Universidad de Las Palmas de Gran Canaria

Proyecto Final de Carrera

MiNube

Autor: Yarilo Villanueva Hernández

Tutor: Francisco Alayón Hernández

Ingeniería Informática

Octubre 2015

"*En toda nuestra historia nunca ha habido un momento como hoy en el que una parte tan grande de nuestra cultura fuera posesión de alguien. Y sin embargo jamás ha habido un momento en el que la concentración de poder para controlar los usos de la cultura se haya aceptado con menos preguntas que como ocurre hoy día*"

Lawrence Lessig – Por una cultura libre (2004)

Agradecimientos

En primer lugar, he de agradecer a mi tutor por su consejo, y guía a lo largo de todo este proyecto. Gracias especialmente por su paciencia y por ayudarme a tener la constancia necesaria para llegar hasta aquí.

A mis amigos, por haber estado ahí y haberme apoyado durante este camino, ya sea de forma directa o indirecta. A David, Aarón, Luis, Yeray, Iago, Héctor y tantos otros, sin los cuales este proceso habría sido mucho más duro.

Por otra parte, agradecer al personal de la administración de la escuela su apoyo y ayuda ante mis continuas dudas sobre el proceso burocrático indispensable para llevar este proyecto a buen puerto.

Finalmente, a mi familia por su confianza incondicional y soporte en todo momento. Ese apoyo, que difícilmente se puede cuantificar y al que intento humildemente hacer honor con estas palabras.

A todos, gracias.

Índice

Lista de tablas

Lista de Figuras

1 . Introducción

El presente documento tiene como objetivo exponer la memoria sobre el Proyecto Final de Carrera "Mi Nube" para la obtención del título de Ingeniero en Informática. Este proyecto es una solución de almacenamiento en la nube para usuarios finales enfocada en la facilidad de uso, privacidad y extensibilidad.

En los últimos años, la oferta de soluciones de almacenamiento y computación en la nube ha crecido de forma exponencial, ofreciendo a los usuarios múltiples herramientas para alojar y tratar sus archivos.

Sin embargo, este aumento de alternativas no ha venido acompañado de un mayor control y privacidad del usuario, más bien el contrario, el usuario se ha visto obligado cada vez más a ceder el dominio de sus datos a agentes externos. Si bien esta solución proporciona ventajas indiscutibles, desde el punto de vista de los trabajos de mantenimiento y disponibilidad hay ocasiones donde es deseable que la infraestructura de almacenamiento en la nube sea gestionada por la propia empresa, garantizando así la privacidad de los datos almacenados.

Este proyecto fin de carrera nace para intentar aportar estas características, ofreciendo una solución integral, tanto para el servidor como para el cliente, de sincronización y almacenamiento de archivos personales.

2 . Objetivos

El presente proyecto final de carrera tiene tres objetivos fundamentales:

1. Desarrollar una solución de almacenamiento en la nube, siguiendo un modelo cliente/servidor, con las siguientes características:
    - Almacenamiento de datos
    - Gestión de usuarios
    - Encriptación de datos personal (las claves sólo las conoce el usuario)
    - Sincronización con repositorios locales
    - Fácil instalación para usuarios finales, tanto en el lado cliente como en el lado del servidor
    - Cliente compatible con Windows, GNU/Linux y Mac OS X
    - Código libre
    - Los usuarios podrán compartir archivos con otros usuarios sin intervención de ninguna clase de autoridad central.
2. Entender la creación y mantenimiento de una aplicación web moderna en cada una de sus fases y a todos los niveles: Desde la configuración y optimización del servidor, pasando por la creación de una API, hasta el desarrollo de la interfaz web y la interacción del usuario con la misma.
3. Facilitar el uso de tecnologías en la nube que respeten la privacidad y libertad del usuario.

En consonancia con el último objetivo, se ha implementado el proyecto usando en la medida de lo posible soluciones de software libre.

3 . Estructura del documento

Tras la introducción esta memoria presenta el Estado del Arte, analizando las características del almacenamiento en la nube y las soluciones existentes actualmente.

En segundo lugar se describirá La Metodología y Planificación usadas, enumerando por horas las tareas realizada así como describiendo la metodología empleada a la hora de abordar las mismas.

En la siguiente sección, Recursos, se enumerarán y detallarán los recursos hardware y software usados para la finalización de este proyecto.

Posteriormente, en la sección de Análisis, se explicarán los requisitos necesarios para la consecución del proyecto. De igual forma, se describirán los actores y casos de uso envueltos en la interacción con la solución realizada.

Una vez expuesto el Análisis del proyecto, se procederá a explicar el diseño de la solución aplicada, dividiéndose ésta en dos secciones fundamentales: El servidor y el cliente, desglosándose este último en el cliente web y cliente de escritorio.

Tras la explicación de la solución implementada, se explicarán los costes en los que se ha incurrido en la sección homónima.

Finalmente, se procederá a explicar las conclusiones aportadas tras la realización del Proyecto Final de Carrera, así como el trabajo futuro a realizar sobre el mismo.

Como nota adicional, la plataforma desarrollada se ha llamado *Dandelion*. Este nombre será usado a lo largo del documento para referirse a la solución de almacenamiento implementada en el presente proyecto.

4 . Estado del Arte

A continuación se presentará el estado del arte exponiendo el campo en el que se basa el presente proyecto: El almacenamiento en la nube, o almacenamiento online, dividiendo el mismo en soluciones privadas y soluciones públicas o abiertas.

4.1 Almacenamiento online

Se conoce como almacenamiento online como aquel servicio online que permite al usuario guardar sus archivos en un servidor remoto de manera que puedan estar accesibles desde cualquier lugar. A diferencia del almacenamiento tradicional, el almacenamiento online suele tener las siguientes ventajas:

- **Accesibilidad**: Los datos almacenados pueden ser consultados desde cualquier lugar con conexión a internet.
- **Sincronización:** Sincronizando el mismo contenido en varios dispositivos a la vez, eliminando por tanto la dependencia con un sólo dispositivo o disco duro.
- **Seguridad**: Los datos almacenados suelen encriptarse en el lado servidor.
- **Redundancia de los datos**: Se pueden tener varias copias del mismo archivo a lo largo de varios dispositivos o clientes, lo que ofrece una menor probabilidad de perder el mismo. Por otra parte, la mayoría de los servicios de almacenamiento online copian de forma redundante los datos en el lado servidor, disminuyendo enormemente la posibilidad de pérdida.

Por su contra, el hecho de usar este tipo de almacenamiento ofrece dos grandes desventajas:

- **Control**: El control de los archivos en la mayoría de soluciones se transfiere a un proveedor externo, teniendo que relegar en los mismos para las ventajas antes mencionadas, especialmente para la seguridad y tratamiento de los archivos.
- **Complejidad**: Las soluciones que ofrecen más control suelen ser más complejas de instalar y administrar para el usuario final, que suele optar por servicios de terceros.

A continuación se expondrán las principales soluciones en este campo, dividiéndose las mismas en

privadas (o tradicionales) o públicas (o de mantenimiento propio), ofreciendo las primeras una mayor facilidad de uso frente a la seguridad y control de las segundas.

4.1.1 Principales soluciones privadas

En esta sección se describirán las principales soluciones de almacenamiento privadas, en las que el usuario utiliza un cliente y aloja el contenido de sus archivos en un proveedor externo.

- **Dropbox**

  Dropbox fue uno de los primeros (y más famosos) servicios para almacenamiento online. Introducido en 2007 de la mano de una startup del mismo nombre, cuenta con unos 200 millones de usuarios.

  Técnicamente hablando se centra en la sincronización de archivo y la compartición de los mismos con otros usuarios, destacando por la facilidad de uso de su aplicación cliente. Dropbox fue pionero en integrar el cliente en el gestor de archivos del sistema operativo del usuario, de forma que para el mismo gestionar y sincronizar los archivos con el servidor sea tan simple como trabajar con una carpeta de su sistema.

- **Google Drive**

  Google Drive es el servicio de almacenamiento de archivos de Google (tal y como su propio nombre indica).

  Originalmente Google Drive era conocido como Google Docs, y servía como suite ofimática web para crear todo tipo de documentos (de texto, hojas de cálculo, etc). En abril de 2012, Google modifica el nombre de Google Docs y oficialmente lanza “Google Drive” [Google, 2012] dándole a la plataforma la capacidad de almacenar y compartir archivos con otros usuarios. Además del cliente web, Google ha desarrollado clientes para Mac OS X, Windows, Android e IOs, con los que permite sincronización de archivos entre el cliente y el servidor.

- **iCloud**

  iCloud es la plataforma de almacenamiento de archivos de Apple.

  Nacida en 2011[Apple, 2011] , está fuertemente orientada a la sincronización de archivos

entre los distintos dispositivos de Apple, creando backups automáticos tanto como de todo tipo de archivos del usuario, como de los contactos del mismo.

iCloud está diseñado para funcionar (e incluido por defecto) únicamente dispositivos de Apple que usen iOS5 o superior o OSX 10.7.2 "Lion" o superior. [Apple, 2015]

En Julio del 2013, iCloud contaba con 320 millones de usuarios [Kahn, 2013]

- **OneDrive**

OneDrive es el servicio de almacenamiento de archivos de Microsoft, lanzado en 2007 bajo el nombre "SkyDrive", pasando a ser conocido como "One Drive" en Febrero de 2014. El servicio pretende, al uso de Google Drive, integrar funciones de offimática online a la vez que sirve para almacenar y sincronizar archivos entre varios dispositivos.

El cliente de OneDrive está disponible para todas las plataformas Microsoft, incluyendo las plataformas de juego como Xbox360 o XboxOne o las plataformas móviles (Windows Phone). Existen clientes oficiales para plataformas y software de la competencia, como para iOS, Mac OS X o Android.

- **Spider Oak**

SpiderOak es otro servicio de almacenamiento online de archivos cuya diferencia con respecto a otros servicios similares es su enfoque por la seguridad: El servicio permite al usuario encriptar los datos antes de subirlos al servidor usando su propia clave personal, de forma que para el servidor es imposible saber exactamente el contenido de los archivos que está guardando. Asimismo, una vez subidos los archivos al servidor, Spider Oak los vuelve a encriptar usando sus propias claves. Todo este proceso ha sido bautizado por el servicio como "Zero Knowledge" [SpiderOak, 2015]

Al igual que sus competidores, SpiderOak ofrece las características comunes a este tipo de servicios, como sincronización y compartición de archivos, historial de versiones, etc.

- **Otros**

Existen multitud de alternativas y soluciones que ofrecen almacenamiento online, de las cuales debemos citar dos más, no por sus innovadoras características o porque tecnológicamente

supongan un avance enorme con respecto a sus competidores, si no por su cuota de mercado, similar (o en algunos casos superior) a las alternativas aquí presentadas.

Debemos hacer mención pues a **Box.net y Copy.** La primera ha tenido una acogida bastante grande entre el público, entre otras cuestiones por ofrecer gratuitamente desde 10 a 50GB (si usa la aplicación móvil).Asismimo, Box.net está fuertemente orientada a ofrecer servicios de almacenamiento online a nivel corporativo, teniendo en octubre de 2012 en 92% de las empresas pertenecientes al top Fortune 500 como cliente [Loeb, 2012]

La segunda, Copy, ha ganado popularidad por ofrecer gratuitamente 15GB de entrada y 5GB más a través de invitaciones a otros usuarios (sin límite aparente).

Ambas ofrecen el mismo tipo de servicio que otras alternativas, tales como sincronización, almacenamiento y compartición de archivos, encriptación de los mismos en el lado servidor, etc.

4.1.2 Principales soluciones públicas o de almacenamiento propio

En esta sección se describirán las soluciones más populares que ofrecen al usuario la capacidad de gestionar sus propios datos, permitiendole instalar y configurar el servidor que se encargará de alojar sus archivos.

- **ownCloud**

  ownCloud fue lanzado en Enero de 2010 por un desarrollador de el escritorio de Linux KDE.

  Fue el primer servicio de alojamiento en la nube que permitía al usuario instalar su propio servidor, siendo actualmente el referente en este campo.

  OwnCloud está escrito en PHP y Javascript, ofreciendo un cliente multiplataforma (disponible para Mac OS X, Linux, Windows y plataformas móviles) así como sincronización y compartición de archivos, encriptación o historial de versionas.

  Finalmente, cabe destacar que ownCloud está licenciado bajo la AGPL3 siendo el referente en software libre en lo que almacenamiento online se refiere.

- **Pydio**

  Pydio, [Pydio, 2013] nacido en 2013, es otra alternativa libre a las soluciones de almacenamiento privadas, intentando dotar al usuario de mayor libertad y control. Su parte

servidor está desarrollada en PHP y Javascript, estando los distintos clientes en Python (versión de escritorio), Java u Objective C (Android y iOS).

Pydio busca diferenciarse ofreciendo un mayor grado de configuración y personalización con respecto a otros sistemas, pudiendo configurarse la apariencia del mismo o el comportamiento de la parte servidor. Ello permite por definir en que sistema se guardarán los archivos (sistema de ficheros local, Amazon S3, FTP..) o cómo distribuir la carga del servidor ante múltiples peticiones.

Este grado de personalización ha hecho que Pydio se centre en el entorno empresarial, ofreciendo varios planes de pago diseñados para entornos empresariales.

4.1.3 Tabla comparativa

Servicio	Sincro	Historial de versiones	Cliente multi plataforma	Encriptación servidor	Encriptación persona /lado cliente	Encriptación en el transporte
Dropbox	Sí	Sí	Sí	AES – 256	No	SSL
Google Drive	Sí	Sí	No	No	No	No
iCloud	Sí	Sí	No	AES – 128/256	No	SSL
SpiderOak	Sí	No	Sí	AES – 256	AES - 256	SSL
OneDrive	Sí	No (sólo documentos)	No	No	No	SSL
Box.net	Sí	Sí	Sí	AES – 256	No	SSL
Copy	Sí	Sí	Sí	AES – 256	No	SSL
OwnCloud	Sí	Sí	Sí	AES – 128	No	SSL
Pydio	Sí	Sí	Sí	Sí - ENCFS	No	SSL

Tabla 1: Tabla comparativa de servicios de almacenamiento online

Leyenda

Sincronización

Indica si el servicio sincroniza el estado de los archivos entre los distintos dispositivos del cliente y los archivos del servidor.

Historial de Versiones:

Indica si el servicio posee la capacidad de revisar versiones de archivos modificados o eliminados.

Cliente multiplataforma:

Entiéndase cliente multiplataforma en este contexto como la existencia de un cliente en Windows, Mac OS X y Linux (dejando plataformas móviles y otras al margen).

Encriptación en el servidor

Entiéndase como encriptación de los archivos una vez estos han sido almacenados en el servidor.

Encriptación personal:

Encriptación de los archivos antes de ser transferidos o almacenados en el servidor, en otras palabras, encriptación en el lado del cliente.

Encriptación en el transporte:
Indica si la transmisión de los archivos desde la máquina cliente al servidor se realiza de forma cifrada (usando un protocolo seguro como HTTPS).

5 . Metodología y Planificación

En este capitulo se expone la planificación a la hora de elaborar este Proyecto Final de Carrera así como la metodología aplicada en la implementación y desarrollo del mismo.

5.1 Metodología

La metodología de software utilizada para este proyecto es la denominada Metodología en Espiral [Boem, 1986], que consta de cuatro fases o iteraciones:

1. Análisis de Requisitos
2. Diseño del software
3. Implementación del prototipo
4. Pruebas

Siguiendo los principios de desarrollo ágil [Beck, 2001], se procura que la duración entre la primera y última fase sea lo más corta posible, obteniendo un feedback lo más temprano posible para mejor continua y gradualmente el software implementado y haciendo especial hincapié en la fase 3 y 4.

Esta metodología se ha aplicado en los tres desarrollos de software de este proyecto: El cliente web, el cliente de escritorio y el servidor.

5.2 Planificación

La planificación del proyecto se ha dividido en tres partes, correspondiendo cada parte a la parte del proyecto realizada: El servidor web, el cliente web y el cliente de escritorio. Dentro de cada parte, y siguiendo la metodología en espiral adoptada, se ha dividido las tareas a realizar en cuatro fases:

1. **Análisis**: Dónde se hace una análisis del problema que esta parte del proyecto, intenta abordar, recogiendo la documentación necesaria y estudiando las herramientas que se usarán, así como definiendo los requisitos de
2. **Diseño**: A partir de la información recogida en la fase anterior, se diseña el software a implementar.
3. **Implementación**: Donde, siguiendo el patrón establecido en función de la pieza de software que se esté desarrollando y el diseño realizado, se implementa la solución escogida.

4. **Pruebas**: En la última fase se probarán exhaustivamente el software implementado, realizando tests unitarios y de integración.

Nótese que si la fase 4 no resulta satisfactoria, se volverá a la fase 2 y 3 ajustando el diseño e implementación para intentar pasar las pruebas realizadas. Este proceso se repetirá hasta que la solución creada pase las pruebas y se ajuste correctamente a los requisitos definidos.

El tiempo total empleado en la realización de cada parte del proyecto (excluyendo la realización de esta memoria) ha sido el siguiente:

Parte	Horas
Servidor	411
Cliente web	280
Cliente de escritorio	205
Total	896

Tabla 2: Resumen planificación de tareas

En las siguientes secciones se desglosarán las tareas y horas empleadas para cada parte del proyecto en función de la parte del mismo realizada.

5.2.1 Planificación servidor

Fase/Tarea	Horas
Análisis	
Adquisición de conocimientos protocolos HTTP	10
Documentación servidores Web en Python	15
Documentación Node.js y API REST	20
NGINX y arquitecturas de red asíncronas	30
Documentación bases de datos NoSQL	20
Documentación representación de estructuras de datos MongoDB	20
Total	115
Diseño	
Diseño prototipo Python	10
Diseño aplicación Node.JS	30
Diseño configuración Nginx	15

Diseño estructura de base de datos	40
Diseño esquema de autenticación	20
Total	115
Implementación	
Implementación prototipo Python	30
Configuración servidor Nginx	20
Implementación aplicación Node.js	25
Implementación estructura base de datos	20
Implementación autenticación	15
Despliegue en instancia de Amazon EC2	6
Total	116
Pruebas	
Pruebas prototipo Python	20
Pruebas base de datos	15
Pruebas aplicación Node.js	20
Pruebas rendimiento servidor	10
Total	65

Tabla 3: Planificación servidor

Nótese como en este fase se comenzó realizando un prototipo en Python, cuyas pruebas iniciales arrojaron un rendimiento muy pobre (ver sección *9. Pruebas*). A partir del desarrollo de este prototipo, se decidió realizar el proyecto en base a una arquitectura de red asíncrona y usar el patrón de diseño reactor, explicado en la sección *6.Recursos*

5.2.2 Planificación cliente web

Fase/Tarea	Horas
Análisis	
Análisis casos de uso	10
Documentación AngularJS	25
Documentación API REST	20
Documentación Bootstrap	10
Documentación Material Design	5
Total	65
Diseño	
Diseño interfaz y árbol de directorios	25
Diseño llamadas HTTP al servidor (API)	15
Total	40
Implementación	
Implementación interfaz y árbol de directorios	35
Implementación llamadas HTTP al servidor (API)	40
Implementación casos de uso	60
Total	135
Pruebas	
Pruebas de llamadas HTTP con el servidor (API)	40
Total	40

Tabla 4: Planificación cliente web

5.2.3 Planificación cliente escritorio

Fase/Tarea	Horas
Análisis	
Análisis casos de uso	5
Documentación algoritmos de sincronización	25
Documentación Electron	10
Total	40
Diseño	

Diseño interfaz cliente	5
Diseño llamadas HTTP (API)	10
Diseño algoritmo de sincronización	25
Total	40
Implementación	
Implementación interfaz	15
Implementación llamadas HTTP al servidor (API)	10
Implementación casos de uso	10
Implementación algoritmo de sincronización	30
Distribución y empaquetamiento cliente	15
Total	70
Pruebas	
Pruebas de integración con servidor	20
Pruebas de sincronización	30
Total	50

Tabla 5: Planificación cliente de escritorio

6 . Recursos utilizados

En esta sección se explicarán los recursos utilizados para la elaboración y desarrollo del proyecto.

6.1 Recursos hardware

Ordenadores portátiles

Para el desarrollo e implementación de tanto el cliente como el servidor, se han utilizado los siguientes ordenadores portátiles:

- MacBook Pro (13 pulgadas), con las siguientes características:
  - Procesador: 2,9 GHz Intel Core i7
  - Memoria:8 GB 1600 MHz DDR3
  - Intel HD Graphics 4000 1024 MB
  - Mac OS X Yosemite
- Asus K53SV, con las siguientes características:
  - Procesador: 2,9 GHz Intel Core™ i7 2670QM
  - Memoria: 4 GB 1333 MHz DDR3
  - NVIDIA GeForce GT 540M con 1GB
  - Arranque dual: Ubuntu 14.04 LTS y Windows 7

Nótese que hemos hecho uso de dos portátiles, uno de ellos con dos sistemas operativos, para asegurar uno de los objetivos del proyecto: Que la aplicación de escritorio funcione correctamente en los tres sistemas operativos mayoritarios: Windows, Mac OS X y GNU/Linux.

Instancia de Amazon EC2

De cara a probar en un entorno real de producción el servidor y la aplicación web desarrollada, se ha hecho uso de una instancia o máquina virtual de Amazon EC2 [Amazon, 2015] con las siguientes características:

- Modelo t2.micro
- 1 CPU virtual Intel Xeon a 2,5 Ghz
- 1 GB Memoria
- Ubuntu 14.04 LTS
- 10 GB Almacenamiento EBS
- Dirección IP pública: http://dandelion.redes.dis.ulpgc.es

6.2 Recursos software

Los recursos software utilizados para la implementación de este proyecto han sido:

Lenguaje Javascript - ECMAScript5

Javascript es el lenguaje principal que se ha elegido a la hora de desarrollar el proyecto, tanto para la parte servidor como para la parte cliente (ya sea web o de escritorio). Ello es así debido a que el uso de este lenguaje permite hacer uso de una serie de tecnologías web ampliamente usadas y establecidas como pueden ser Node.js o AngularJS . Asimismo, el usar Javascript nos permite utilizar el mismo lenguaje de programación tanto en la parte servidor (con Node.js) como en la parte cliente (AngularJS, Electron), lo cual redunda en un desarrollo más ágil.

La especificación del lenguaje usada ha sido ECMAScript5 [Ecma, 2011].

Servidor web Nginx

Nginx [Nginx, 2015] es un servidor web caracterizado por tener una arquitectura destinada a eventos. Fue diseñado específica Mientras que los servidores web tradicionales abren un proceso o hilo nuevo por cada petición y esperan ante peticiones costosas en tiempo, Nginx no espera por este tipo de peticiones y pasa a la siguiente hasta que éstas finalicen, momento en el cual vuelve a consultar el estado de las peticiones bloqueantes. Este patrón de diseño es conocido como *"Patrón reactor"* [Schmidt, 2000]

Más detalladamente, Nginx posee una serie de procesos denominados "*workers"* que se conectan a un socket común para recibir conexiones HTTP/S. Cada *worker* ejecuta un bucle en el que espera a que le llegue una petición (un *evento*). Cuando ello ocurre, el *worker* trata la petición de forma asíncrona, de forma que si la misma ha de acceder a E/S o a una base de datos (operaciones por las que habría que

esperar a su finalización), el *worker* pasa a otra petición y en la siguiente iteración del bucle consulta si la petición bloqueante ha finalizado, y de ser así, emite la respuest. [Alexeev, 2008]

Cabe destacar que el número de procesos worker es fijo una vez que se arranca Nginx, por lo que, al no abrirse nuevos procesos, el consumo de memoria es bastante comedido.

