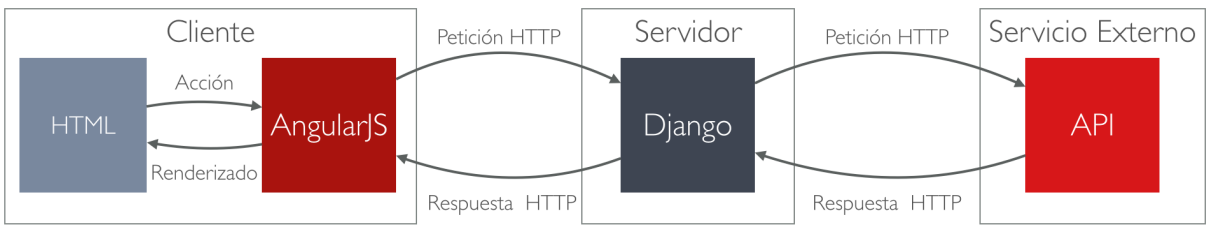


Figura 3.10: Recorrido de una petición HTTP y respuesta. Elaboración propia

Capítulo 4

Resultados

Para finalizar, queremos mostrar el resultado de nuestra aplicación, viendo la interfaz que hemos desarrollado para cada dispositivo. Además, haremos un repaso a las funcionalidades que presenta cada pantalla, así como la justificación de la disposición de los elementos que hemos realziado en función del tamaño del dispositivo.

4.1. Pantalla principal: Listado de elementos

En esta sección, es donde se encuentra el grueso de la aplicación, ya que es donde mostramos al usuario toda la información contenida en sus distintas cuentas. Como comentamos en los objetivos, nuestra idea es que para el usuario la integración sea totalmente transparente, por lo que mostraremos todos los archivos y carpetas que se encuentren en la carpeta raíz de sus cuentas, en un listado completo.

Partiendo desde el dispositivo móvil, nuestra interfaz se adapta a la perfección, ofreciendo una navegación sencilla mediante *migas de pan*, que permiten retroceder a carpetas anteriores. También en esta pantalla podremos cargar nuevos archivos y descargarlos si hacemos click en su nombre.

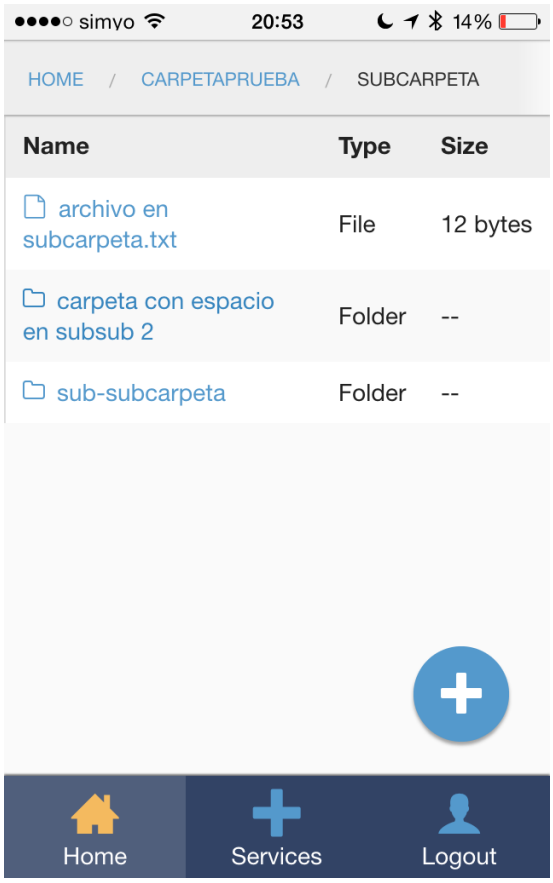


Figura 4.1: Listado de ficheros y carpetas en dispositivo móvil

El diseño en tablet es bastante similar, aunque decidimos incluir el menú en la barra lateral, adoptando un diseño más parecido a las aplicaciones nativas para este tipo de dispositivos. Tanto en móvil como tableta, podemos ver que el botón de acción principal es fácilmente alcanzable, lo que facilita el uso de la aplicación.

Por último, en el diseño de escritorio, hemos añadido una barra lateral para ver el detalle de un archivo. Además, tendremos dos botones: Uno para descargar el archivo y otro para eliminarlo.

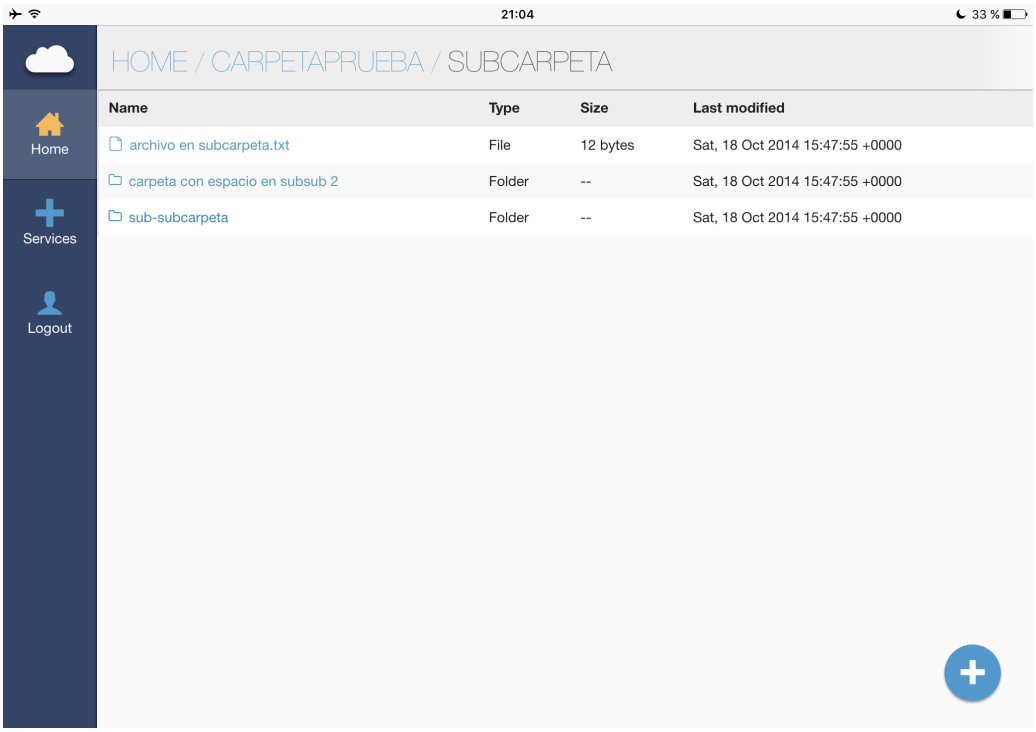


Figura 4.2: Listado de ficheros y carpetas en tableta

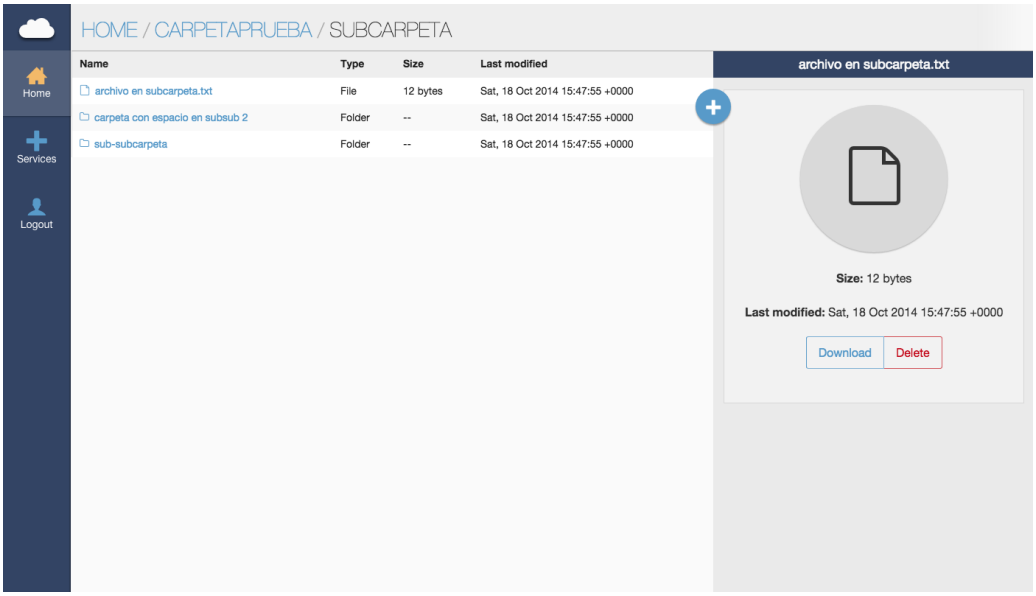


Figura 4.3: Listado de ficheros y carpetas en ordenador

4.2. Configuración de servicios

En este apartado, se encuentra la interfaz para realizar la gestión de las cuentas del usuario. Aquí podremos añadirlas o eliminarlas, de nuevo con una interfaz muy sencilla. Cabe destacar la gran diferencia entre la sencillez que presentamos al usuario y la relativa complejidad que supone añadir una cuenta de usuario nueva que, como hemos comentado anteriormente, se realiza gracias a OAuth2. Tras conectar una nueva cuenta, el usuario es redirigido a esta misma pantalla y obtendrá un aviso de que la cuenta se ha añadido de manera satisfactoria.