Nginx también suele usarse, entre otros, como balanceador de carga entre varias aplicaciones web o proxy reverso (un proxy entre un grupo de servidores y el cliente final)

Node.js

Node.js es una plataforma para construir aplicaciones de red usando el lenguaje Javascript. Posee tanto un entorno de ejecución (construido sobre el motor V8 de Google [Google, 2015]) que se encarga de compilar y ejecutar el código, cómo una API con librerías y funciones para interactuar con el mismo [Node.js, 2015].

Al igual que Nginx, la arquitectura de Node.js destaca por ser asíncrona y no bloqueante, siguiendo el patrón de diseño reactor [Schmidt, 2000]. Ello lo convierte en ideal para aplicaciones de red que hagan un uso intenso de operaciones de entrada salida, como pueden ser el acceso a ficheros o las consultas a una base de datos, adaptándose por tanto al caso de uso del presente proyecto.

Node.js ejecuta un bucle en un sólo proceso en el que va ejecutando código secuencialmente. Dicho código puede utilizar en algunos segmentos técnicas como *callbacks* o *promesas [Nathan, 2011]* que indicarán a Node.js que no debe esperar por esos segmentos, y que consulte a los mismos en la siguiente iteración del bucle.

NPM

NPM [Node, 2015] es el gestor de paquetes de Node. Permite instalar, actualizar y eliminar librerías y paquetes de la plataforma Node.js desde la línea de comandos, al uso de otros gestores de paquetes más generalistas como APT o Homebrew.

NPM se ha usado para instalar las librerías y dependencias necesarias para el correcto funcionamiento de la parte servidor desarrollada en Node.js. La lista completa de librerías se puede encontrar en el *Anexo 13.1 Lista de dependencias del servidor*

Express.js

Express.js [Strongloop, 2015] es un framework web para crear servidores basado en la plataforma

Node.js. Permite asociar una serie de llamadas HTTP que se realizan al servidor a una serie de acciones, definiendo una API para acceder al mismo.

En el proyecto, Express.js se usa para crear una API para realizar operaciones con el servidor web implementado mediante Node y Nginx.

Un ejemplo de un servidor web usando Express.js es el siguiente:

```
var express = require('express');
var app = express();

app.get('/', function (req, res){
  res.send('Hello World!');
});
// accept PUT request at /user
app.put('/user', createUser);

var server = app.listen(3000, function (){
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

Figura 1: Ejemplo servidor web con dos rutas definidas

JSON Web Tokens (JWT)

JSON Web Tokens o JWT [Auth0, 2015] es un estándar de la IETF [Jones, 2015] usado para la generación y uso de tokens de autenticación. Es ideal para su uso en APIs que se usarán desde varios dispositivos y que necesitan de un método seguro y sencillo para autenticar peticiones HTTP.

El estándar JWT permite establacer tokens formados por:

- Header: En los que se indica el algoritmo de cifrado y tipo de token usado. Por ejemplo:

```
{ "alg": "HS256",
  "typ": "JWT"}
```

- Payload: Datos principales usados en la generación del token (generalmente las credenciales del usuario). Por ejemplo:

```
{ "sub": "1234567890",
  "name": "John Doe",
  "admin": true}
```

- Firma de verificación: Usada para verificar que header y payload son correctos. Requiere de una clave secreta que debe ser proporcionada por el servidor, siendo normalmente de la siguiente forma:

```
HMACSHA256( base64UrlEncode(header) + "." +  base64UrlEncode(payload),secret)secret base64 encoded
```

Utilizando la firma de verificación a partir de la clave secreta proporcionada por el servidor, se establace la validez del token, que será de la forma:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

En este proyecto, JWT ha sido usado para crear tokens de autenticación a partir de las credenciales iniciales del usuario, permitiendo así establecer un mecanismo de autenticación válido para cualquier tipo de entorno (web, de escritorio, etc).

MongoDB

MongoDB [MongoDB, 2015] es una base de datos NoSQL, es decir, una base de datos que no utiliza un modelo relacional (como pudiera ser todas aquellas bases basadas en SQL), no estableciendo relaciones explícitas entre los datos que almacena.

MongoDB utiliza el concepto de colecciones, semejantes a las tablas SQL y el concepto de documentos, que se asemejarían a las filas. Una colección se compone de varios documentos, no teniendo estos por qué tener una estructura prefijada (pudiendo varios documentos tener campos distintos en la misma colección) y no albergan claves primarias o secundarias (como sería el caso de una fila SQL). Estas características proporcionan a la base de datos una gran flexibilidad y rapidez, razones por las que se ha elegido MongoDB para este proyecto.

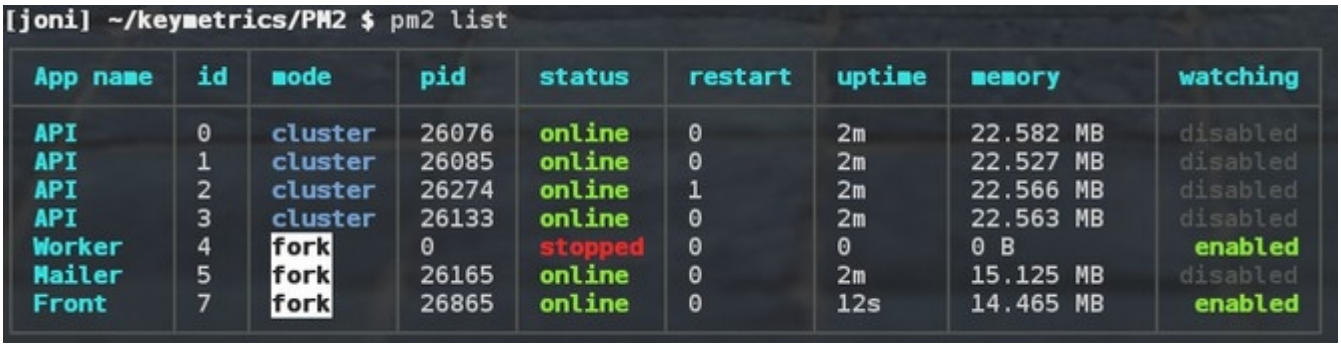
Más concretamente, MongoDB es utilizado en el proyecto para gestionar los recursos y los usuarios, así como los permisos que estós últimos tienen sobre los primeros.

PM2

PM2 [Unitech, 2015] es un gestor de procesos en entornos de producción. Permite mantener uno o varios procesos activos indefinidamente, reiniciándolos en caso de error de la aplicación y manteniendo en todo momento un log de actividad para cada proceso.

PM2 es usado en el proyecto para gestionar la actividad de los procesos Node.js en la instancia de

Amazon EC2, como se explica en la sección *8.1.17.*



```
[joni] ~/keymetrics/PM2 $ pm2 list
```

App name	id	mode	pid	status	restart	uptime	memory	watching
API	0	cluster	26076	online	0	2m	22.582 MB	disabled
API	1	cluster	26085	online	0	2m	22.527 MB	disabled
API	2	cluster	26274	online	1	2m	22.566 MB	disabled
API	3	cluster	26133	online	0	2m	22.563 MB	disabled
Worker	4	fork	0	stopped	0	0	0 B	enabled
Mailer	5	fork	26165	online	0	2m	15.125 MB	disabled
Front	7	fork	26865	online	0	12s	14.465 MB	enabled

Figura 2: Ejemplo gestor de procesos PM2

AngularJS Framework

AngularJS [Google, 2010] es un framework de interacción web desarrollado por Google y basado en los patrones MVC (Modelo – Vista – Controlador) [Glenn, 1988] y MVVM (Modelo – Vista – VistaModelo) [Gossman, 2005], ofreciendo por tanto:

- **Vistas**: Renderizadas por el navegador del cliente
- **Controladores**: Definen la lógica presente en las vistas en base a los modelos.
- **Modelos**: Representan los datos de la aplicación cliente. A diferencia de otros frameworks, en Angular no es necesario especificarlos explícitamente, siendo estos muchas veces inferidos en función de las vistas o delegando su representación en agentes externos (base de datos, archivos JSON..)

Asimismo, Angular se caracteriza entre otros por:

- Separar el diseño de las vistas de la lógica de negocio de las mismas (uno de los pilares del MVVM)
- Sincronizar automáticamente los datos introducidos en las vistas con el modelo, y viceversa (“two-way data-binding” en el argot de Angular)
- El uso de “directivas”, que permiten especificar comportamiento a elementos HTML o definir reglas para la generación de dichos elementos. Por ejemplo:
  - Repite un determinado código HTML al renderizar la web dependiendo de los datos de un determinado modelo (*ng-repeat* [Google, 2010])

- Carga una clase CSS distinta en función del estado de una variable en el controlador (*ng-class,* [Google, 2010])
- Muestra una parte de la web sólo si el modelo de la aplicación está en un estado determinado (directiva *ng-show,* [Google, 2010])
- Etc.

En la siguiente imagen se puede apreciar el uso de una directiva que hará que el navegador renderice una lista (<*li*>) con propiedades de cada coche para cada coche de un hipotético conjunto de los mismos.