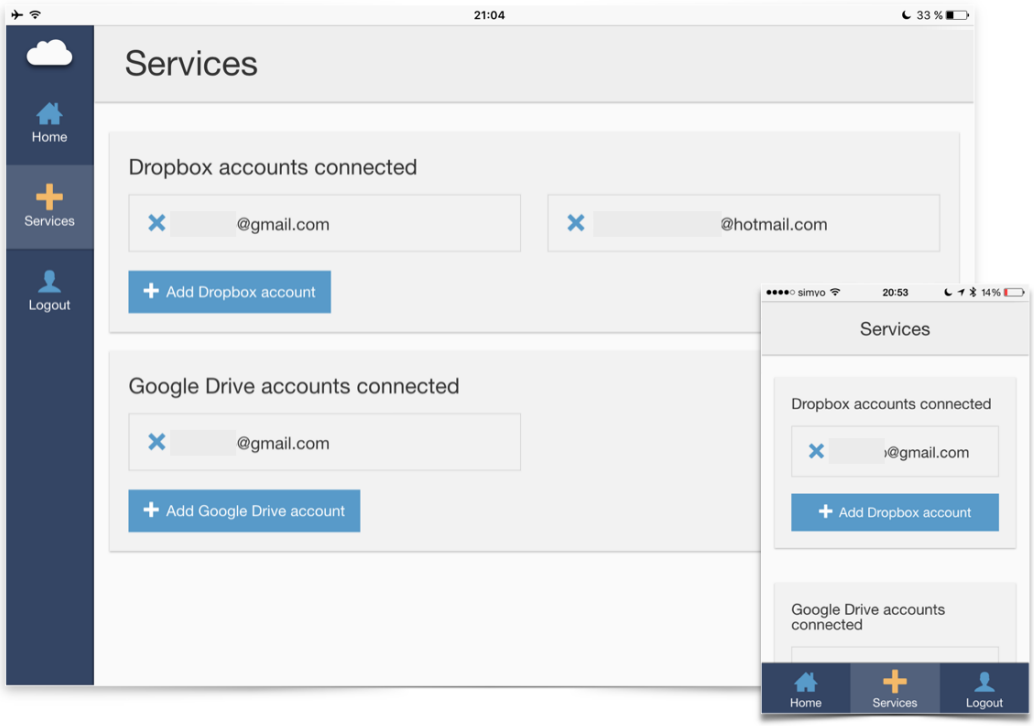


Figura 4.4: Gestión de cuentas en móvil y tableta

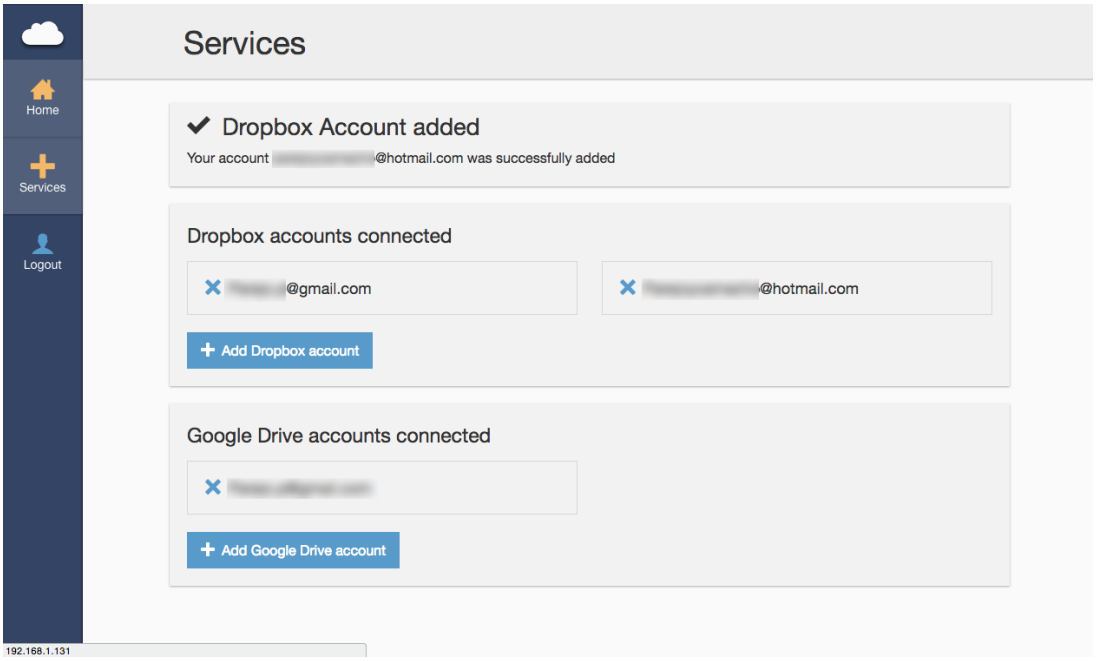


Figura 4.5: Confirmación de cuenta añadida con éxito

Capítulo 5

Conclusiones

En este último capítulo, presentamos, a modo de conclusión, los logros alcanzados durante el proyecto. Además, nos gustaría realizar un resumen de todo lo aprendido en el desarrollo del mismo, y las posibles mejoras que podrían realizarse a esta aplicación.

5.1. Logros

En la introducción de esta memoria, hemos establecido una serie de objetivos a llevar a cabo en el proyecto. Teniendo en cuenta los capítulos de Desarrollo e Implementación y Resultados, podemos hacer una valoración final de la consecución de los mismos.

1. **Integración de servicios de almacenamiento en una apliación:** Éste se planteaba como nuestro objetivo principal, ya que en él residía el grueso de nuestro proyecto. Este objetivo se ha cumplido satisfactoriamente, ya que hemos conseguido tener una aplicación totalmente funcional, capaz de integrar varias cuentas de distintos servicios de almacenamiento en una sola interfaz web. Además, la integración con estos servicios se ha realizado siguiendo los estándares definidos por la documentación y guías de desarrollo de las mismas.

2. **Uso de tecnologías web avanzadas:** Como hemos podido ver en anteriores capítulos, han sido varias las tecnologías con las que hemos tenido oportunidad de enfrentarnos, y otras cuantas que se han valorado para comprobar si nos podían ayudar a realizar un desarrollo más sencillo y correcto. Por tanto, el uso de tecnologías innovadoras y de

última generación se ha llevado a cabo durante todo el proyecto, siguiendo una filosofía pragmática e iterando sobre el código ya generado para conseguir esta meta.

3. **Uso de Git y Gitflow:** En este caso, no hemos podido llegar a usar toda la potencia de Gitflow, puesto que no nos ha sido posible realizar pruebas en un servidor real con la aplicación, donde podríamos haber empleado Gitflow para realizar un despliegue y corrección de errores desde Git. Aún así, sí que hemos hecho un uso bastante avanzado de Git, ya que hemos empleado ramas para las distintas versiones e iteraciones del servicio. Por otro lado, hemos utilizado GitHub para la publicación del código.

4. **Diseño de interfaz y experiencia de usuario:** Como se extrae del capítulo de resultados, hemos realizado una aplicación bastante fácil de utilizar y con un diseño adaptado a todos los dispositivos sobre los que nuestra aplicación podría ser utilizada. Cabe destacar aquí también el trabajo en cuanto a CSS llevado a cabo, donde hemos realizado varias iteraciones hasta llegar al diseño final.

5.2. Valoración final

Por último, vamos a realizar una valoración de todo lo aprendido en este proyecto, así como los conocimientos que hemos sido capaces de aplicar gracias a lo estudiado durante el trasncurso del Grado.

5.2.1. Conocimientos aplicados

Gracias a los conocimientos ya aprendidos, hemos sido capaces comenzar con una base de programación de Python y JavaScript que nos ha ayudado sobre todo en el inicio del proyecto. Por otro lado, para realizar el diseño de la aplicación, han sido imprescindibles los conceptos de HTTP y el diseño de aplicaciones cliente-servidor que empleamos en asignaturas como Laboratorio de Tecnologías Audiovisuales en la Web. También en esta asignatura, vimos una pequeña introducción a Django, aunque parte del desarrollo de este proyecto ya había comenzado en el inicio de la misma.