```
<li ng-repeat="car in cars">
    <ul> {{car.model}}</ul>
    <ul> {{car.color}}</ul>
    <ul> {{car.owner}}</ul>
</li>
```

Figura 3: Ejemplo directiva AngularJS

Finalmente, AngularJS se usa escribiendo únicamente en HTML y Javascript, lo cual permite que la implementación del proyecto en todas sus fases se pueda realizar bajo un mismo lenguaje, reutilizando código entre el cliente y servidor y favoreciendo una mayor cohesión entre todas las partes del mismo.

Bower

Bower [Bower, 2015] es el gestor de paquetes para clientes web. Se ha usado en el proyecto para manejar la gestión de librerías y dependencias necesarias para configurar el proyecto web.

La completa lista de dependencias del cliente web se encuentre en*Anexo 13.2 Lista de dependencias del cliente*.

Bootstrap Framework

Bootstrap [Twitter, 2011] es un framework libre para la presentación y estilo de páginas web usando CSS y HTML. Aporta una serie de elementos visuales y reglas que facilitan el maquetado y diseño de páginas web *"responsive"* (ajustando el contenido al tamaño de la ventana). Asimismo ofrece varias plantillas predefinidas que, apoyándose en una serie de reglas y componentes, nos permiten rápidamente aplicar varios estilos visuales a la web.

Bootstrap ha sido utilizado para desarrollar la apariencia del cliente web, así como para permitir que el

mismo se vea correctamente en dispositivos con pantallas de diferente tamaño. En concreto, se ha utilizado la plantilla Paper[Park, 2015], que ofrece un aspecto limpio e intuitivo, basado en los principios de diseño web Material Design, ideados por Google [Google, 2014].

Electron Framework

Electron [Github, 2014] es un framework libre para desarrollar aplicaciones de escritorio utilizando tecnologías web, permitiendo ejecutar una página web como si fuera una aplicación de escritorio nativa. El código escrito en Electron es ejecutado en Node.js, lo que facilita distribuir una app con el mismo código en varios entornos de escritorio. Por ello, este framework es idóneo para el caso de uso de este proyecto, cuya aplicación de escritorio tiene el objetivo de ser usada en las tres plataformas principales: Mac OS X, Windows y GNU/Linux

Git

Git [Torvalds, 2009] es un software de control de versiones, similar a otros como CVS o Subversion. Al igual que éstos, nos permite gestionar los cambios de un proyecto software, revisándolos o deshaciéndolos en cualquier momento.

Mientras que en otros sistemas de control de versiones los cambios en el software son subidos a un repositorio central, Git utiliza un acercamiento descentralizado, pudiendo cada desarrollador del proyecto realizar cambios sobre su repositorio local o su repositorio remoto. Esto facilita enormemente el desarrollo en proyectos grandes, con un gran número de desarrolladores o en los que no es recomendable probar nuevas características sobre un mismo repositorio.

Asimismo, Git nos permite:

- Documentar cambios.
- Deshacer en cualquier momento cambios realizados sobre un archivo.
- Ver las diferencias entre varias versiones del mismo archivo.
- Crear *ramas* de desarrollo, probando nuevas características sin modificar el código original.

En este proyecto Git se ha usado junto con el servicio web Github [Github, 2008], que nos permite realizar las operaciones anteriormente mencionadas fácilmente.

Podemos ver un ejemplo de la historia de cambios o commits (en el argot de git) en la siguiente imagen:

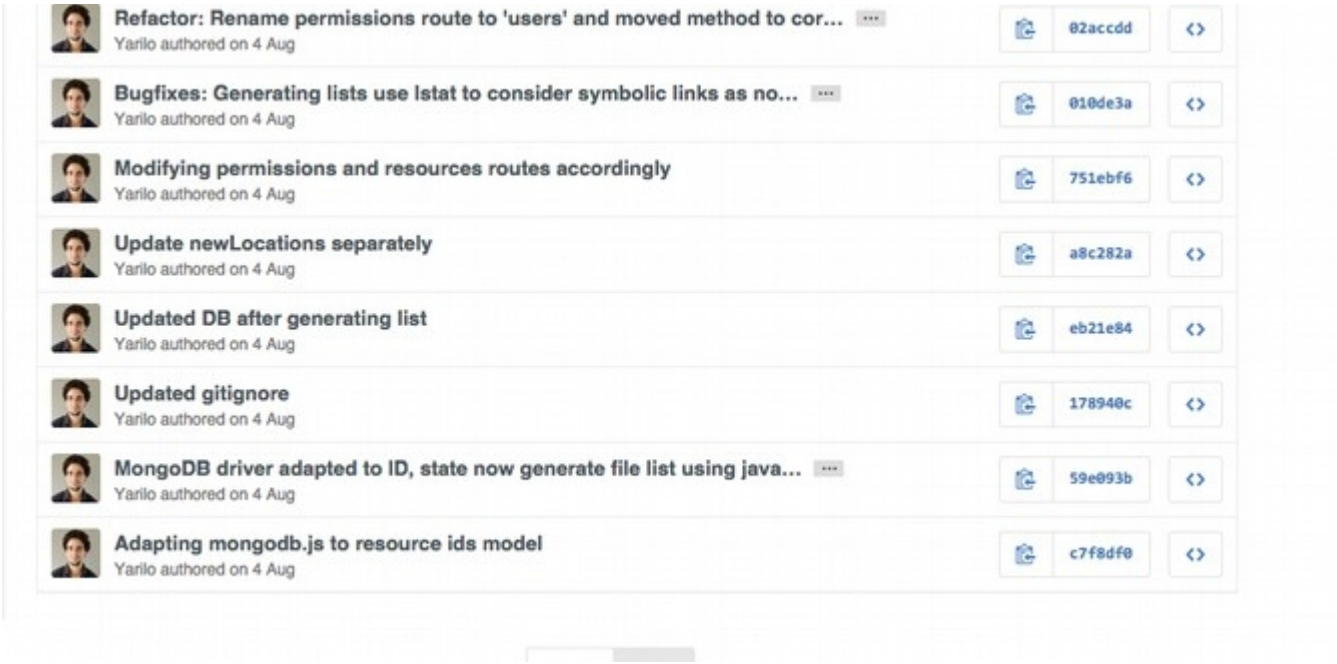


Figura 4: Historia de commits repositorio Git alojado en Github

En este proyecto, se han creado tres repositorios remotos en Github con la ayuda de Git: Uno para la aplicación web (formada por el cliente web y el servidor), otro para el cliente de escritorio y un último repositorio para poder seguir la realización de esta memoria y los cambios sobre la misma.

Mailgun

Para enviar mails para restaurar la contraseña o validar la cuenta de un usuario se necesita un servidor de correo SMTP o bien un servicio externo dedicado a ello. Para este proyecto hemos elegido Mailgun [Rackspace, 2015] por su facilidad de uso y las estadísticas que ofrece, así como por ofrecer más fiabilidad que otros servicios no dedicados como pudiera ser Gmail u Outlook, donde los correos pueden no ser enviados al confundirse con spam.

Figura 5: Ejemplo estadísticas Mailgun

Editor Atom

Atom [Github, 2015], es un editor de texto desarrollado por los creadores de Github. Esta creado usando tecnologías web (y en concreto utiliza el Framework Electron, mencionado anteriormente), y ofrece, entre otros:

- Coloreado de sintaxis.
- Integración con Git y Github.
- Detección de errores de sintaxis comunes.
- Personalización y configuración: El editor es libre y ha sido creado poniendo énfasis en la personalización, existiendo por tanto muchos paquetes y extensiones que ofrecen nuevas características o modifican el comportamiento y apariencia del mismo.

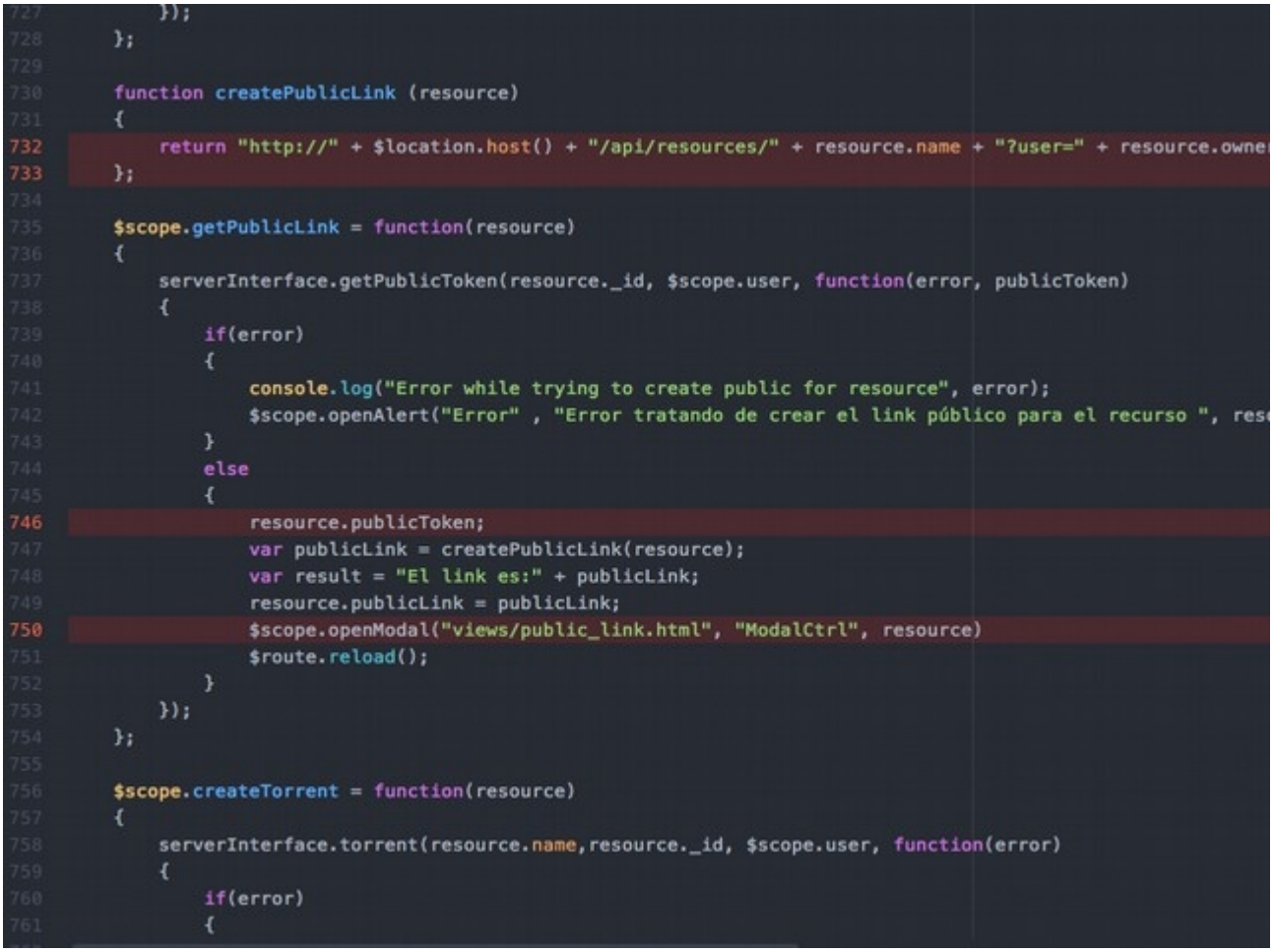


Figura 6: Ejemplo edición de código con el editor de texto Atom

Draw.io

Draw.io [Jgraph, 2015] es un software de modelado en línea que permite crear diagramas UML desde el navegador sin necesidad de descargar ningún tipo de software. Draw.io ha sido utilizado a la hora de desarrollar los diagramas UML descritos en este proyecto en la sección *7.3 Casos de uso.*

Editor de textos LibreOffice

Para la realización de esta memoria se ha optado por el uso del editor de textos LibreOffice [The Document Foundation, 2015], un clon libre del popular software de Microsoft, Microsoft Word.

Microsoft Power Point

Para realizar los esquemas y diagramas (a excepción de los UML) realizados en la memoria se ha utilizado el popular software de presentaciones de Microsoft, Microsft Power Point, que ofrece varias

herramientas para crear este tipo de esquemas.

7 . Análisis

En esta sección se expondrán los requisitos necesarios para implementar el proyecto así como los actores y casos de uso envueltos en el mismo.

7.1 Requisitos

A continuación se presentarán los requisitos necesarios para este proyecto, dividiéndose los mismos en: Requisitos Funcionales, Requisitos de Seguridad y Requisitos de Disponibilidad.

7.1.1 Requisitos Funcionales

Servidor

Número requisito	Descripción
RF1	Ofrecer archivos bajo protocolo HTTP
RF2	Sincronización con repositorio local
RF3	Compartición de archivos y carpetas
RF4	Generación de ficheros P2P en base a recursos almacenados
RF5	Descarga de archivos parcial
RF6	Subida de archivos parcial

Tabla 6: Requisitos funcionales del servidor

Cliente web

Número Requisito	Descripción
RF1	Soporte protocolo HTTP
RF2	Sincronización con repositorio remoto
RF3	Representación web del sistema de ficheros del usuario

Tabla 7: Requisitos funcionales del cliente web

Cliente escritorio

Número Requisito	Descripción
RF1	Soporte protocolo HTTP
RF2	Sincronización con repositorio remoto y gestión de carpetas compartidas
RF3	Gestión de conflictos entre versiones del mismo archivo.

Tabla 8: Requisitos funcionales del cliente escritorio

7.1.2 Requisitos de seguridad

Servidor

Número Requisito	Descripción
RS1	Gestión de usuarios y permisos
RS2	Encriptación bajo clave conocida sólo por usuario
RS3	Conexiones seguras: HTTPS/SSL

Tabla 9: Requisitos de seguridad del servidor

Cliente web

Número Requisito	Descripción
RS1	Conexiones seguras: HTTPS/SSL

Tabla 10: Requisitos seguridad del cliente web

Cliente escritorio:

Número Requisito	Descripción
RS1	Conexiones seguras: HTTPS/SSL

Tabla 11: Requisitos seguridad del cliente de escritorio

7.1.3 Requisitos de disponibilidad

Servidor

Número Requisito	Descripción
RDISP1	Mac OS X, GNU/Linux

Tabla 12: Requisitos de disponibilidad del servidor

Cliente web

Número Requisito	Descripción
RDISP1	Chrome, Firefox, Internet Explorer 9 y superiores.

Tabla 13: Requisitos de disponibilidad del cliente web

Cliente escritorio

Número Requisito	Descripción
RDISP1	Multiplataforma: Mac OS X, GNU/Linux, Windows

Tabla 14: Requisitos de disponibilidad del cliente de escritorio

7.2 Identificación de actores

Podemos encontrar tres tipos de actores a lo largo de toda la solución implementada: *Usuarios no registrados*, *Usuarios registrados no autenticados* y *Usuarios Registrados* autenticados en la aplicación.

Nótese que son actores distintos y no una generalización en tanto que no comparten ciertos casos de uso (un usuario no registrado no puede iniciar sesión y un usuario registrado y no autenticado no puede registrarse, por ejemplo).

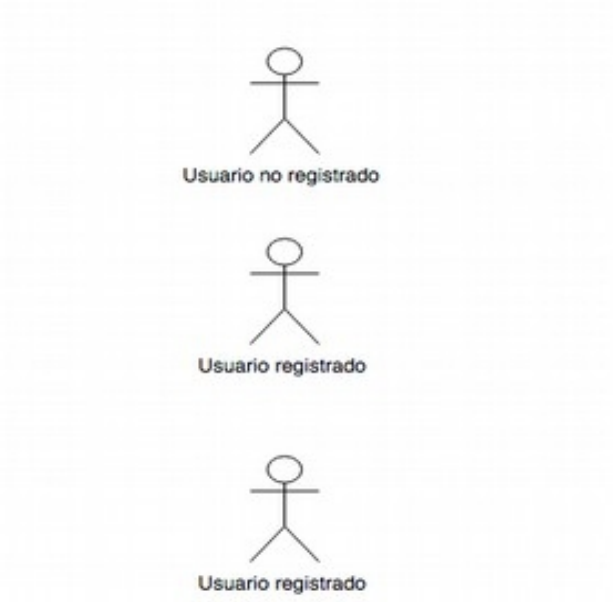


Figura 7: Actores de la solución implementada

- **Usuario no registrado:** Todo usuario que no se haya registrado en la aplicación vía el formulario web (ver sección *8.2 Cliente web*) tiene el acceso restringido a la misma. Las únicas acciones posibles de este tipo de usuario son la de ver archivos compartidos públicamente y registrarse.
- **Usuario registrado no autenticado:** Usuario con cuenta en la aplicación, pero sin haberse identificado en la misma mediante usuario y contraseña.
- **Usuario registrado autenticado:** Todo aquel usuario con cuenta en la aplicación. Puede iniciar sesión e interactuar con sus archivos en cualquier grado. Puede compartir archivos con ciertos

permisos y puede interactuar con sus archivos compartidos en la medida en que los permisos otorgados sobre esos archivos se lo permitan.

En la siguiente tabla se definen más concretamente los objetivos de estos actores:

Actor	Objetivos	Descripción
Usuario no registrado	Registrarse	Registrarse en la aplicación usando el formulario web.
	Ver archivos públicos	Ver archivos compartidos públicamente por usuarios registrados
Usuario registrado no autenticado	Iniciar sesión	Iniciar sesión en la aplicación ya sea mediante el cliente web o el cliente de escritorio, convirtiéndose en un usuario registrado autenticado.
	Recuperar credenciales	Si el usuario ha olvidado la contraseña de acceso, puede recuperarla haciendo uso de la aplicación web
Usuario registrado autenticado	Gestionar sus recursos	Realizar operaciones sobre los recursos de los que es dueño
	Compartir sus recursos	Compartir recursos de los que es dueño otorgando permisos a los mismos.
	Interactuar con recursos compartidos	Realizar operaciones sobre recursos compartidos hasta el nivel de permisos que posee sobre esos recursos.
	Cerrar sesión	Cerrar sesión, convirtiéndose en un usuario no autenticado.
	Cerrar cuenta	Cerrar cuenta, convirtiéndose en un usuario no registrado.

Tabla 15: Lista de actores y objetivos

7.3 Casos de uso

A continuación se describen los posibles casos de uso de cada actor, tanto del cliente web como del cliente de escritorio, usándose principalmente diagramas UML.

7.3.1 Casos de uso cliente web

Usuario no registrado

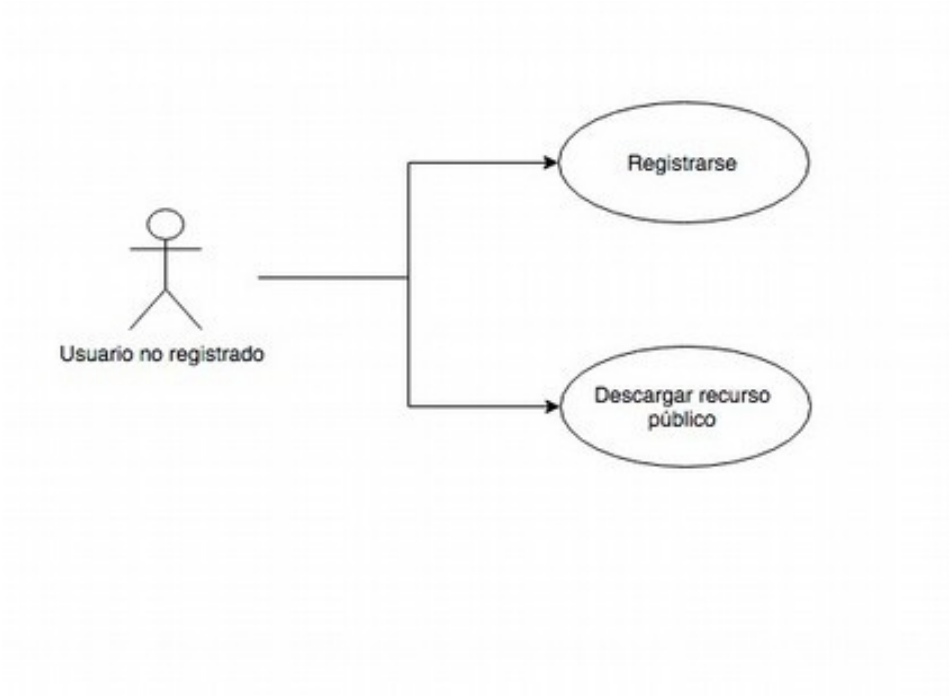


Figura 8: Casos de uso usuario no registrado – Cliente web

Usuario registrado no autenticado

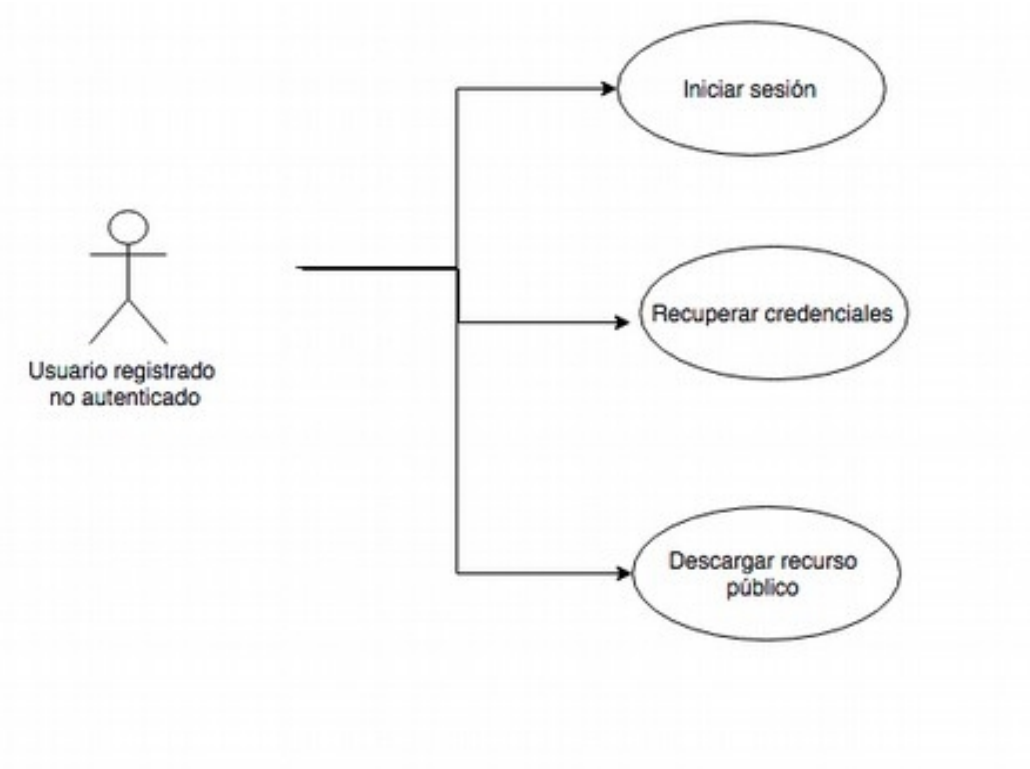


Figura 9: Casos de uso usuario registrado no autenticado – Cliente web

Usuario registrado autenticado

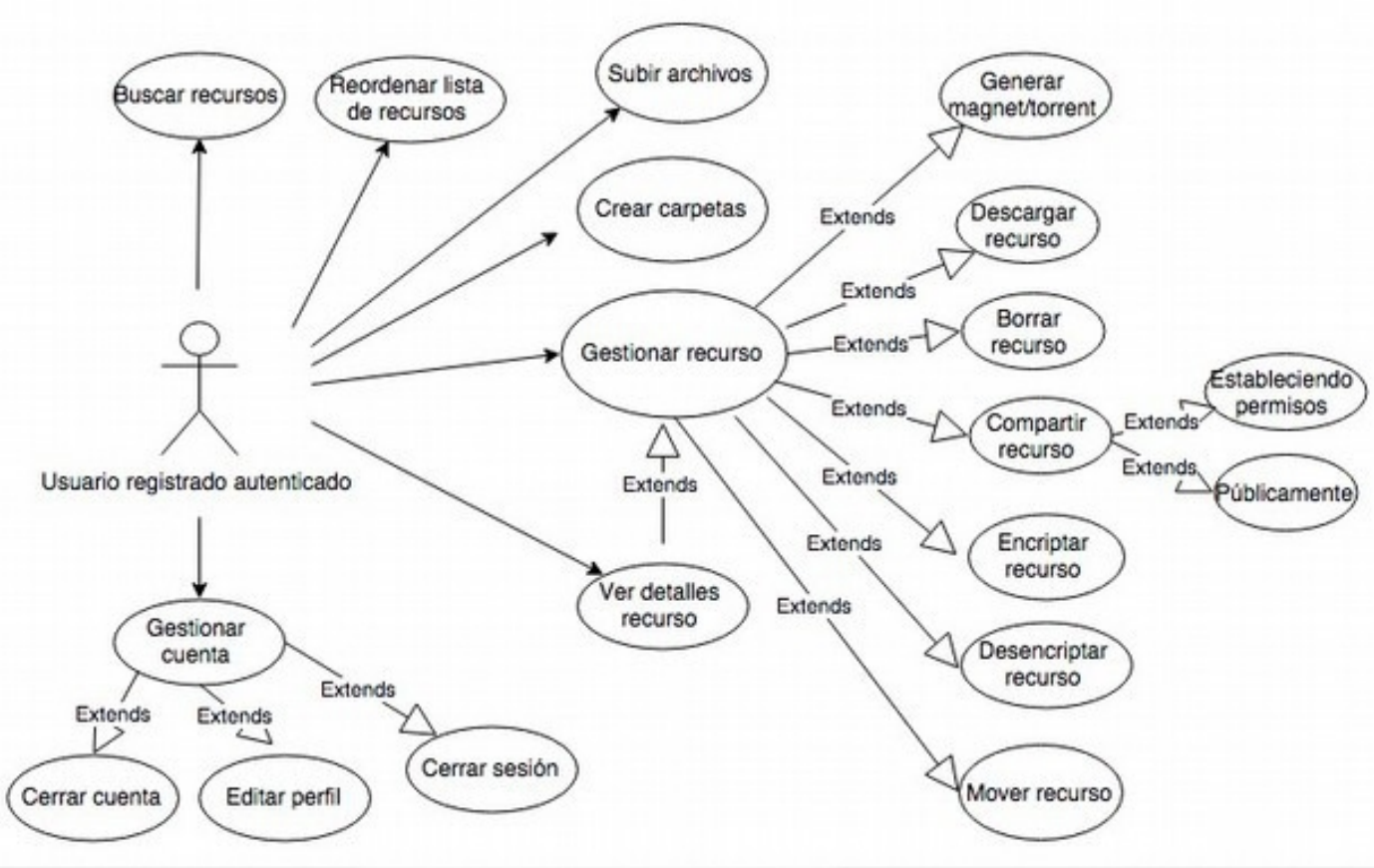


Figura 10: Casos de uso usuario registrado y autenticado – Cliente web

Nótese que todas las acciones asociadas a *Gestionar Recurso* sólo podrán realizarse si el usuario tiene los permisos necesarios sobre el recurso.

7.3.1 Casos de uso cliente escritorio

Usuario no registrado

Este actor no tiene ningún tipo de caso de uso asociado con la aplicación de escritorio, teniendo que realizar todas sus accioens en el cliente web.

Usuario registrado y no autenticado

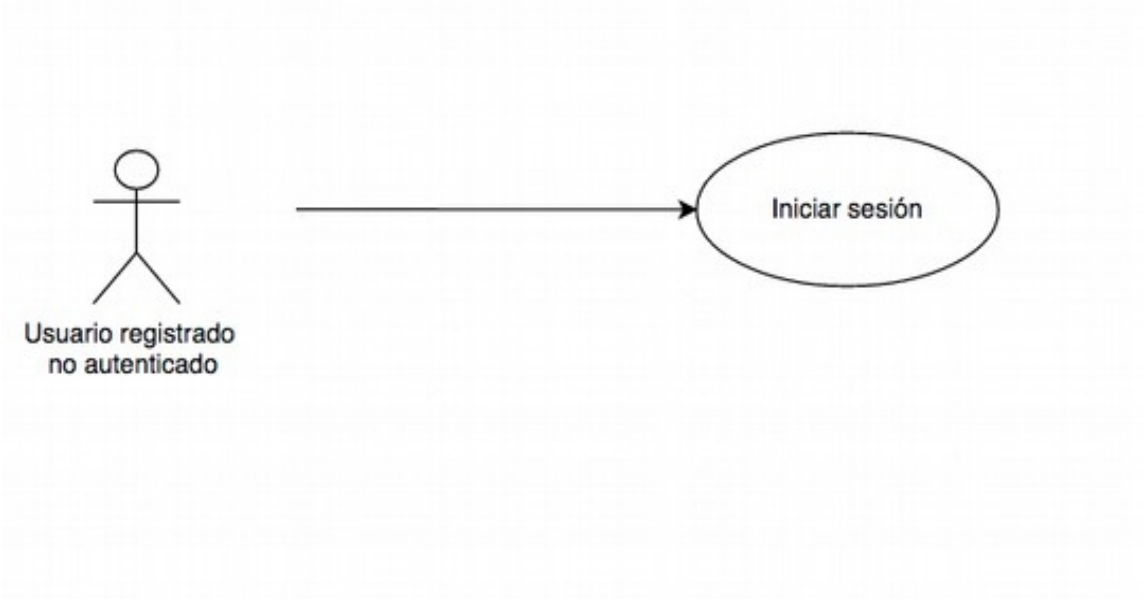


Figura 11: Casos de uso usuario registrado no autenticado - Cliente escritorio

Usuario registrado y autenticado

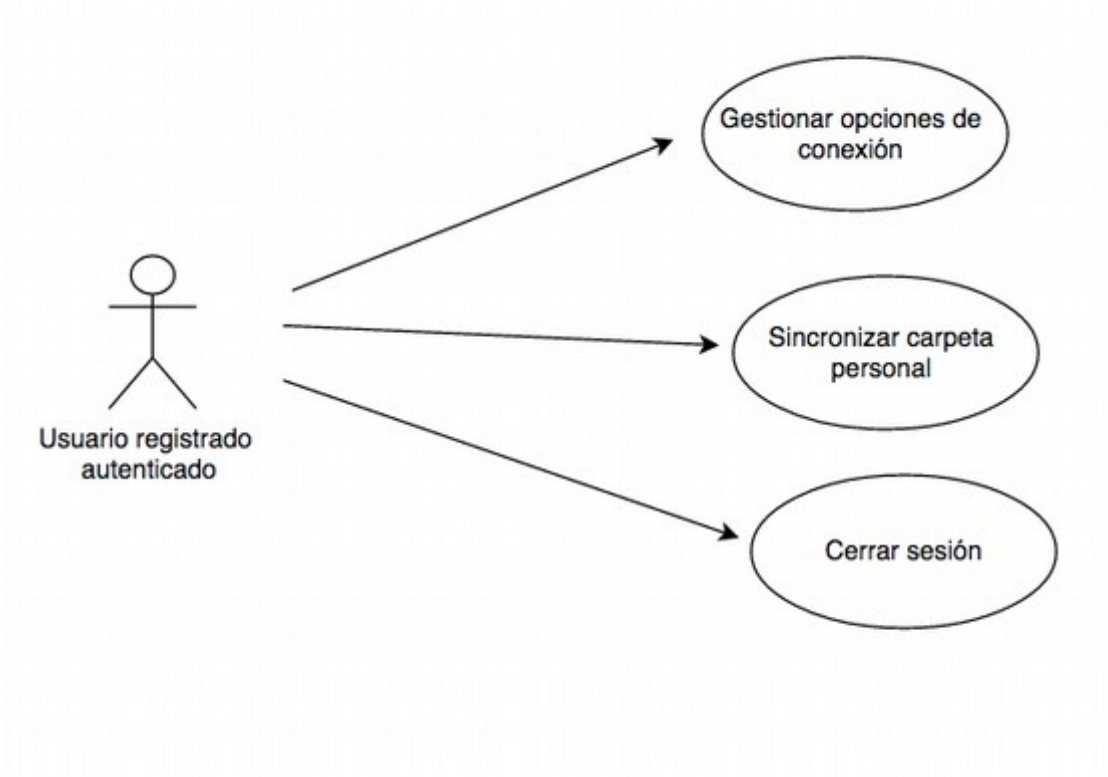


Figura 12: Casos de uso Usuario registrado y autenticado - Cliente escritorio

7.3.2 Listado de casos de uso

Cliente web

Actor	Caso de uso	Número caso de uso
Usuario no registrado	Registrarse	CW01
	Descargar recurso público	CW02
Usuario registrado no autenticado	Iniciar sesión	CW03
	Recuperar credenciales	CW04
	Descargar recurso público	CW02
Usuario registrado autenticado	Buscar recursos	CW05
	Reordenar lista de recursos	CW06
	Subir archivos	CW07
	Crear carpetas	CW08
	Gestionar recurso	CW09
	Ver detalles recurso	CW10
	Generar magnet/torrent	CW11
	Descargar recurso	CW12
	Compartir recurso	CW13
	Borrar recurso	CW14
	Encriptar recurso	CW15
	Desencriptar recurso	CW16
	Mover recurso	CW17
	Gestionar cuenta	CW18
	Cerrar cuenta	CW19
	Editar perfil	CW20
	Cerrar sesión	CW21

Tabla 16: Casos de uso cliente web

Cliente escritorio

Actor	Caso de uso	Número caso de uso
Usuario no registrado	-	-
Usuario registrado no autenticado	Iniciar sesión	CE01
	Sincronizar carpeta personal	CE02
	Gestionar opciones de conexión	CE03
	Cerrar sesión	CE04

Tabla 17: Casos de uso cliente escritorio

8 . Diseño

En este apartado se explicará en detalle la solución implementada, dividiéndose el mismo en

el diseño de la aplicación servidor y en el diseño de la aplicación cliente.

8.1 Servidor

En este apartado se describará el servidor de almacenamiento y su arquitectura (o *stack*) implementada.

8.1.1 Visión general

Para abordar la implementación del servidor se ha utilizado una arquitectura cuyos componentes principales son :

- Servidor web Nginx: Servidor web asíncrono que se encarga de manejar las peticiones HTTP.
- Aplicación Node.JS: Que consulta autenticación y permisos de usuarios con la base de datos.
- MongoDB: Base de datos NoSQL que aloja los datos y permisos de los usuarios.
- Patrón de diseño reactor [Schmidt, 2000]: Con el que construimos una arquitectura asíncrona que no se bloquea ante operaciones de entrada/salida.

El resultado es un servidor con las siguientes características:.

- **Arquitectura asíncrona**

Tanto NGINX como Node.js siguen una arquitectura asíncrona y no bloqueante basada en el patrón reactor [Schmidt, 2000]. A grandes rasgos, esta arquitectura, en lugar de esperar ante peticiones que puedan bloquear el sistema, como una consulta a base de datos o una petición sobre un archivo,, pasa a la siguiente consulta y una vez que la petición original bloqueante finaliza, vuelve a atenderla y genera su respuesta.

Esta arquitectura es ideal para aplicaciones web en la que los accesos a disco o a base de datos son numerosos, permitiendo aguantar una mayor carga simultánea de usuarios (concurrencia) sin que el rendimiento global se vea afectado.

- **Caching**

  El servidor web NGINX soporta caching de las peticiones mediante GET condicionales y Etags.

Si un cliente pregunta por un contenido que no ha cambiado desde la última vez que accedió a él, se le informa y no se le vuelve a enviar el fichero. Ello ahorra ancho de banda y permite al servidor centrarse en otras peticiones.

- **Compresión**

  Las respuestas HTTP que realiza NGINX se comprimen, ahorrando en ancho de banda y haciendo que la conexión del cliente sea más rápida.

- **Conexión segura**

  Mediante HTTPS, el servidor permite el uso de conexiones cifradas, pudiendo transferirse los datos de manera segura.

- **Autenticación de usuarios:**

  Guardando el usuario y contraseña encriptados en la base de datos (mediante el uso de Bcrypt [Provos, 1998]) que son consultados por la aplicación escrita en Node.js.

- **Gestión de permisos de usuarios**:

  Cada usuario posee una serie de recursos y permisos asociados a los mismos, creando así un sistema de control de permisos que permite compartir recursos entre distintos usuarios.

- **Encriptación personal AES-256**:

Usando el algoritmo de encriptación AES-256 [Daemen, 2002] , los datos del usuario están encriptados con su clave, todo ello de forma transparente al usuario, realizándose la encriptación y desencriptación automáticamente. Las claves sólo las conoce el usuario, por lo que es imposible desencriptar el contenido por parte de un hipotético administrador del servidor.

- **Descargas y uploads parciales**

  El servidor permite las descargas y las subidas parciales, por lo que si una de estas operaciones es interrumpida (ya sea manualmente o por un fallo en la red) puede ser reanudada más adelante. Esta característica es crucial para manejar archivos de gran tamaño que tardan más en ser transferidos.

- **P2P**

  El servidor es capaz de generar, a partir de un archivo almacenado, su correspondiente enlace torrent o magnet para poder compartir ese archivo vía peer-to-peer.

- **Generación de tokens para validación**

  Usados para validar una cuenta o recuperar una contraseña, el servidor generará tokens aleatorios cuya validez tendrá una duración determinada.ç

- **Generación de enlaces públicos**

  Para compartir los archivos con personas que no son usuarios de la plataforma, el servidor es capaz de generar enlaces únicos para cada fichero y salvaguardando que el fichero esté disponible únicamente a los poseedores del enlace o a los usuarios con permisos sobre el mismo.

8.1.2 Arquitectura general

A continuación exponemos una figura de la arquitectura general del servidor:

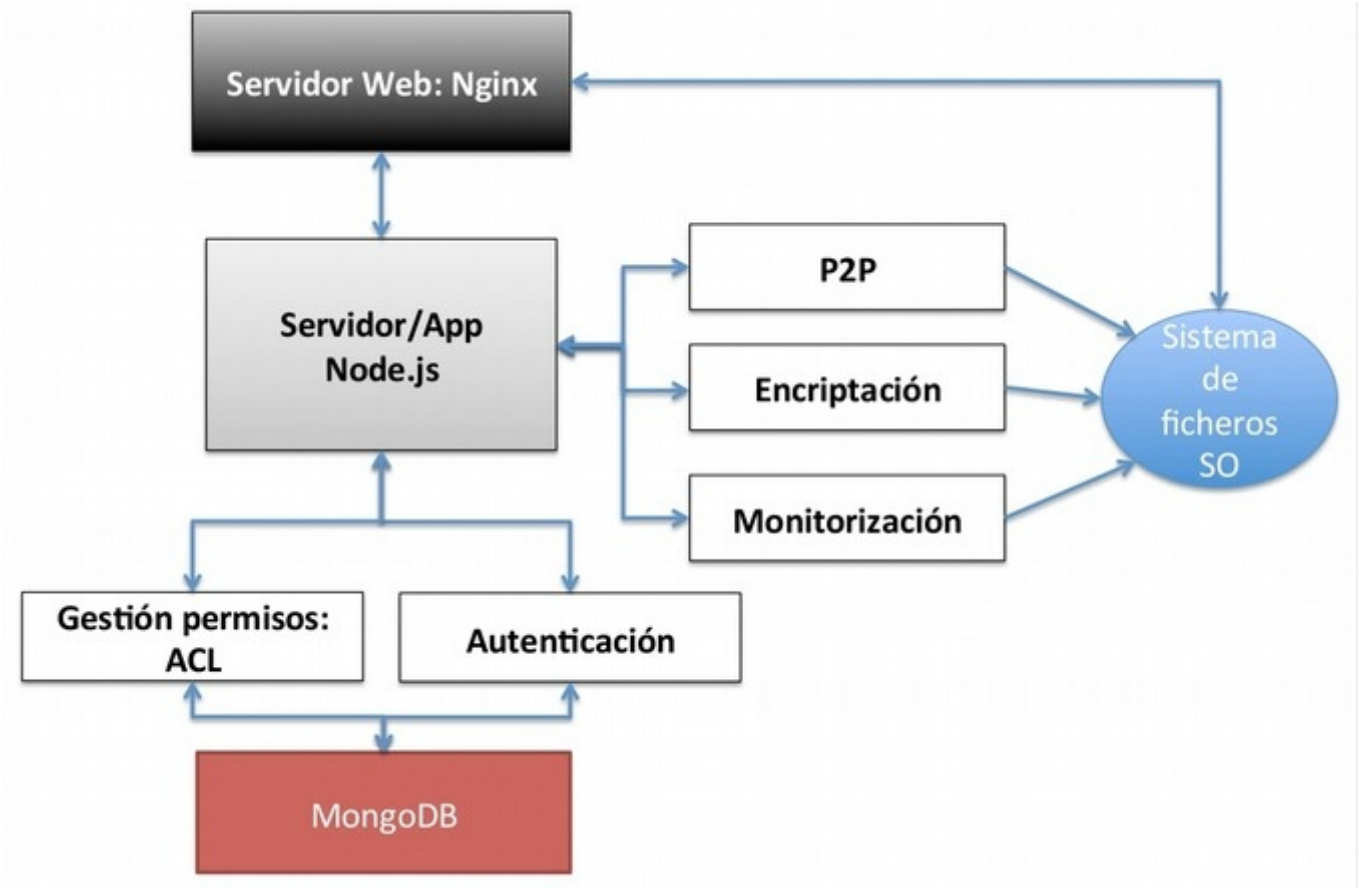


Figura 13: Arquitectura general del servidor

Ejemplo ciclo petición – respuesta

A continuación se detalla el proceso que sigue una petición HTTP desde que llega al servidor web hasta que se emite una respuesta a la misma:

1. La petición HTTP(S) llega a NGINX que la redirige a Node.js
2. La aplicación Node.js recoge la petición, que contendrá en sus parámetros usuario y contraseña y realiza dos consultas a la base de datos: Para autenticar al usuario y para consultar sus permisos sobre la operación y recurso deseado.
3. La base de datos MongoDB devuelve a Node.js las consultas realizadas.
4. Si el resultado de la consulta es positivo, Node.js redirige nuevamente la petición a Nginx
5. Si el resultado es negativo, Node.js envía directamente la respuesta al cliente. (401, Unauthorized ó 403, Forbidden)
6. Nginx recibe la petición interna y responde interactuando directamente con el sistema de ficheros del sistema operativo.

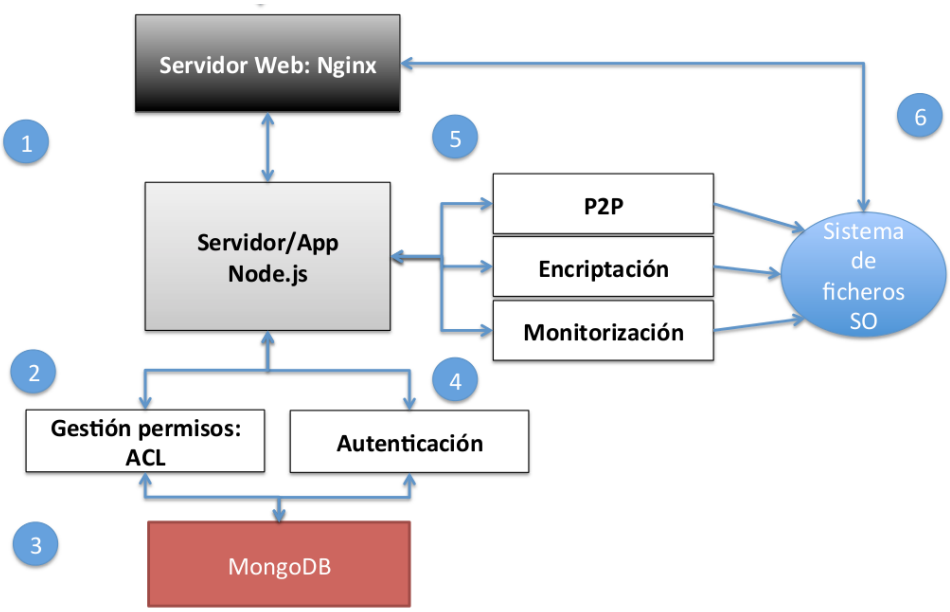


Figura 14: Fases ciclo petición-respuesta

Nótese como en el punto 4, Node.js puede realizar acciones adicionales, como generar un enlace P2P, encriptar el archivo o monitorizar el sistema de ficheros.

8.1.3 Servidor Web NGINX

En este proyecto, Nginx se encarga de:

1. Servir y tratar archivos estáticos: Todas las operaciones sobre archivos (leer ,escribir, borrar un archivo) son realizadas por Nginx a través del protocolo HTTP y su extensión, el protocolo WebDAV [IETF, 2007]
2. Optimizar el ancho de banda: Comprimiendo las respuestas y activando el caching para no devolver archivos que no han cambiado en el cliente.
3. Control de acceso: Nginx sólo servirá archivos estáticos si Node.js le envía peticiones internas para ello.

A continuación se expone el esqueleto del archivo de configuración de Nginx que se ha escrito:

```
upstream dandelion_backend { #Node app address
  server 127.0.0.1:3000;
}
server {
    server_name localhost;
    listen 80;
    listen 443 ssl default_server;
    #Serve angular web app
    location / {
                    ...
    }
   #API requests are forwarded to Node
    location /api {
    proxy_pass http://dandelion_backend;
    }
    location /authorizated_download{
                    internal;
    }
    location /authorizated {
      allow 127.0.0.1;
      deny all;
    }
}
```

Se puede observar, en primer lugar, que le indicamos a Nginx donde está la aplicación/servidor Node.js mediante el bloque *upstream dandelion_backend*. Posteriormente tenemos 4 bloques *location*, en los que, se detalla el tratamiento que se le hace a la petición en función de su URL. Así pues, si la URL es "/" le serviremos al cliente los archivos de la aplicación web Angular (ver sección *8.2 Cliente web*). Si la URL comienza por */api/*, redirigemos esta petición al servidor Node (cuya dirección se ha especificado al comienzo en el bloque *upstream*).

Finalmente, el servidor Node.js tratará la petición y la redirigirá, o bien a */authorized_download* si es una descarga (GET), o bien a */authorizated* si se trata de otro tipo de petición (PUT, DELETE..).

Obsérvese que las cláusulas (directivas, en el contexto de Nginx): *internal; allow 127.0.0.1; deny all;* nos permiten restringir el tráfico únicamente a peticiones internas o que provengan de las IP específicadas. Hemos diferenciado entre peticiones *internas* de Nginx o de la propia máquina (127.0.0.1) ya que Nginx proporciona un mecanismo para servir archivos estáticos de forma rápida tras pasar una autenticación en un backend, conocido como *X-Accel-Redirect* [Nginx, 2015] .

El archivo de configuración completo se divide en dos archivos localizados en *server/dandelion/nginx.conf* yen *server/nginx/dandelion.conf* En ellos se encuentran, convenientemente documentadas y explicadas, otras características como la compresión, el caching o el soporte para SSL. Estos archivos pueden consultarse en el *Anexo 13.4 Configuración de Nginx.*

8.1.4 Servidor REST Node.js

La base de la solución de almacenamiento online reside en el servidor encargado de alojar y servir los archivos, sin embargo para acceder al mismo y realizar operaciones con él, ha de definirse una interfaz que será gestionada por una aplicación o servidor escrito en Node.js

Arquitectura

Nuestra aplicación Node.js se encargará principalmente de recibir las conexiones de Nginx y consultar la base de datos para comprobar la autenticación y autorización de las mismas. Asimismo, también se encargará de:

- **Encriptar/Desencriptar archivos:** Usando cifrado AES 256 y la clave del usuario.
- **Generar archivos .torrent y .magnet**: Para la posterior compartición de los archivos vía P2P
- **Gestionar de permisos:** Actualizando los permisos de usuarios y ficheros.
- **Monitorización del sistema de ficheros:** Manteniendo listas de los ficheros en el servidor de cada usuario, para permitir entre otros, la sincronización de ficheros con clientes de escritorio.
- **Subidas ("uploads" parciales)**: Subiendo archivos de gran tamaño en varias partes, lo cual permite que la subida no tenga que reanudarse desde el comienzo si la conexión entre un cliente y el servidor es interrumpida.
- **Generación de tokens**: Ya sea para autenticación, validación de la cuenta o para recuperar la contraseña.

Para implementar todo ello, se ha hecho uso de un servidor o aplicación desarrollado en express.js (framework desarrollado en Node.js) y una API REST. El servidor se encargará de asociar una ruta y método con una acción determinada. Por ejemplo, si el usuario quiere acceder un archivo haría una petición GET a:

/api/resources/nombre_archivo

Nuestro servidor express.js, leería la petición y buscaría la función asociada que se encarga de manejar dicha petición, que en nuestro caso haría algunas consultas a base de datos para controlar los permisos y redirigirá la petición a NGINX

Por tanto, en nuestra aplicación hemos definido rutas para todas las posibles acciones anteriores (Encriptar/Desencriptar archivos, Gestionar permisos) y unos manejadores (o *handlers*) asociados.

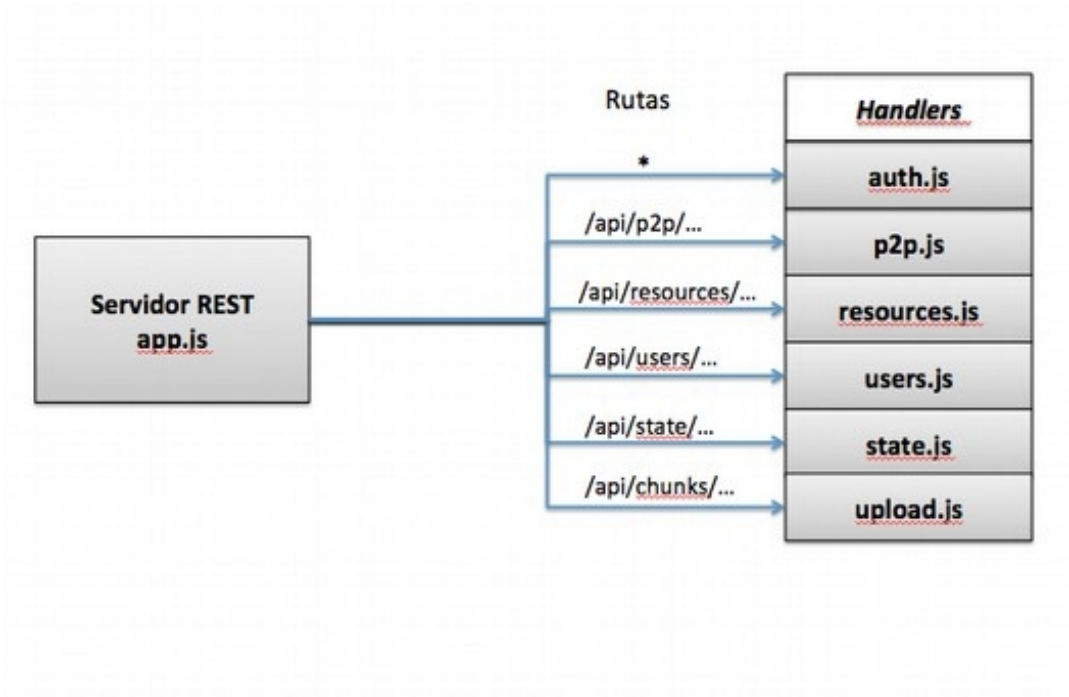


Figura 15: Rutas y handlers asociados servidor REST

Cómo se puede observar, cada una de las rutas descritas en la imagen se corresponde con un archivo encargado de manejar la acción que esta representa. Nótese que el manejador de autenticación se ejecuta para todas las rutas (*), ya que antes de realizar cualquier acción nuestro servidor comprueba que el usuario esté correctamente autenticado.

8.1.5 Autenticación

Para implementar la autenticación del servidor hemos utilizado un acercamiento basado en tokens. En concreto se ha hecho uso del estándar JWT [Jones, 2015] (explicado en la sección *6. Recursos*) que establece cómo deben generarse y manejarse tokens en aplicaciones web..

A la hora de elegir este método en lugar de otros más tradicionales se han tenido en cuenta los siguientes factores:

1. La API del servidor web será usada desde varios medios: navegador y clientes de escritorio. Por ello no debemos usar cookies o sesiones web.
2. En base al punto anterior, cada petición debe llevar unas credenciales que permitan saber al servidor que el usuario está autenticado.
3. Las credenciales deben ser lo suficientemente complejas para no ser reproducidas de forma intencionada por un hipotético atacante.

La autenticación de la aplicación consta de tres pasos:

1. El cliente inicia sesión en la aplicación, enviando sus credenciales en la petición HTTP.
2. Servidor valida las credenciales, y si estas son correctas, envía un token único al cliente.
3. El cliente deberá utilizar el token proporcionado en cada llamada.

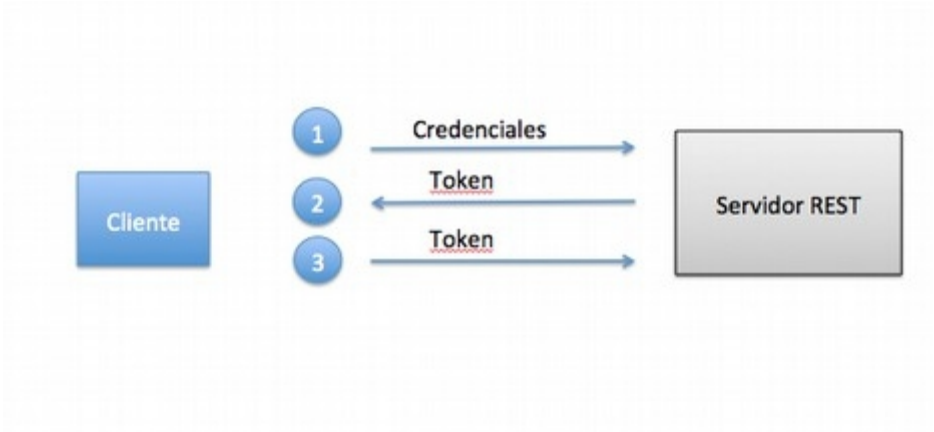


Figura 16: Proceso de autenticación usando tokens JWT

Destacar lo siguiente:

- El token puede tener fecha de expiración, obligando al usuario a renovarlo por cuestiones de seguridad. En el caso que ocupa a este proyecto, el token tiene una expiración de 24 horas.

- La contraseña del usuario se guarda encriptada en la base de datos, usando para ello el protocolo Bcrypt. Cuando se realiza el paso 1 en el diagrama anterior, primero se encripta la contraseña y luego se compara la cadena encriptada con la que se encuentra en la base de datos. Esta medida de seguridad adicional provoca que si la base de datos es comprometida el atacante no pueda acceder a las contraseñas almacendas.

Con esta estrategia se envía el usuario y token en claro, por lo que es indispensable el uso de *HTTPS* (que hemos activado en la configuración de Nginx) para evitar que un tercero intercepte datos de la conexión y sea capaz de averiguar los tokens usados.

8.1.6 Representación de datos

Los datos de la aplicación se han representado usando la base de datos noSQL MongoDB.

Cómo se explica en la sección *6. Recursos*, MongoDB no utiliza el concepto clásico de tablas y filas SQL. En su lugar se utilizan conceptos similares: Colecciones y Documentos.

En el servidor REST, los datos han sido representados de la siguiente forma:

- Una colección de usuarios, en la que cada usuario es un documento con la siguiente estructura:

```
{ "_id" : ObjectId("55e49375241df0f807afcac2"),
"name" : "yarilo", "password" : "$2a$10$P2vBgVFdFMkYr3LixkyEN.Z6qMXQ.iAuIOXpiuzvJVNcScSL/OLPG", "email" :
"yarilo.villanueva101@alu.ulpgc.es" }
```

  Donde:

  - **_id**: Es un id unívoco asignado a cada usuario, de tipo ObjectID [MongoDB, 2015].
  - **name**: Nombre del usuario.
  - **password**: Password del usuario, cifrada usando Bcrypt.
  - **email**: Email del usuario.

- Una colección de recursos por cada usuario existente en la colección anterior. Cada recurso es un documento de la forma:

```
{"_id": ObjectId("55ff0f298d1dc3000063386e"),
"name": "sample_folder/sample.txt",
"resource_kind": "file",
"owner": "yarilo",
"mtime": 1442778921000,
"size": 5988,
"parent": "sample",
"locations": [ "/real/location/on/filesystem",],
"edit": [    "yarilo",    "luis"],
"view": [   "david"  ]}
```

  El significado de cada campo es:

- **_ id**: Identificador único del recurso, de tipo ObjectID
- **name**: Ruta completa del recurso en el sistema de directorios, teniendo como directorio raíz la carpeta del usuario.
- **resource_kind:** Tipo de recurso: Archivo o directorio.
- **owner**: Dueño del recurso.

- **mtime**: Último tiempo de modificación del archivo (se usa la hora del servidor).
- **size**: Tamaño en bytes.
- **parent**: Carpeta contenedora del recurso actual.
- **locations**: Array de localizaciones físicas reales. Si este recurso está compartido por varios usuarios, la localización física de cada usuario aparecerá aquí.
- **edit:** Array de usuarios que tienen permisos de edición en este archivo.
- **view**: Array de usuarios que tienen permisos de lectura en este archivo

Se puede observar que existe una relación clara entre la colección de recursos y la de cada usuario, pudiendo tener un usuario muchos recursos y un recurso muchos usuarios. En este punto, se han adoptado varias decisiones de diseño:

1. **Usar una colección de recurso por usuario:**

   Duplicando los recursos compartidos en cada colección en lugar de tener una sóla tabla con todos los recursos. Este decisión ofrece dos ventajas y un inconveniente:

   - Permite que cada usuario tenga su propio espacio de nombres, que no pueden coincidir con el de otros usuarios. Ello hace que la lógica al compartir archivos sea más sencilla, no teniendo que "recordar" la aplicación cómo guardaba un determinado recurso cada usuario.
   - Escalabilidad: A medida que el número de usuarios y recursos crece, las operaciones sobre la tabla de recursos que contiene todos los recursos de todos los usuarios serían más costosas.
   - Por contra, cada vez que se hace un cambio en un recurso compartido debe replicarse ese cambio en todas las colecciones y recursos compartidos de cada usuario (ya que los datos están duplicados).

2. **En los arrays de recursos se guardan los nombres de los usuarios con permisos:** Y no al revés, ya que los usuarios de cada archivo ya están representados por su colección de recursos. Esta decisión permite saber rápidamente cuantos recursos tiene un determinado o usuario o qué usuarios comparten un mismo archivo.

Cabe destacar que el diseño aquí adoptado es similar al que usan soluciones ampliamente establecidas

como Dropbox [Koorapati, 2014].

Finalmente, a la hora de decidir como representar el árbol de directorios se ha utilizado el esquema propuesto por MongoDB: "*Materialized Paths"* [MongoDB, 2008], en el que se guarda en el nombre de cada documento la ruta del mismo, tal que:

```
"name": "sample_folder/sample.txt",
```

Pudiendo averiguar con simples expresiones regulares todas las carpetas padres o hijas de un determinado recurso, evitando así accesos a la base de datos.

8.1.7 Autorización y gestión de permisos

En base a la representación de datos anteriormente representada, se ha creado un sistema de autorización (qué puede hacer cada usuario) y gestión de permisos específico.

De esta forma se han definido tres tipos de permisos que cada usuario puede tener sobre un recurso determinado:

- **Propietario**: Tiene control total sobre el recurso y puede compartir el recurso.
- **Edición**: Puede acceder, modificar o borrar el recurso, pero no compartirlo.
- **Lectura**: Sólo puede acceder al recurso.

Cuando un usuario realiza una operación sobre un determinado recurso, el servidor REST lanza una consulta a la base de datos preguntando si ese usuario tiene permisos para realizar la acción que desea sobre ese recurso. En caso afirmativo, se modificará la base de datos y el sistema de ficheros, tal y como se muestra en el siguiente esquema:

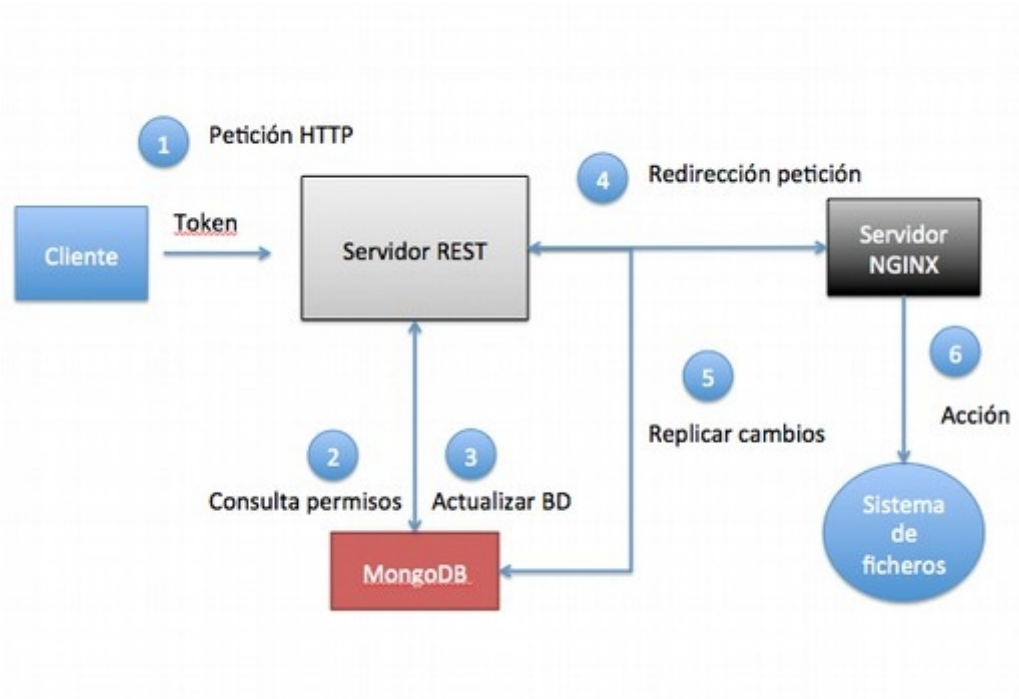


Figura 17: Esquema autorización petición HTTP

Siendo cada uno de los pasos:

1. El cliente emite una petición HTTP con un token de autenticación, expresando que un usuario desea realizar una operación sobre un recurso.

2. Si la petición está correctamente autenticada, el servidor REST envía una consulta a MongoDB preguntando si el usuario tiene permisos para realizar la acción deseada.

3. En caso afirmativo, la petición es reenviada a NGINX (en caso negativo, se devuelve un 403 en la respuesta).

4. Node redirecciona la petición a Nginx

5. El servidor REST replica los cambios realizados en el archivo compartido, tanto en las colecciones del resto de usuarios (cambiando el estado del documento que representa al recurso) como en el sistema físico (lanzando nuevas peticiones a NGINX para operar sobre archivos).

6. NGINX ejecuta la operación deseada sobre el sistema de ficheros.

El manejo de permisos se encuentra repartido entre los ficheros */server/routes/users.j*s. Y *server/routes/consistency.js*.

8.1.8 Recursos

Node.js se encarga de actualizar los permisos de los recursos del usuario en la BD, tal y como hemos visto en la anterior sección, haciendo uso de las funciones que se encuentran en */server/fs/routes* y de las siguientes llamadas HTTP asociados:

- GET */api/resources/** : Para obtener un recurso. No modifica la BD.
- MKCOL */api/resources/dirs/*:* Crea un directorio.
- PUT */api/resources/**: Crea cualquier tipo de archivo.
- COPY */api/resources/**: Copia un archivo de un lugar a otro
- MOVE */api/resources/**: Mueve un archivo de un lugar a otro.
- DELETE */api/resources/**: Borra un archivo.

En todos los casos se actualiza la base de datos como corresponda y se redirecciona la petición al servidor Nginx (que se encarga de lidiar con el sistema de ficheros) mediante *http-proxy* [Nodejitsu, 2015] . Mención especial a la llamada GET, que no modifica ni necesita acceder a la base de datos y cuya redirección se ha implementado haciendo uso de X-*Accel-Redirect* (como se ha comentado anteriormente) para una mayor rapidez.

8.1.9 Monitorización sistema de ficheros

La monitorización del sistema de ficheros comprende las siguientes llamadas HTTP:

- GET */api/state*
- GET /*api/state/list*

La primera llamada, */api/state*, se utiliza para generar desde cero la lista de archivos pertenecientes a un usuario. Esta llamada será ejecutada cuando inicie el cliente para comprobar si ha habido algún cambio mientras el cliente no era ejecutado.

Para generar la lista se ha utilizado *readdirp* [Lorenz, 2015], que analiza recursivamente una carpeta especificada. Para cada recurso físico analizado se crea un objeto como el que se muestra en el siguiente ejemplo:

Una vez se ha generado toda la lista de recursos, se envía la lista a la base de datos, actualizando los documentos que así lo requieran.

Por su parte, la segunda llamada, /*api/state/list* se utiliza simplemente para que el servidor devuelva al cliente la lista de recursos del usuario en cuestión (consultándola en la base de datos), una vez que el cliente ha arrancado y se ha generado la lista con la primera llamada.

8.1.10 Encriptación de recursos

La encriptación o desencriptación de archivos se realiza bajo las siguientes llamadas:

- PUT */api/encrypt/* :* Para encriptar un archivo, siendo * la ruta completa en el servidor del mismo.
- PUT */api/decrypt/*:* Para desencriptar un archivo, siendo * la ruta completa en el servidor del mismo.

Usando la librería *file-encryptor* [Modulus, 2015] y la contraseña de cada usuario como clave, los archivos son encriptados bajo el algoritmo AES-256 [Daemen, 2002]. Asimismo, los archivos encriptados son marcados en base de datos con el permiso *encrypted,* lo que nos permite consultar el estado de encriptación de un recurso con:

- GET a */api/status/encrypted/**

8.1.11 Subidas parciales

Dada su complejidad, las subidas parciales se han implementado en Node.js en lugar de Nginx, siendo Node el encargado de interactuar con el sistema de ficheros y coordinar la interrupción y reanudación de este tipo de descargas.

Para ello se hace uso de la librería *Flow.js y* las funciones implementadas en */server/routes/upload.js*.

Las llamadas al API correspondientes son:

- POST */api/chunks/ :* Sube un archivo y/o comprueba que se ha subido correctamente.
- OPTIONS */api/chunks/:* Comprueba que el servidor tiene los permisos necesarios (que permite CORS [Mozilla, 2015])
- GET */api/chunks/:* De uso interno para averiguar el número de trozos subidos correctamente.
- GET */api/chunks/download/:identifier :* De uso interno para averiguar el id de un determinado trozo del archivo.

El algoritmo seguido para subir archivos de forma parcial (interrumpiendo/reanudando la subida) se presenta en la siguiente página.

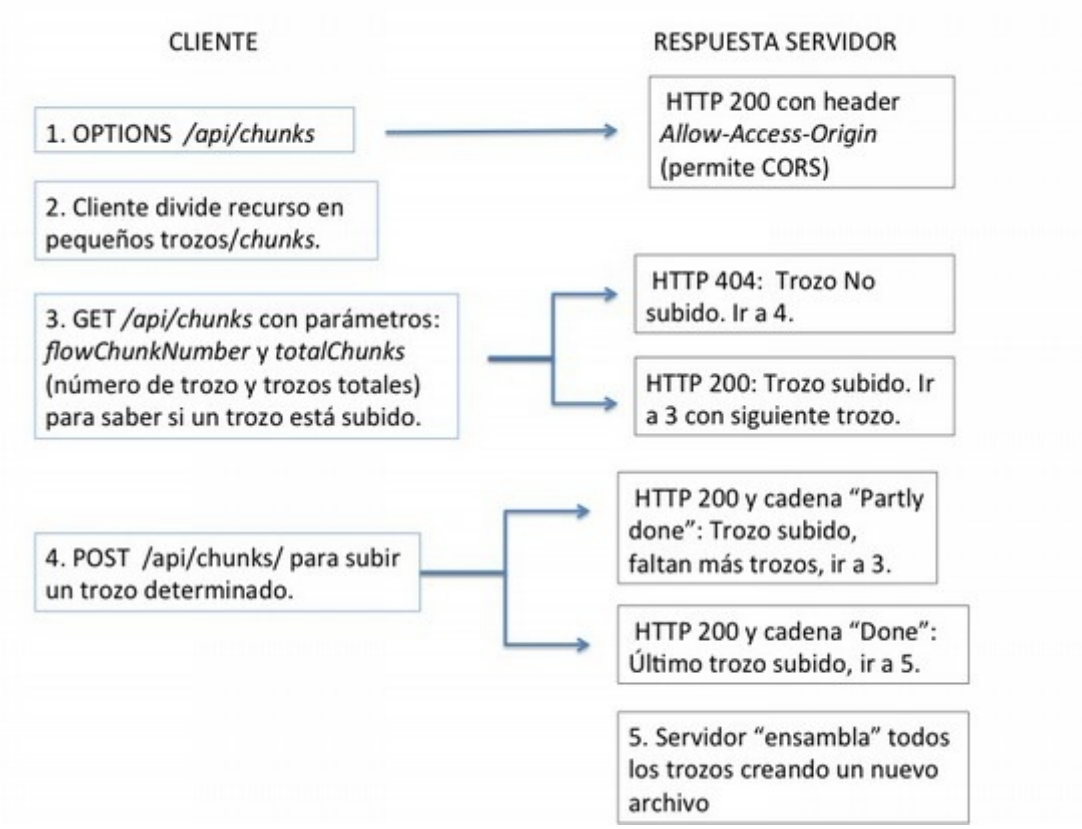


Figura 18: Esquema algoritmo subidas parciales

En caso de que se pare la subida por cualquier motivo (pausa manual, fallo en la conexión, etc), al reanudarse la misma, el cliente, al llegar al paso 3 comprobará que el servidor ya tiene los trozos que antes se habían subido y no los volverá a subir, por lo que la subida comenzará desde el punto en el que fue interrumpida.

8.1.12 Peer-to-Peer (P2P)

El servidor es capaz de generar torrent y magnet links utilizando las llamadas REST:

- GET */api/p2p/torrent/*
- GET */api/p2p/magnet/*

En ambos casos nos valemos de las funciones implementadas en la ruta *server/routes/app/p2p.js* y de las librerías *create-torrent* [Aboukhadijeh, 2015] *y magnet-link* [Goldman, 2015] , que, como su propio nombre indican, son capaces de generar un archivo torrent y un link magnet a partir de un archivo normal y un torrent respectivamente.

8.1.13 Generación de tokens para validación y cambio de contraseña

Cómo se verá en la sección *8.2.6 Interfaz*, se hacen uso de correos electrónicos para validar cuentas nuevas o reiniciar la contraseña de un usuario.

En los dos casos se sigue el mismo protocolo.

1. Usuario rellena un formulario (De nuevo registro o de recuperar contraseña)
2. Si los datos introducidos son correctos, se realizan los siguientes pasos: en el servidor REST:
    1. Se genera un token aleatorio
    2. Se guarda este token en el documento de la base de datos asociado al usuario. Tendrá validez de una hora
    3. Se crea un enlace de recuperación del tipo (haciendo uso de la librería estándar de Node.js Crypto [Node.js, 2015]):

        http://SERVER_ADDRESS/#/validate/d706ef911f1e0ab89f39129c553e82c17c5dba91

        ó

        http://SERVER_ADDRESS/#/reset/d706ef911f1e0ab89f39129c553e82c17c5dba91
    4. Se envía al usuario un correo con este enlace e instrucciones adicionales haciendo uso de Mailgun.

3. Cuando el usuario hace click en el enlace, se comprueba que el token usado es válido y no ha expirado.

4. Si se cumplen estas condiciones, se realiza la acción deseada: Validación de la cuenta o cambio de contraseña.

Nótese que los tokens generados aquí no son JWT ya que estos han sido diseñados para un proceso de autenticación en el que el usuario conoce sus credenciales , y en este caso el usuario carece de las mismas (bien porque está creando la cuenta o bien porque ha olvidado sus datos de acceso). Asimismo, la excesiva longitud de los tokens JWT no los hace idóneos para ser usados en enlaces por usuarios finales.

8.1.14 Generación de enlaces públicos

El servidor REST es capaz de generar enlaces públicos para archivos de forma que cualquiera que tenga acceso al enlace (sea usuario registrado o no) pueda acceder al archivo.

Para ello, una vez que el propietario del archivo decide compartir públicamente el mismo mediante el cliente web (cómo se explica en la sección *8.2.6 Interfaz*), éste realiza una petición GET a:

/api/token/:id

Dónde *:id* es el identificador del recurso a compartir. La petición contendrá en la query los siguientes parámetros:

- *user:* Indiciando el nombre del usuario que proporciona el recursos
- *token:* Indicando el token de autenticación del usuario.

Si la autenticación y autorización son correctas, el servidor genera un token aleatorio de 20 bytes(utilizando Crypto, al igual que en la sección anterior), guarda el token en el documento del recurso y lo envía en la respuesta. A partir de este token, el cliente genera un enlace público del tipo:

http://SERVER_ADDRESS/api/resources/55faf818d3687deff0922751/?user=yarilo&public_token=d61904e6ab9e89fa1b408bf9c4811f24ab722cb9

Cada vez que se usa este enlace, el servidor comprueba que el recurso especificado (55faf81..) tiene el token público que se indica en el enlace ofreciendo o denegando el acceso al archivo en función de esta condición.

8.1.15 Modo cluster

Normalmente todas las aplicaciones Node.js se ejecutan en un sólo proceso. Si bien ello simplifica enormemente la gestión de memoria y similares, en múltiples ocasiones puede ser beneficioso tener varios procesos compartiendo una misma serie de tareas y así poder paralelizar las mismas.

Por ello, se posibilita la opción al usuario de ejecutar la aplicación en el denominado "Modo cluster" en el cual se crean tantos procesos como núcleos tenga la CPU del usuario. Dichos procesos compartirán el mismo socket y puerto y se repartirán automáticamente las peticiones entrantes, lo cual redundará en una mayor eficacia y concurrencia del servidor.

Por defecto, el servidor se ejecuta usando este modo.

8.1.16 Instalación y uso

Instalación

Para facilitar la instalación por parte del usuario final se ha creado un script denominado en bash denominado `deployment`, por lo que el usuario tan sólo tiene que ejecutar desde una terminal:

```
sh deployment/deploy.sh
```

Y se arrancará el script que se encargará de bajar e instalar las dependencias necesarias (Nginx, Node y las librerías adicionales), modificar los permisos de la carpeta donde se almacenarán los archivos, crear el usuario de administración y arrancar el servidor. El script puede ser ejecutado bajo Ubuntu Linux, Red Hat OS (y derivados) y Mac OS X.

Uso

Una vez ejecutado el script anterior nuestro servidor estará arrancado totalmente. Si se desea hacerlo de forma manual simplemente hemos de arrancar Nginx y la app Node.js por separado:

```
sudo nginx #Arranca Nginx
npm start # Arranca el servidor de Node.js
```

Nótese que Nginx se ejecutará de forma distinta en función del sistema operativo, mientras que la aplicación Node siempre se ejecutará desde el directorio raíz del proyecto mediante la instrucción anterior.

8.1.17 Despliegue

Para ofrecer un mayor control al tutor del proyecto, así como para probar en un entorno real la parte servidor de este proyecto y su aplicación web, se ha hecho uso de una instancia de Amazon EC2 en la que se ha desplegado el código de dicho servidor y aplicación web.

Para ello se ha:

- Contratado una máquina gratuita en AWS con las siguientes características:
  - 1 CPU virtual Intel Xeon a 2,5 Ghz
  - 1 GB Memoria
  - Ubuntu 14.04 LTS

  - 10 GB Almacenamiento EBS

- Configurado la máquina para que tenga acceso al exterior:
  - Asignándole una IP pública: http://dandelion.redes.dis.ulpgc.es
  - Definiendo por defecto los puertos abiertos:
    - SSH, puerto 22: Todo el tráfico entrante/saliente
    - HTTP, puerto 80: Todo el tráfico entrante/saliente
    - HTTPS, puerto 443: Todo el tráfico entrante/saliente
    - Resto puertos cerrados.
- Ejecutado el script de deployment mencionado anteriormente: *deployment/deploy.sh* (que puede consultarse en el *Anexo 13.6*)
- Instalado y hecho uso del gestor de procesos PM2 , que permite, entre otras cosas, reiniciar automáticamente la aplicación Node.js ante cualquier eventualidad, vigilar su consumo de memoria o aplicar los cambios que podamos hacer "en caliente" (sin necesidad de parar y arrancar el servidor Node.js)

```
ubuntu@ip-172-31-30-53:~$ pm2 show 0
Describing process with pid 0 - name app
```

status	online
name	app
id	0
path	/home/ubuntu/dandelion-backend/server/app.js
args	
exec cwd	/home/ubuntu/dandelion-backend
error log path	/home/ubuntu/.pm2/logs/app-error-0.log
out log path	/home/ubuntu/.pm2/logs/app-out-0.log
pid path	/home/ubuntu/.pm2/pids/app-0.pid
mode	fork_mode
node v8 arguments	
watch & reload	✘
interpreter	node
restarts	0
unstable restarts	0
uptime	18D
created at	2015-05-12T17:37:59.945Z