En general, este Grado me ha servido para descubrir algo que me apasiona como es la programación, y me ha ayudando a comprender conceptos antes totalmente desconocidos. Gran parte de este proyecto se ha basado en investigación y conseguir interactuar con elementos externos, lo que nos ha llevado a *pelearnos* con documentación y guías de desarrollo, algo que no podríamos haber hecho sin el afán investigador y la cultura inculcada durante la carrera.

5.2.2. Conocimientos aprendidos

No cabe duda de que este proyecto tiene un componente de investigación bastante alto, ya que muchas de las tecnologías empleadas no las hemos estudiado durante la carrera.

Por un lado, hemos tenido que usar frameworks basados en JavaScript. Comenzando por jQuery, donde decidimos dar un paso atrás y valorar otroas herramientas, donde Backbone o EmberJS fueron tenidos en cuenta a la hora de elegir una librería que facilitara el uso de MVC y permitiera obtener un desarrollo mejor estructurado. En esta valoración, finalmente decidimos aprender AngularJS, que gracias a recursos encontrados en Internet, hemos sido capaces de aplicar en este proyecto.

Por otro lado, también hemos aprendido Stylus, el preprocesador de CSS que nos ha permitido realizar un desarrollo más ágil; así como Foundation, que nos ha ayudado a simplificar nuestro diseño responsivo y adaptado a cualquier dispositivo.

Hemos aprendido muchísimo de APIs. Tanto la documentación de Dropbox como la de Google Drive, nos han permitido realizar una integración de ambos de la manera correcta. Gracias a los equipos encargados de estos servicios, hemos podido encontrar aplicaciones de ejemplo y descripciones detalladas de cómo realizar este desarrollo. También hemos aprendido a utilizar OAuth2, que está siendo utilizado en servicios como Twitter o Facebook, por lo que es una herramienta muy útil. También hemos investigado acerca de APIs REST, aunque nuestra implementación no ha sido del todo correcta, pero ahora tenemos mucho más claros los conceptos de su implementación y cómo podríamos desarrollar un API en el futuro.

Sobre todo, este proyecto me ha servido para enfrentarme de una forma más real con el mundo del desarrollo en general, ya que, como he comentado, gran parte del tiempo lo hemos invertido en profundicar en la documentación, tanto de frameworks – en servidor y cliente – como de APIs. Viendo todos los recursos y enlaces que hemos añadido lo largo de la memoria, creemos que la mejor escuela para aprender a programar web es la propia web. Por suerte,

disponemos de cientos de artículos y recursos en Internet, además de una enorme base de datos de dudas como es StackOverflow. Gracias a todos estos recursos hemos conseguido resolver muchas dudas, aprender de los errores y poder llegar a nuestra meta de manera satisfactoria.

5.3. Publicación del código

El código de este proyecto está disponible íntegramente en GitHub, en la siguiente url:
https://github.com/pabloparejo/finalYearProject

Bibliografía

[1] Alex Berson. *Client - server architecture*. Jay Ranade series on computer communications. McGraw-Hill, New York, NY, 1992.

[2] Bear Bibeault and Yehuda Kats. *jQuery in Action*. Dreamtech Press, 2008.

[3] Scott Chacon. *Pro git*. Apress, 2009.

[4] Benoit Chesneau. Gunicorn-python wsgi http server for unix.

[5] Douglas Crockford. The application/json media type for javascript object notation (json). 2006.

[6] Peter Bacon Darwin and Pawel Kozlowski. *AngularJS web application development*. Packt Publ., 2013.

[7] Juan Diego Gauchat. *El gran libro de HTML5, CSS3 y JavaScript*. Marcombo, 2012.

[8] Adrian Holovaty and Jacob Kaplan-Moss. *The definitive guide to Django: Web development done right*. Apress, 2009.

[9] Azat Mardan. Applying stylus, less, and sass. In *Pro Express. js*, pages 181–183. Springer, 2014.

[10] Mark Masse. *REST API design rulebook*. .º'Reilly Media, Inc.", 2011.

[11] Mark Pilgrim. *HTML5: up and running*. .º'Reilly Media, Inc.", 2010.

[12] Will Reese. Nginx: the high-performance web server and reverse proxy. *Linux Journal*, 2008(173):2, 2008.