```
Revision control metadata
```

Figura 19: Gestor de procesos PM2, detalles servidor Node.js

8.2 Cliente web

En este apartado se describirá el cliente web desarrollado para integrarse con el servidor Nginx/Node.js implementado.

8.2.1 Visión general

Para realizar el cliente web nos hemos servido del framework AngularJS, que permite crear webs dinámicas siguiendo una serie de patrones.

De esta forma, nuestro cliente web posee las siguientes características:

- **Aplicación web "Single page":**

  El cliente ha sido desarrollado como una aplicación web con una sóla página HTML, y que genera dinámicamente el resto de contenidos, lo cual libera de recursos al servidor y minimiza ancho de banda, haciendo más fluida la experiencia del usuario.

- **Aplicación web "responsiva"**

  La aplicación web es capaz de modificar su presentación en función del tamaño de la pantalla del cliente, ajustando los contenidos mostrados a la misma.

- **Acoplamiento con el servidor Nginx/Node.js**

  Utilizando el mismo protocolo que en el servidor y una API REST, el cliente es capaz de transferir archivos con el servidor.

- **Desarrollado en Javascript:**

  Lo que facilita la conexión al servidor, que utiliza el mismo lenguaje (Node.js)

- **Gestión de permisos**

  El cliente es capaz de modificar los permisos de los recursos, permitiendo la compartición de los mismos entre varios usuarios.

- **Generación de magnet links y torrents**

  A partir de un determinado archivo el cliente es capaz de generar un magnet link o torrent para compartir dicho archivo con otros clientes sin necesidad de que el servidor intervenga.

- **Generación de links públicos a archivos**

La aplicación web es capaz de generar links para compartir un archivo de forma pública, pudiendo compartirse éste con personas que no sean usuarias de la aplicación.

- **Patrón MVC y MVVM :**

  El patrón usado es principalmente el del Modelo – Vista – Controlador [Glenn, 1988], con la salvedad de que el modelo es generado por AngularJS implícitamente en las vistas a partir de los datos obtenidos en la base de datos (ver sección *8.1.7 Autorización y Gestión de permisos*), insipirándose en el patrón Modelo-Vista-VistaModelo [Gossman, 2005]

8.2.2 Arquitectura general

El cliente web ha sido desarrollado siguiendo se conforma de una serie de vistas que son renderizadas por el navegador del cliente, cuya lógica está definida por los controladores, que interactúan entre sí y con el servidor a través de unos determinados servicios, como se aprecia en la siguiente figura:

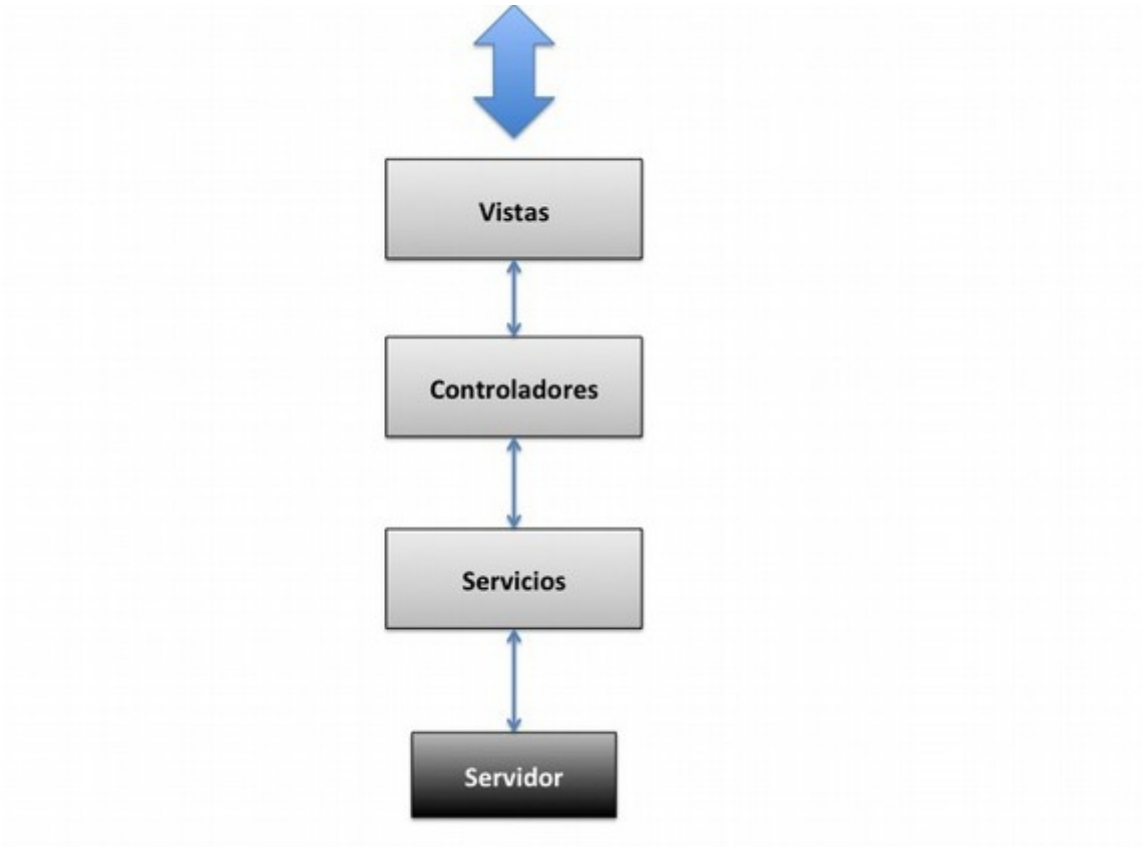


Figura 20: Arquitectura general cliente web

En las siguientes secciones se detallarán estos aspectos del cliente.

8.2.3 Vistas

Las vistas son los archivos HTML que son finalmente renderizados en el navegador del usuario. Además de código HTML, contienen etiquetas o directivas [Google, 2010] que son capaces de dotar de un cierto comportamiento y extensibilidad a una sección del código de la vista. Cabe destacar que AngularJS permite utilizar varios archivos HTML para formar una sóla vista, lo que permite reutilizar varios archivos en la elaboración de múltiples vistas.

Todas las vistas se encuentran en la carpeta *client/app/views/.* Las podemos dividir en función de si son accesible por un usuario autenticado o no:

Vistas usuario registrado/no registrado y no autenticado

Conformadas principalmente por la vista de login, recuperar contraseña y registrarse, correspondientes a los archivos: *login.html, forgot_password.html y signup.html.*

Vistas usuario registrado autenticado:

Accesibles sólo para usuarios registrados y autenticados, estas vistas son:

- **Vista principal:** En la que se muestra el árbol de directorios y se realizan las acciones principales. Conformada por los archivos *index.html* y *resources_list.html.*
- **Vista configuración:** Contiene lo relativo a la configuración del usuario y se encuentra en el archivo *configuration.html*
- **Vistas modales o auxiliar:** Conforman todo lo relativo a los diálogos de la aplicación. Cómo su nombre indica, se encuentran en la carpeta */client/app/views/modals.* La aplicación web utiliza numerosas vistas de este tipo, siendo principalmente:
    - *alert.html*: Vista genérica para crear alertas modales que informen al usuario del estado de una acción ("archivo descargado con éxito", "Error borrando el archivo"..etc).
    - *new-directory.html*: Vista modal para crear un directorio nuevo
    - *p2p.html*: Vista para que el usuario decida si quiere compartir el archivo a través de un torrent o a través de un link magnet.
    - *resource-details.html*: Muestra los detalles del archivo seleccionado.
    - *share.html*: Vista para compartir el archivo seleccionado.
    - *permissions.html:* Vista que muestra los permisos del recurso seleccionado.
    - *move.html :* Vista para mover un archivo o carpeta a otra carpeta.
    - *public_link.html:* Vista para generar enlaces hacia archivos y compartirlos públicamente
    - *torrent.html*: Vista para descargar el torrent generado a partir de un determinado archivo.

Todas ellas hacen uso del controlador principal y del controlador modal, determinando el primero el

contenido de la vista y el segundo el comportamiento de la misma

Un ejemplo de estas vistas lo podemos ver en la vista de detalles de un archivo:

Figura 21: Vista modal: resource-details.html

En la sección *8.2.6 Interfaz* se explicará de forma pormenorizada cada una de las acciones posibles en las vistas aquí mencionadas.

8.2.4 Controladores

Los controladores son los encargados de implementar la lógica y la funcionalidad detrás de las acciones de las vistas. Asimismo, interactúan con otros controladores y con el servidor a través de los servicios (expuestos en la sección siguiente).

Controlador principal

El controlador principal se encarga de manejar la lógica principal de la aplicación, manejando la vista principal y la vista de login. En concreto, este controlador se encarga de realizar las siguientes acciones:

- Loguear al usuario: Haciendo uso del *Servicio de login*
- Mostrar y actualizar la lista de recursos del usuario: Realizando peticiones al servidor mediante el S*ervicio de interfaz* y el estado actual del árbol de directorios mediante el *Servicio de Directorios*.

- Mostrar los iconos en la vista principal usando el *Servicio de Datos*
- Efectuar la lista de tareas del menú contextual, a través del *Servicio de interfaz*:
  - Actualizar permisos sobre un archivo, compartiéndolo.
  - Borrar un archivo
  - Generar un enlace torrent/magnet
  - Generar un enlace público
  - Encriptar/desencriptar un archivo
  - Mostrar los detalles de un archivo
  - Mover archivos entre carpetas
- Abrir todas las vistas modales apoyándose en el *Controlador Modal*

  El código de este controlador se encuentra bajo */client/app/scripts/controller/home-controller.js*

Controlador cabecera

El controlador cabecera se encarga, como su propio nombre indica, de definir la mayor parte de acciones presentes en la cabecera de la aplicación:

Figura 22: Cabecera aplicación web

Por ello, se encarga de:

- Crear nuevos directorios
- Subir nuevos archivos
- Fijar las opciones de configuración del usuario, más concretamente, cambiar usuario y contraseña.

Para todas esas acciones, el controlador llama al controlador modal, generando la vista correspondiente y posteriormente realiza las peticiones pertinentes a través del *Servicio de interfaz*. El código del

controlador cabecera se encuentra en */client/app/scripts/controller/header-controller.js*

Controladores modales

Los controlador modales se encargan de generar todas las vistas modales, pasando los datos del resto de controladores a las vistas generadas, así como recogiendo el input del usuario para pasarlo de vuelta a dichos controladores. Estos controladores se encargan por tanto de todas las vistas auxiliares definidas en las sección *8.2.3*.

El código de todos estos controladores se encuentra bajo */clients/app/controllers/modal-controller.js*.

8.2.5 Servicios

Servicio de login

El servicio de login se encarga simplemente de guardar la información introducida en la vista de login para que sea consultada por los controladores, especialmente por el controlador principal. De esta forma, cuando un controlador quiera efectuar una acción en en función de si el usuario está o no logueado, consultará las credenciales del usuario haciendo uso de este servicio.

El servicio de login se encuentra en */client/app/services/login.js*

Servicio de interfaz con el servidor

En este servicio se implementan todas las llamadas a la API REST para comunicarse con el servidor.

Son definidas aquí por tanto, las peticiones de la API explicadas en la sección del servidor. Este servicio, que se puede encontrar en *client/app/services/interface.js* se usa principalmente desde el controlador cabecera y el controlador principal.

Para más detalles sobre la API REST del servidor, se puede consultar el *Anexo 13.7 API REST Servidor Node.js*

Servicio de datos

El servicio de datos se encarga de pasar todos aquellos datos estáticos del cliente a los distintos controladores para posteriormente ser renderizados en las vistas. En conreto, este servicio se encarga de definir los tipos de icono que se verán en las diferentes vistas. De esta forma, si se quiere añadir un icono nuevo, cambiarlo o simplemente añadir algún elemento visual más, se incluirá en este servicio que, de manera centralizada, será accedido por el resto de controladores.

Este servicio se encuentra en *client/app/services/data.js*

Servicio de directorios

El servicio de directorios es un servicio que es consultado por el Controlador principal y el Controlador cabecera para obtener la lista de recursos del usuario, averiguar en que directorio se encuentra el usuario o cambiar el mismo. Su información será posteriormente usada en la vista principal mientras el usuario navega por el árbol de directorios.

Este servicio se encuentra en *client/app/services/directories.js*

8.2.6 Interfaz

En esta sección se describirá la interfaz de la aplicación web y se detallarán las posibles acciones de la misma, en base a los casos de uso descritos en la sección *7.3*. Asimismo se procurará describir el flujo que sucede desde la aplicación web hasta el servidor.

Iniciar sesión

La vista inicial en la que el usuario debe introducir las credenciales es la siguiente:

Figura 23: Pantalla de iniciar sesión

Si las credenciales son introducidas correctamente, el usuario será redirigirá a la vista principal. De lo contrario se instará al mismo a introducir las credenciales nuevamente.

Registro usuario

Para registrarse el usuario debe hacer click en “Registrarse” en la vista de login descrita anteriormente.

Una vez hecho eso, verá la siguiente pantalla:

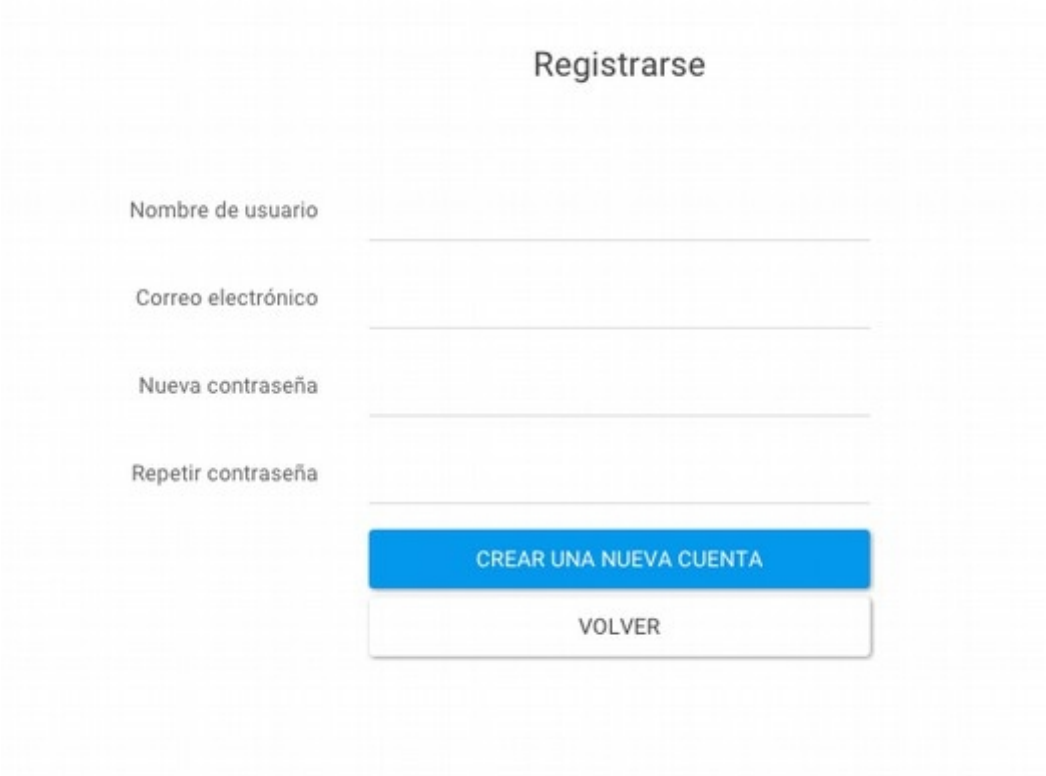


Figura 24: Formulario de registro

Al completar los campos correctamente (la aplicación web realiza pequeñas validaciones como asegurar que el formato del correo está correcto o la contraseña no se repite), el cliente web envía una petición al servidor. El servidor se encarga de realizar las comprobaciones pertinentes (usuario no duplicado, correo no duplicado, etc) y envía un correo para validar la cuenta al usuario:

Figura 25: Correo validación de cuenta

Cómo se observa en la captura de pantalla, el usuario debe hacer visitar el enlace proporcionado para validar la cuenta. Al hacerlo verá la siguiente pantalla:

Figura 26: Validación de cuenta satisfactoria

Los enlaces de validación generados tienen una hora de expiración. Si la validación falla, bien porque el enlace ha expirado o bien porque ya ha sido usado o es inexistente, el usuario verá la siguiente pantalla:

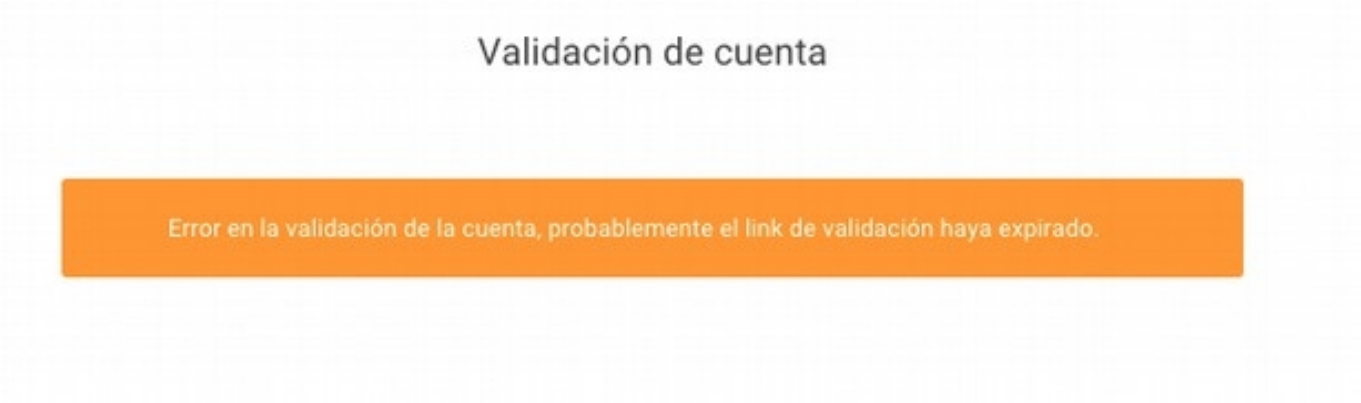


Figura 27: Validación de cuenta fallida

Recuperación de contraseña

En caso de olvidar su contraseña, el usuario puede recuperarla haciendo click en *"¿Olvidó su contraseña?"* en la vista de login descrita anteriormente. En ese caso, se le mostrará la siguiente pantalla:

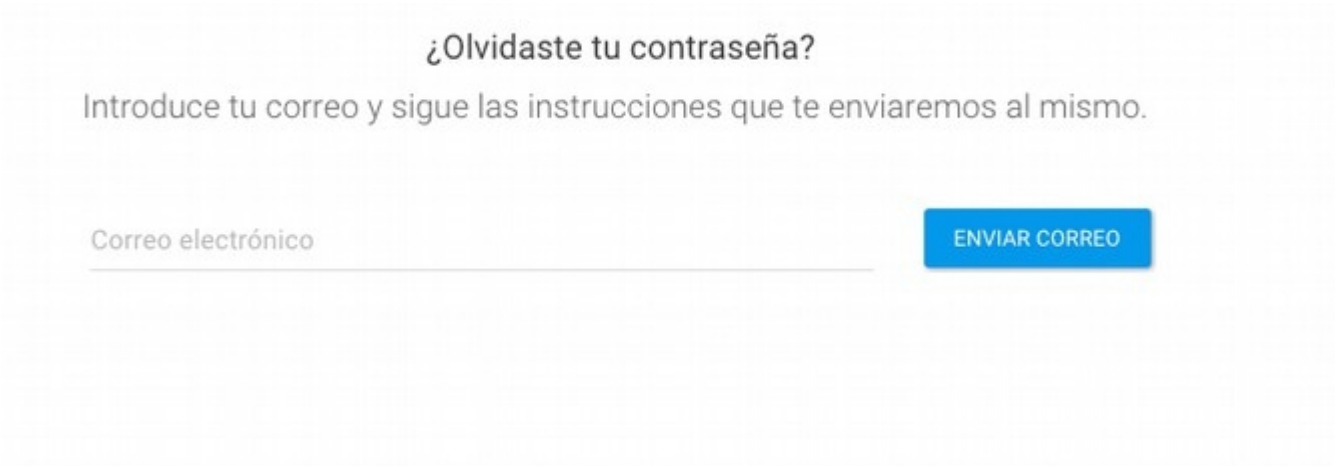


Figura 28: Vista "¿Olvidó su contraseña?"

Al introducir el correo correctamente, se le envía un correo al usuario con un enlace de recuperación:

Figura 29: Correo resetear contraseña

Que le enviará a la siguiente pantalla:

Figura 30: Vista de resetear contraseña

El enlace de recuperar contraseña tiene una expiración de una hora. Si dicho enlace no ha expirado o no ha sido usado anteriormente, se permitirá el cambio de contraseña y se instará al usuario a iniciar sesión. De lo contrario, se informará al usuario de que ha habido un error.

Vista principal

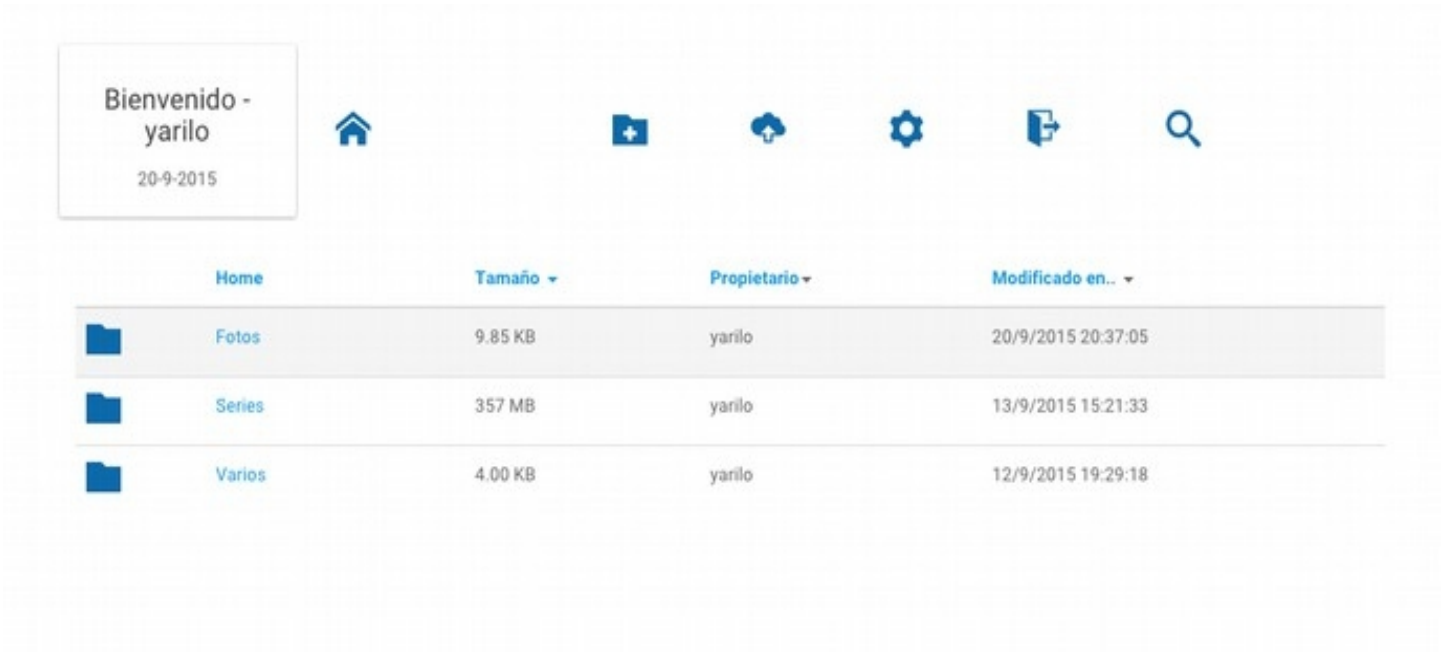


Figura 31: Vista principal

La pantalla principal es la representación del árbol de directorios de los recursos del usuario. Sólo se puede acceder a ella una vez el usuario haya sido logueado exitosamente en la *vista de login.*

La vista muestra los directorios y archivos del usuario así como permite la navegación por el árbol de directorios. De igual forma muestra una cabecera que permite realizar las actividades más comunes (añadir directorio, subir ficheros, buscar..).

Al hacer click derecho en un elemento, la vista despliega un menú contextual, tal y como se muestra a continuación:

Figura 32: Menú contextual

Al hacer click sobre una de estas acciones la vista contacta a través del controlador y este del servicio, con el servidor para llevarla a cabo. Ciertas acciones requieren de más feedback o control por parte del usuario, por lo que para cada una de ellas se genera una vista auxiliar en forma de modal (ver *Vistas* en sección anterior*)*.

Examinando un recurso

Al hacer click en "*Detalles"* en el menú contextual anteriormente descrito, se muestra el siguiente diálogo modal:

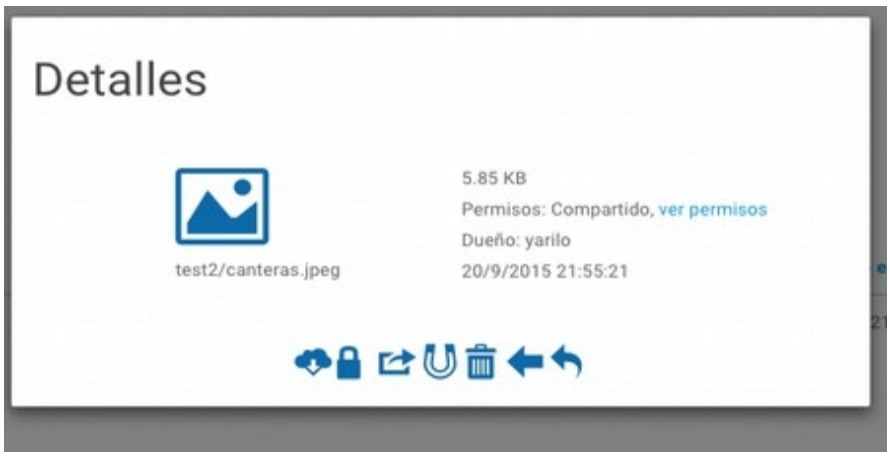


Figura 33: Diálogo detalles de recursos

En donde se pueden consultar todos los detalles del archivo. Si se hace click en *"Ver permisos"*, se mostrará el siguiente diálogo:

Figura 34: Diálogo de permisos

Creando una carpeta

Al intentar crear una carpeta pulsando sobre el siguiente icono: 

El usuario se encuentra con el siguiente diálogo de interacción:

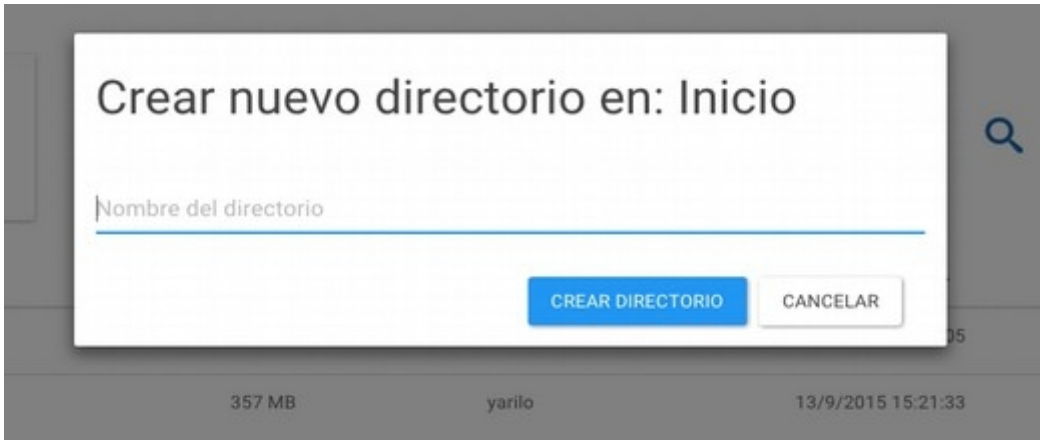


Figura 35: Diálogo crear carpeta

Al introducir el nuevo nombre de carpeta, se enviará la petición al servidor y se creará el nuevo directorio.

Subiendo un archivo

Para subir un archivo el usuario debe hacer click en el icono de la vista principal: 

Tras lo cual se mostrará el siguiente diálogo, en el que usuario puede seleccionar uno o varios archivos para subir, así como pausar o reanudar la subida de los mismos:

Figura 36: Diálogo subir archivo

El proceso completo de una subida está descrito en la sección *8.1.11, Subidas Parciales.*

Moviendo un recurso

Al hacer click en la acción de mover, ya sea en el menú contextual o al examinar un recurso, se presenta el siguiente diálogo, en el que el usuario puede decidir el lugar al que quiere mover el recurso:

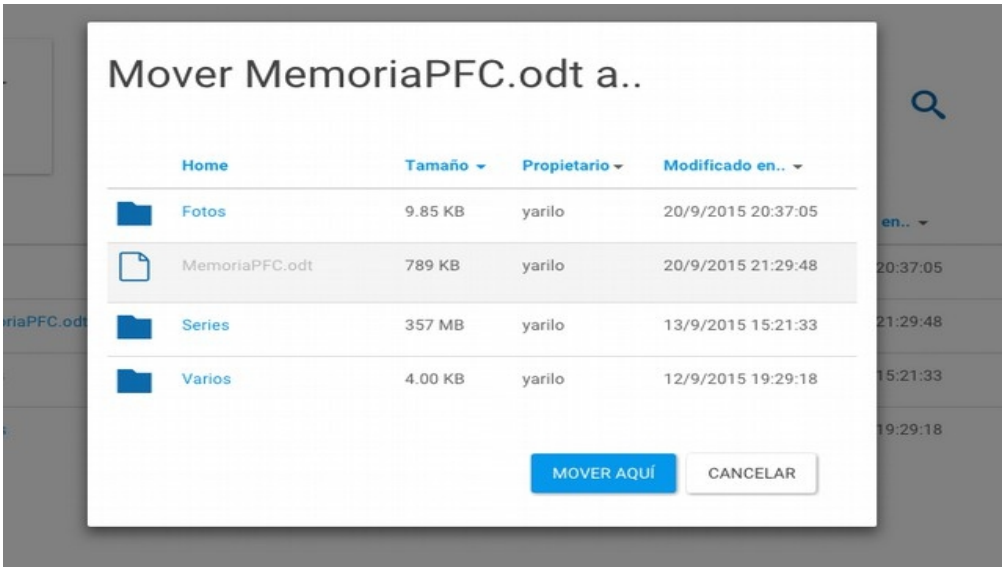


Figura 37: Diálogo Mover recurso

Borrando archivos

Al intentar borrar un recurso, se pide confirmación al usuario:

Figura 38: Diálogo confirmación borrado

Si el usuario decide seguir adelante, el recurso es borrado permanentemente del servidor.

Compartiendo recursos

Existen dos formas de compartir recursos: Otorgando permisos sobre el mismo a usuarios de la aplicación o de forma pública:

Figura 39: Diálogo compartir recurso

Si el usuario hace click en “Compartir”, el recurso será compartido con los permisos establecidos al usuario elegido.

Figura 40: Diálogo de enlace público

Si por el contrario se genera un enlace público el archivo será público para aquellos que tengan el enlace y se mostrará el siguiente diálogo:

Encriptando/desencriptando archivos

La interfaz web permite encriptar o desencriptar archivos mediante el menú contextual o la vista detalles. Aquellos archivos encriptados serán mostrado de distinta forma, como se puede ver a continuación:

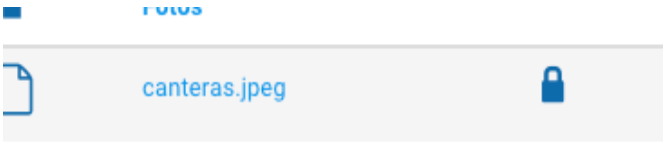


Figura 41: Ejemplo archivo encriptado

Cabe destacar que sólo se pueden encriptar archivos (nunca carpetas) de los que el usuario es dueño.

Generando un torrent/magnet

Una forma distinta de compartir archivos es mediante la generación de torrents o enlaces magnet:

Figura 42: Diálogo torrent/magnet

Si se decide crear un torrent:

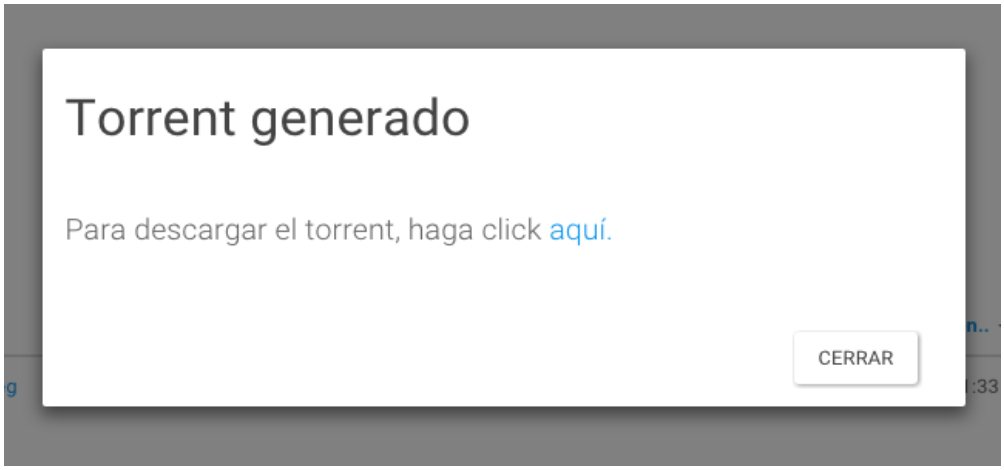


Figura 43: Diálogo torrent generado

Si por el contrario se decide generar un magnet:

Figura 44: Diálogo magnet creado

En cualquier caso, sólo se puede generar torrents o magnet de archivos (no de carpetas) de los que sé es dueño.

Vista de configuración

Al hacer click en el icono de la vista principal: 

Se muestra la siguiente vista:

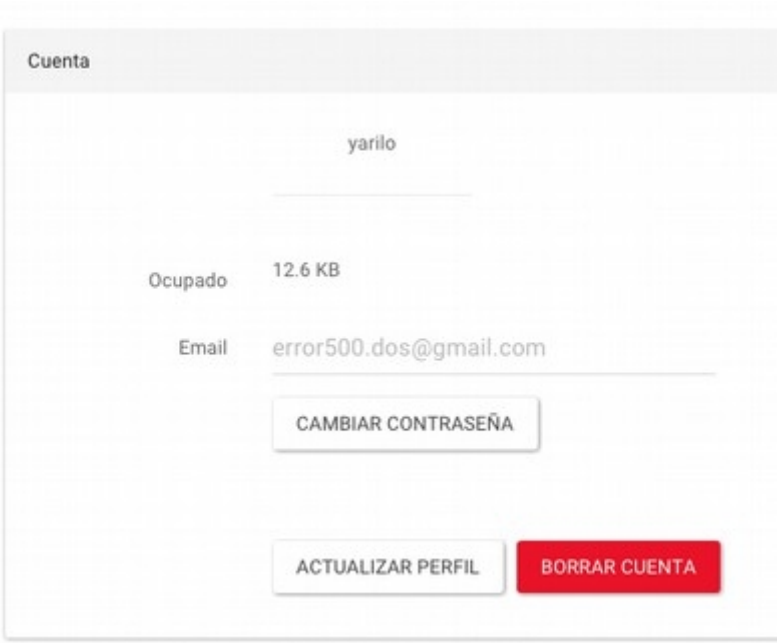


Figura 45: Vista configuración

Como puede observarse, en esta vista el usuario puede modificar su correo, contraseña o incluso borrar su cuenta.

Si el usuario decide esto último, se borrarán irreversiblemente todos sus recursos así como todas las carpetas de las que es dueño.

8.3 Cliente escritorio

En este apartado se describirá el cliente desarrollado para funcionar con el servidor Nginx/Node.js que hemos implementado.

8.3.1 Visión general

El cliente ha sido desarrollado utilizando el Framework Electron, que permite utilizar código Javascript y escribir aplicaciones Node.js para el escritorio, facilitando la portabilidad del mismo código a varias plataformas.

El cliente desarrollado tiene las siguientes características:

- **Acoplamiento con el servidor Nginx/Node.js**

  Utilizando el mismo protocolo HTTP que en el servidor, el cliente es capaz de transferir recursos al mismo.

- **Sincronización**

  Mantiene la carpeta del usuario en su ordenador local y la del servidor totalmente sincronizadas.

- **Resolución de conflictos**

  Si varios usuarios están modificando un archivo, el cliente guardará todas las copias del mismo para evitar una pérdida de los datos.

- **Multiplataforma**

  El cliente está disponible para los sistemas operativos Windows, Mac OS X y GNU/Linux.

8.3.2 Arquitectura

El cliente desarrollado es, usando las propiedades de Electron, una aplicación web realizada en HTML/CSS y Node.js.

La siguiente figura muestra un esquema de la arquitectura general del cliente:

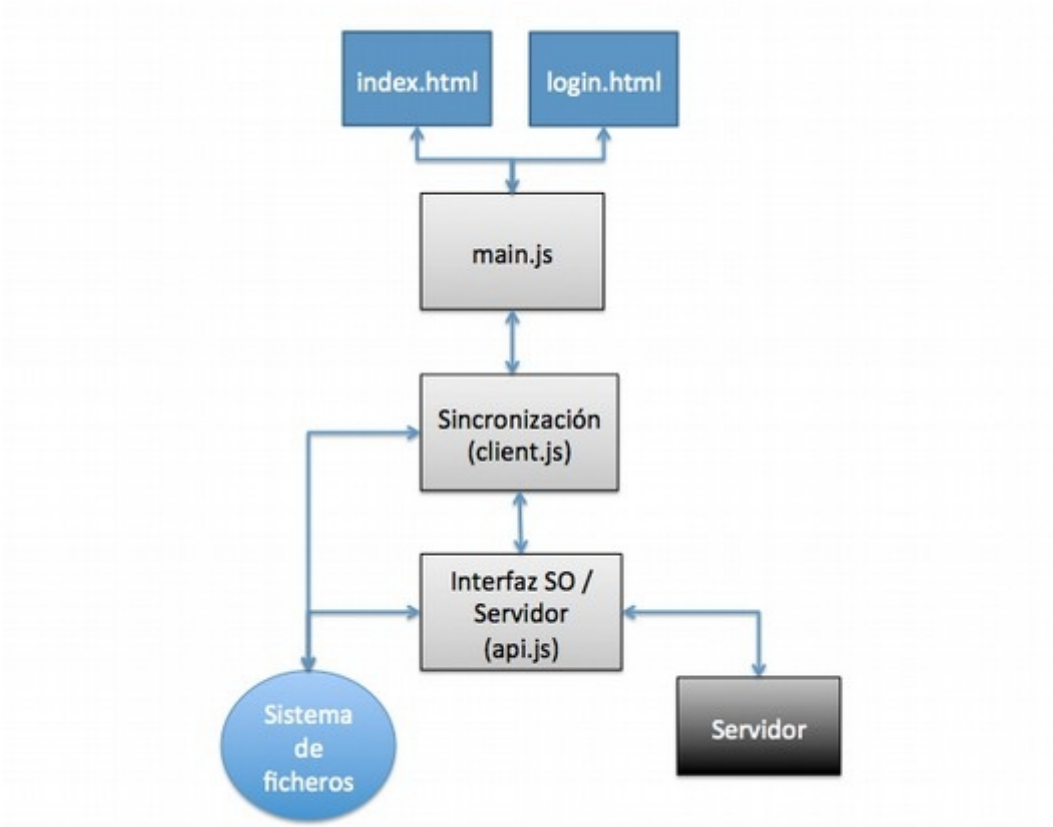


Figura 46: Arquitectura general cliente

De esta forma, el cliente consta de 4 partes diferenciadas:

- **Interfaz de usuario**: Creada en HTML y CSS y formada por los archivos *index.html* y *login.html*
- **Archivo principal (*main.js*)**: Es el punto de ejecución principal de la aplicación, se encarga de mostrar la interfaz y comenzar o parar la sincronización. Su código se
- **Sincronización**: Realizada por el archivo *client.js,* que inspecciona cambios en el sistema de ficheros y en el servidor.
- **Interfaz SO / Servidor**: En el archivo *api.js* se han definido una serie de llamadas (una interfaz), tanto hacia el sistema de ficheros como hacia el servidor web. Estas llamadas son ejecutadas únicamente por *client.js* cuando detecta cambios en la carpeta sincronizada.

En las siguientes secciones se describirán más detalladamente cada una de estas partes.

8.3.3 Interfaz de usuario

La interfaz de usuario se compone de los archivos *login.html* e *index.html*. Adicionalmente se ha usado CSS y el framework Bootstrap para que la apariencia del cliente de escritorio sea similar a la del cliente web, manteniendo la coherencia visual en toda la plataforma.

A continuación se expondrán las distintas secciones y vistas que conforman dicha interfaz.

Vista inicial - Primera ejecución

La primera vez que se ejecuta la aplicación, el usuario verá la siguiente pantalla:

Figura 47: Vista inicial - Primera ejecución cliente

Cómo se puede observar, el usuario tiene que rellenar su nombre de usuario, contraseña, dirección del servidor y carpeta a sincronizar.

Estas variables son guardadas como variables de entorno controladas por la librería estándar *process* [Node.js, 2015] de Node.js. Una vez que el usuario introduce estas variables, se realiza una primera conexión al servidor y se intenta autenticar al mismo. Si la autenticación es exitosa, se guardan las variables en el archivo *settings,* se le abre al usuario la vista de preferencias (explicada a continuación) así como se crea un icono en la bandeja del sistema.

Vista preferencias

La vista de preferencias se muestra al usuario cuando se autentica correctamente o cuando es ejecutada desde el icono de la bandeja del sistema (ver siguiente sección). En la siguiente figura se aprecia el estado inicial de la vista

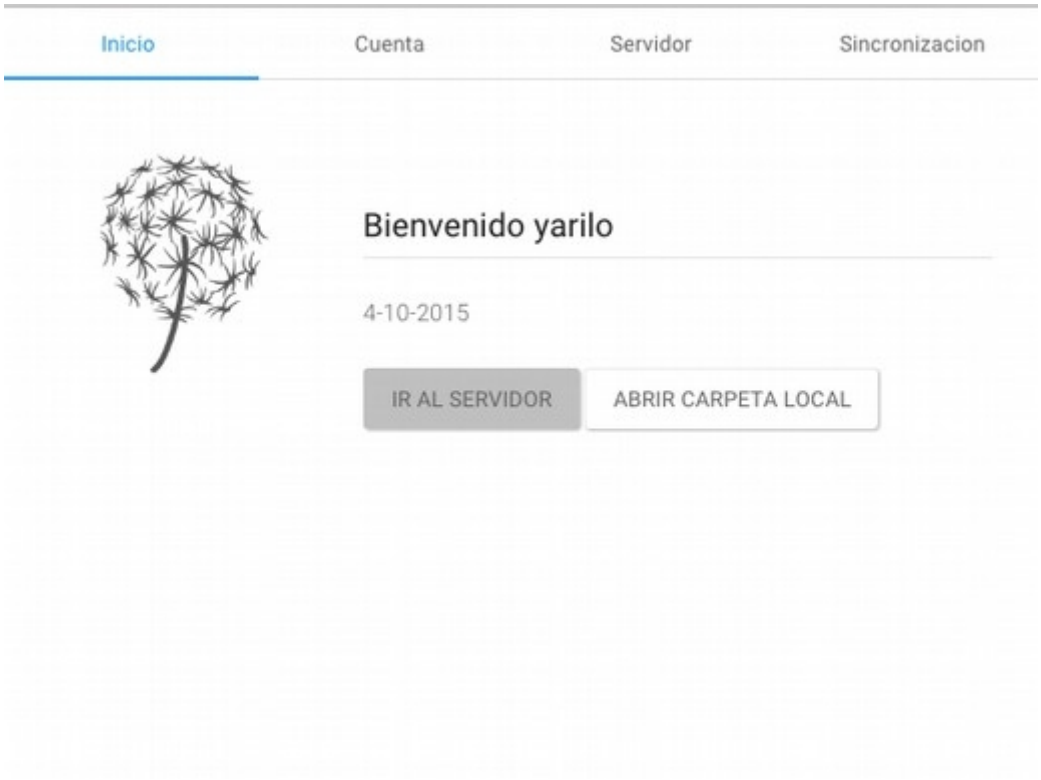


Figura 48: Vista configuración - Inicio

Obsérvese como, además de la vista inicial en la que el usuario puede ir al servidor o abrir su carpeta local, las preferencias están formadas por tres pestañas más, a saber:

- **Cuenta**: Donde el usuario puede cambiar el usuario y contraseña usados

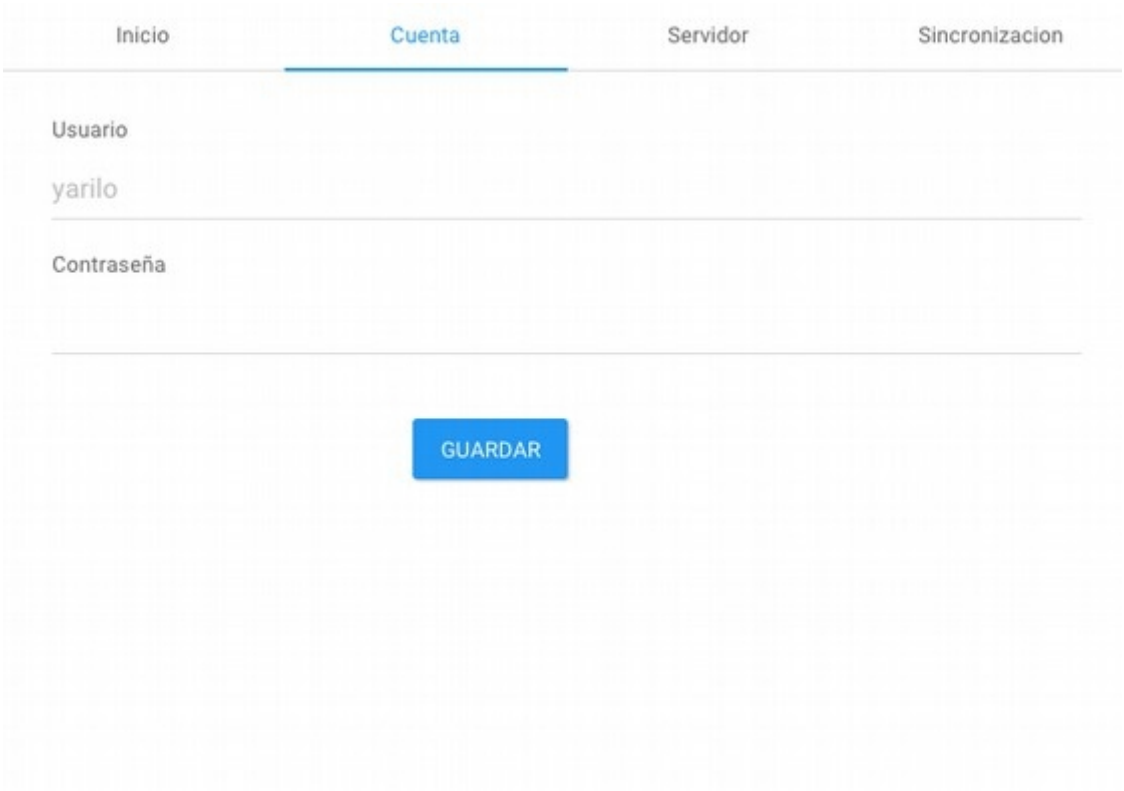


Figura 49: Vista preferencias - Cuenta

- **Servidor:** Donde se cambian los aspectos básicos de conexión hacia el servidor

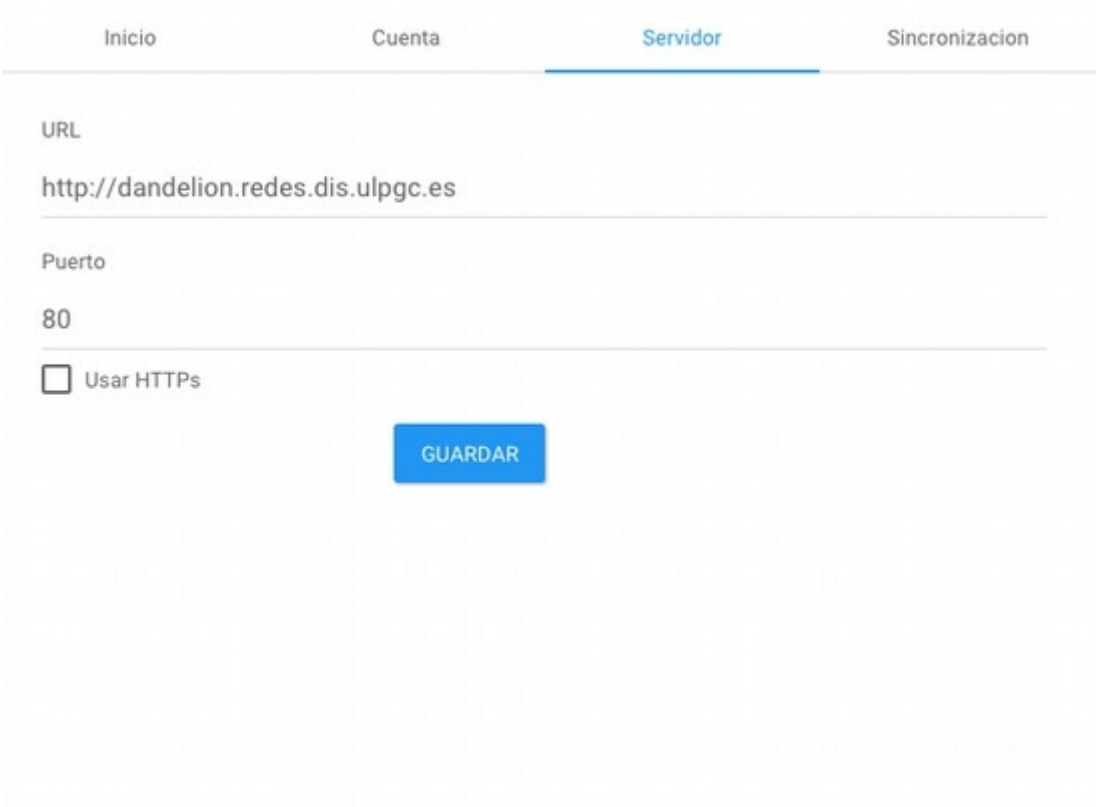


Figura 50: Vista preferencias - Servidor

- **Sincronización:** Donde se puede cambiar la frecuencia de sincronización y la carpeta sincronizada

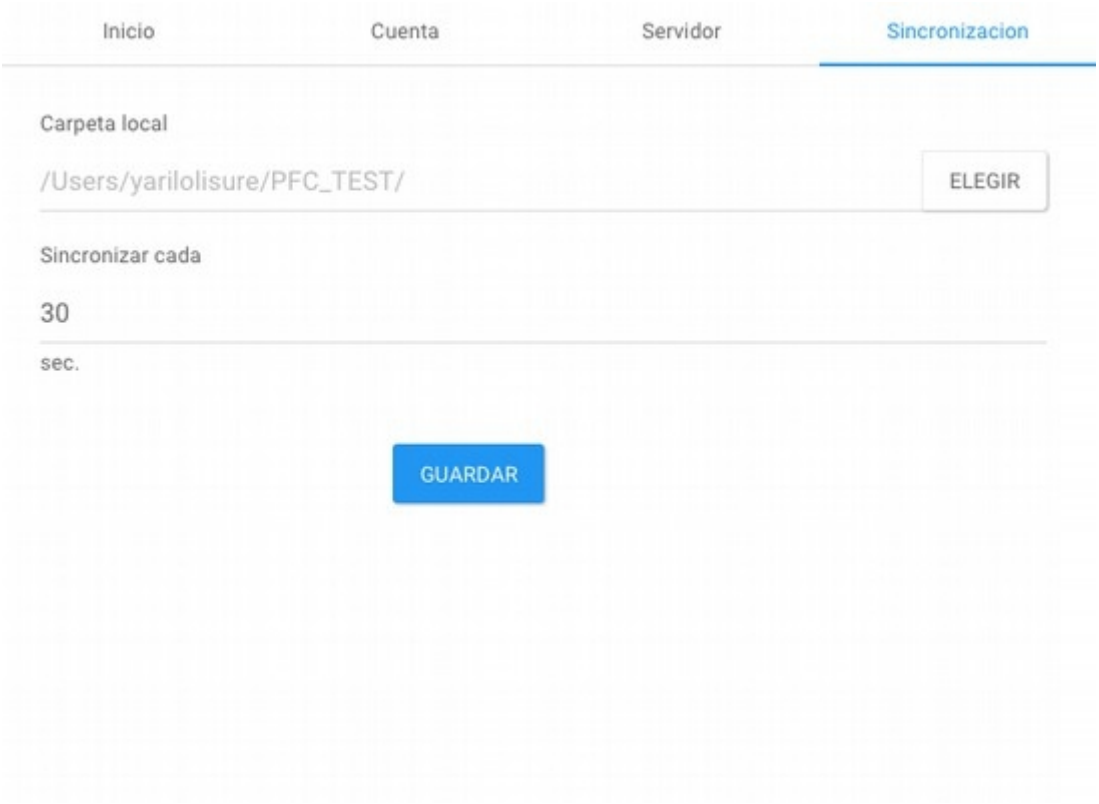


Figura 51: Vista preferencias - Sincronización

Icono de bandeja del sistema

Cuando se inicia el cliente, se crea un icono en la bandeja del sistema que tendrá el siguiente aspecto:

Figura 52: Icono de bandeja del cliente de escritorio

Si se hace click en el mismo, se abrirá el siguiente menú:

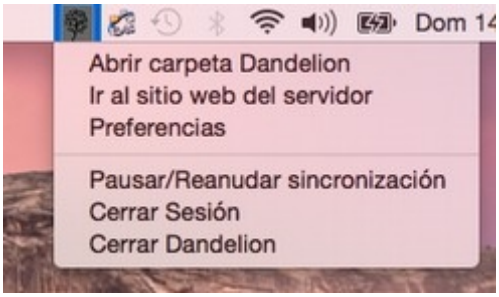


Figura 53: Icono de bandeja del cliente de escritorio - Menú

Nótese la diferencia entre cerrar sesión y cerrar la aplicación ("Dandelion"), ya que la aplicación continuará siempre ejecutándose en segundo plano a no ser que se salga explícitamente de la misma en este diálogo.

8.3.4 Sincronización

El algoritmo de sincronización, escrito en el archivo *client.js,* es el aspecto clave del cliente de escritorio. Ha sido desarrollado en base a los siguientes pilares:

- ***Sync run***

  O simplemente "sincronización", define el momento en el que el cliente y el servidor intercambian información para mantener el mismo estado en la carpeta local y remota. La frecuencia con la que esto ocurre es definida por el usuario en la interfaz y controlada por un *timer interno.*

- ***Watcher* del sistema de ficheros:**

  Un servicio que monitoriza cambios en el sistema de ficheros local e informa de los mismos al cliente. Implementado usando *chokidar* [Miller, 2015].

- **Cola de tareas pendientes sobre el sistema de ficheros**:

  Creada por el *watcher*. Dado que en cualquier momento el usuario puede realizar un cambio sobre el sistema de ficheros, se mantiene una cola de tareas que se enviarán al servidor una vez se inicie la sincronización.

- **Lista de archivos servidor**:

  Un array de objetos, en el que cada objeto tiene el formato indicado en *8.1.6 Representación de*

datos. Obtenida, al igual que con el cliente web, haciendo una llamada HTTP a */api/state.*

- **Lista de archivos clientes:**

  Con el mismo formato que la lista obtenida por el servidor, y generada por la misma librería (*readdirp)*. Esta lista es guardada en el sistema de ficheros en formato JSON guardando el estado de los recursos del cliente cuando se sale de la aplicación.

De esta forma, el algoritmo creado es el que se observa en el siguiente trozo de pseudocódigo:

```
comprobarCambiosOffline();
while (true){ //"Sync run" ejecutado con la frecuencia indicada en las preferencias
        pararTimer();
        aplicarCambiosenServidor(tareasPendientes);
        var listaServidor = obtenerListaServidor();
        var listalocal = obtenerListaLocal();
        var listaDiffServidor = compararListas(listaServidor, listaLocal);
        var listaDiffCliente = compararListas(listaLocal, listaServidor);
        var listaFinal = obtenerListaFinal(listaDiffServidor, listaDiffCliente);
        aplicarCambiosenSistemaDeFicheros(listaFinal);
        actualizarListaLocal();
        reanudarTimer();
}
```

La explicación de este algoritmo, paso a paso, es la siguiente:

0. **Comprobar cambios offline**

   Si se han realizado cambios mientras que el cliente estaba activo, se detectarán leyendo la última lista JSON generada y creando una nueva en este momento. Si hay cambios, éstos se subirán al servidor.

1. **Parar *timer***

   Para evitar que dos sincronizaciones se solapen.

2. **Aplicar tareas pendientes del sistema de ficheros en el servidor**

   En base a la lista de tareas creadas por el *watcher*.

3. **Obtener lista de recursos del servidor**

   Realizando una llamada HTTP a */api/state*

4. **Generar lista de recursos local**

   Examinando los recursos actuales en ese momento y guardando el resultado en un archivo JSON.

5. **Comparar listas desde el punto de vista del servidor**

Generando una lista con la diferencia de los cambios.

6. **Comparar listas desde el punto de vista del cliente**

   Generando una lista con la diferencia de los cambios.

7. **Examinar conflictos**

   Si un recurso está en las dos listas generadas en los pasos **5.** y **6.** quiere decir que puede haber un conflicto y que tanto cliente como servidor quieren realizar modificaciones sobre el mismo.

   Para determinar si el conflicto es real, el algoritmo comprueba que ese recurso no esté en la lista de tareas pendientes (lo cual significaría que el cliente quiere actualizarlo):

   - Si está en la lista de tareas pendientes, ignoramos este recurso por ahora, será actualizado en el paso **1.** en la próxima sincronización
   - Si no está, guardamos la acción que quiere realizar el servidor y creamos una copia de la versión del cliente (ver *Estrategia de Resolución de Conflictos*)

8. **Se aplican los cambios del servidor en el sistema de ficheros**
9. **Se actualiza la lista local**
10. **Se reanuda el *timer***

    Cuando el *timer* lo indique, el programa volverá al paso 1.

Es importante destacar que estos pasos se ejecutan secuencialmente, no empezando un paso hasta la finalización del anterior.

Estrategia de resolución de conflictos

Se define un *"conflicto"* como un archivo o carpeta que ha sido modificado por el servidor y el cliente en el periodo comprendido entre dos sincronizaciones. Cuando ocurre esta situación, el cliente crea una copia del archivo y le añade el nombre "copia en conflicto – fecha – hora", como se observa en:

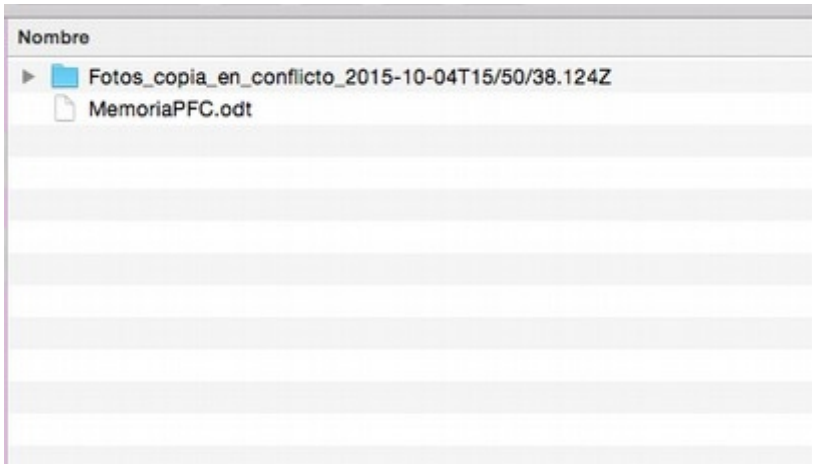


Figura 54: Ejemplo copia en conflicto cliente de escritorio

Esta estrategia evita pérdidas involuntarias de información y deja que el usuario sea el encargado de resolver el conflicto. Si bien se podría utilizar la fecha de modificación más reciente para resolver el conflicto, esta fecha no tiene porque estar sincronizada entre varios ordenadores (en el caso de una carpeta compartida) o con el servidor. En el hipotético caso de que la fecha estuviese sincronizada, las últimas modificaciones hechas por el servidor pueden no ser las deseadas por el usuario, que de otra forma sufriría posibles pérdidas de datos.

Esta estrategia se ha inspirado en la utilizada por Drobpox para resolver conflictos [Dropbox,2009]

Interfaz SO / Servidor

Para realizar los cambios detectados por el algoritmo de sincronización, se han definido una serie de llamadas al sistema operativo y al servidor. Estas llamadas se encuentra en el archivo *api.js* y se han realizado a partir de la librería *request* [Request, 2015] y la librería *fs-extra* [Richardson, 2015] para llamadas HTTP o sobre el sistema de ficheros, respectivamente. Cada una de estas llamadas tiene una *callback* que es ejecutada cuando la operación finaliza, notificando al algoritmo de sincronización de la finalización de la misma.

La lista completa de llamadas se puede encontrar en el *Anexo 13.8 Interfaz SO/Servidor cliente de escritorio*

8.3.5 Distribución e instalación

A la hora de distribuir la aplicación, se han creado paquetes e instaladores específicos para cada arquitectura mediante los siguientes pasos:

1. Empaquetación del código fuente del cliente.
2. Creación de un instalador o ejecutable para cada plataforma.
3. Ejecución del instalador o ejecutable.

Para el primer paso se hace uso del paquete *electron-packager* [Ogden, 2015] creando un paquete mediante el siguiente comando:

```
electron-packager ./ dandelion --out=carpeta_destino --platform=win32/linux/darwin --arch=x64 --version=0.33.6 --asar=true --icon=assets/tray_logo.ico"
```

Indicando la plataforma para que la cual queremos empaquetar el código fuente (win32, linux o darwin) en el parámetro *platform.* Nótese también el uso del parámetro *asar,* en el que indicamos que queremos comprimir el código de nuestra aplicación usando el formato *asar* [Github, 2014]. El uso de este formato, creado por Electron, permite comprimir todo el código en un sólo archivo, así como eliminar errores que pueden aparecer en Windows al usar rutas largas [Microsoft, 2013], situación que puede darse con frecuencia si la aplicación usada tiene muchas dependencias.

Una vez ejecutado el comando anterior, se obtendrá una carpeta lista para distribuir en la plataforma indicada. En este proyecto, se han creado tres carpetas bajo la carpeta *dist/* :

- *dandelion_linux_x64*
- *dandelion_win32_x64*
- *dandelion_darwin_x64*

Cada carpeta contiene un ejecutable adaptado a la plataforma en cuestión denominado *dandelion,*

dandelion.exe o dandelion.app (según el sistema operativo). En el caso de windows, la aplicación debe ejecutarse como administrador para poder operar correctamente.

Con el código empaquetado, se ha hecho uso del paquete *electron-builder* y los siguientes comandos para crear los instaladores pertinentes:

electron-builder dist/osx/dandelion-win32-x64/ --platform=win --out=dist/osx/ --config=config.json

electron-builder dist/win/dandelion-darwin-x64/ --platform=osx --out=dist/win/ --config=config.json

Los instaladores se encuentran bajo */dist/osx* o *dist/win*.

En las siguientes imágenes se exponen figuras del instalador en Windows y Mac OS X:

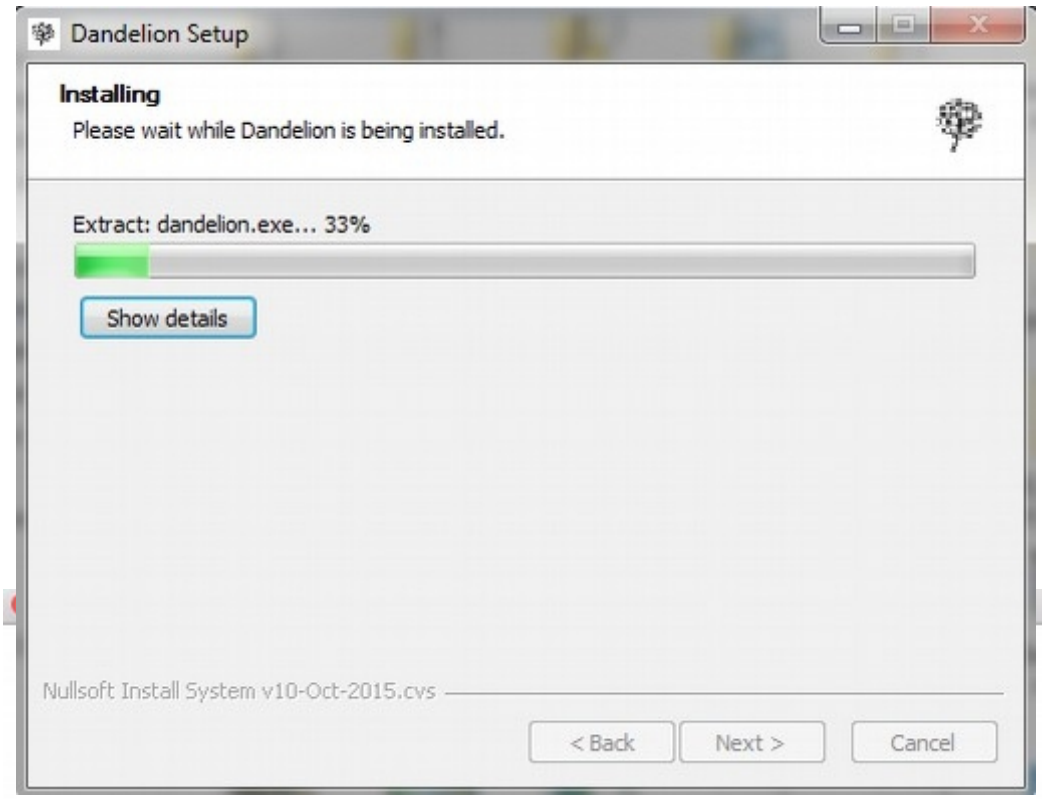


Figura 55: Instalador Windows

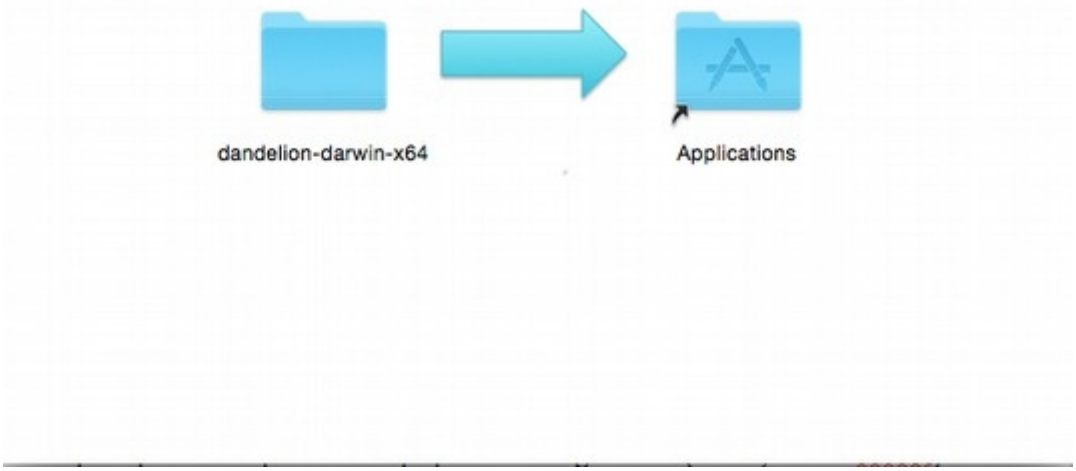


Figura 56: Instalador Mac OS X

Nótese que, ante la gran cantidad de distribuciones existentes, no se ha creado instalador para Linux , pudiéndose abrir la aplicación directamente ejecutando el archivo *dist/dandelion-linux-x64/dandelion*

Finalmente, y tal como se explica en el *Anexo 13.3*, se han guardado los comandos anteriores en el *package.json*. De esta forma se puede resumir todo el proceso anterior haciendo uso de NPM en una sóla instrucción:

```
npm run build
```

9 . Pruebas

En esta sección se presentan las principales pruebas realizadas a la solución implementada. Dividiéndose las mismas en pruebas a la base de datos, componente clave de la plataforma, y pruebas de rendimiento.

9.1 Pruebas de la base de datos

Las operaciones más comunes e importantes de la base de datos, como pueden ser crear un usuario o un recurso, se han sometido a una serie de tests unitarios. Estos tests no sólo garantizan el correcto funcionamiento de cada operación, sino que se aseguran de la consistencia de la base de datos (como pudieran ser documentos incorrectas o duplicados). Los tests, que se encuentran en el fichero

server/util/mongodb.js, pueden ser ejecutados de la siguiente forma:

```
node server/util/mongodb.js
npm start
```

Los tests completos, así como la salida de los mismos, se encuentran en el Anexo *13.6 Tests de la base de datos.*

9.2 Pruebas de rendimiento

Para comprobar la respuesta del servidor ante un escenario real, se le han pasado una serie de pruebas de stress y analizado su comportamiento.

Para ello se ha utilizado la herramienta de benchmarking del servidor Apache , comúnmente conocida como *ab* [Apache, 2015]. Esta herramienta permite especificar un número de peticiones a realizar al servidor, así como un número de usuarios que se conectarían de manera concurrente al mismo, p. ej:

```
ab -n 200 -c 5  http://server_address/
```

Indica que se realizaran 200 peticiones GET con 5 usuarios a la vez (es decir, 200 peticiones de 5 en 5).

En nuestro caso hemos utilizado la siguiente línea de ab:

```
ab -n X -c Y  http://127.0.0.1/?api/resources/test.jpg?user=test&password=test
```

Obteniendo una imagen de 168 bytes para cade petición, y modificando la X (número de peticiones) y la Y (peticiones concurrentes) cómo se puede apreciar en los resultados que se muestran a continuación.

Peticiones no concurrentes

Número de peticiones	Concurrencia	Correctas (%)	Tiempo medio/petición (ms)	Peticiones / segundo	Tiempo total (s)
100	1	100 %	79,299	12,61	7,930
500	1	100 %	79,247	12,62	39,623
1000	1	100 %	79,395	12,60	79,935
10000	1	100 %	78,413	12,75	784,125

Tabla 18: Peticiones no concurrentes vs rendimiento servidor

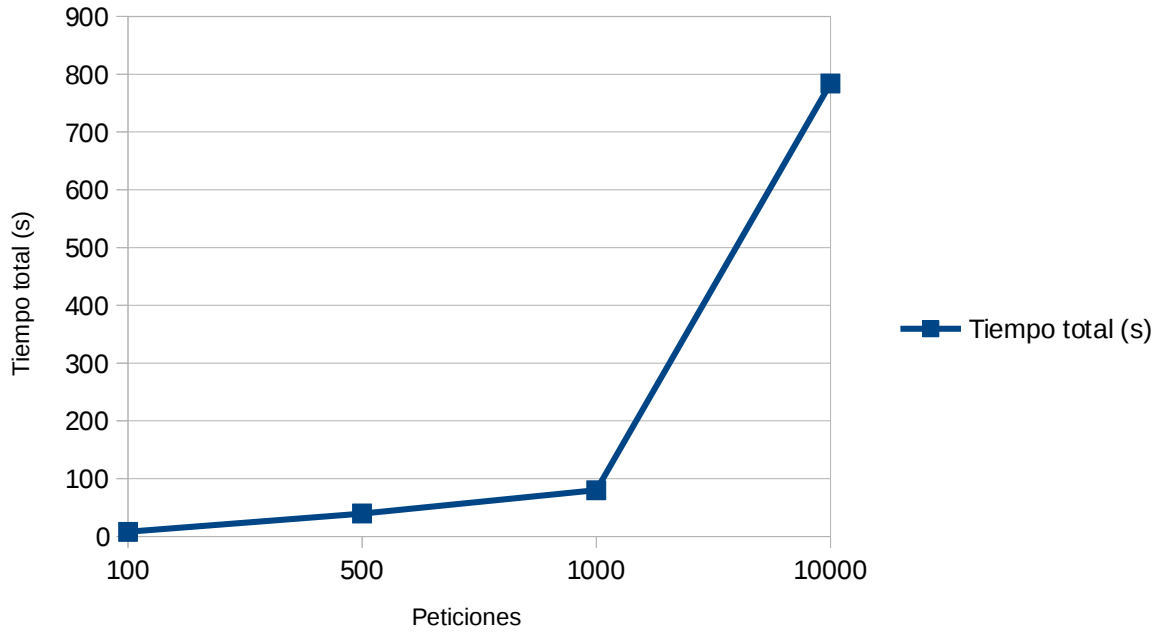


Figura 57: Peticiones no concurrentes vs Tiempo total

Podemos observar como el tiempo total aumenta a medida que aumentamos el número de peticiones (como es lógico), colocándose en 784,125 segundos (13, 07 minutos).

Para el resto de métricas:

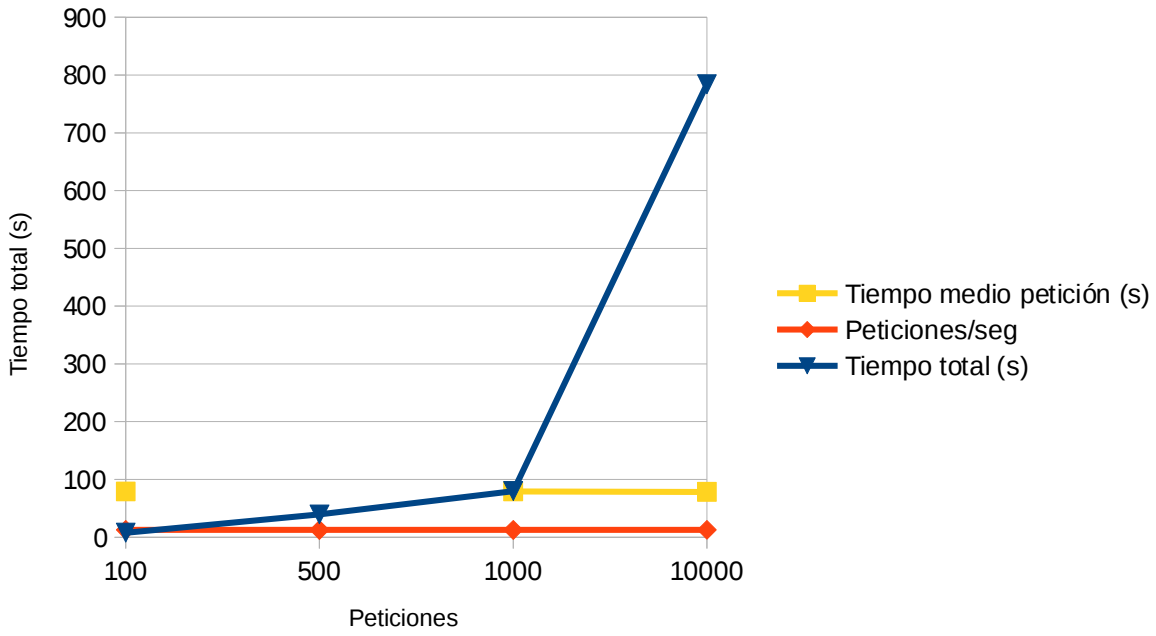


Figura 58: Peticiones no concurrentes vs tiempo medio, peticiones/seg y y tiempo total

Se aprecia que, tanto las peticiones por segundo como el tiempo medio por petición se mantienen independientemente de las peticiones (en torno a las 12 peticiones/seg y 78-79 ms, respectivamente), lo cual nos indica que el rendimiento medio del servidor no varía significativamente en función de las peticiones.

Peticiones concurrentes

Número de peticiones	Concurrencia	Correctas (%)	Tiempo medio/petición (ms)	Peticiones /segundo	Tiempo total (s)
10000	100	100 %	23,226	43,06	232,256
10000	500	100 %	24,233	41,27	242, 330
10000	800	100%	24,208	41,31	242,084
10000	1000	100 %	23,618	42,34	236,176

Tabla 19: Peticiones concurrentes vs rendimiento servidor

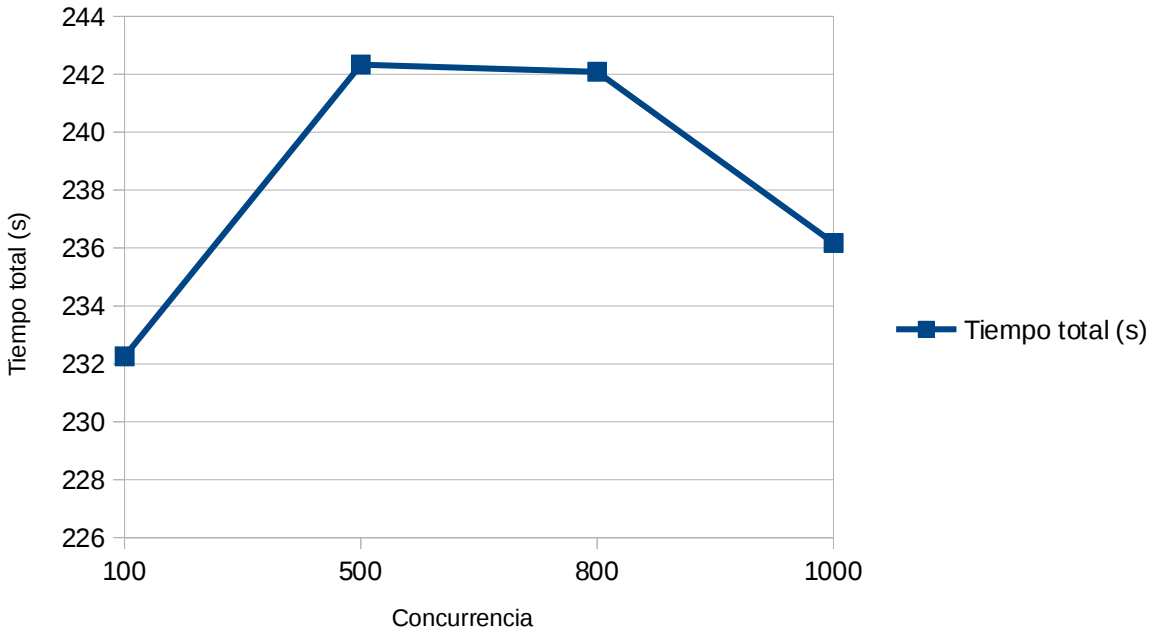


Figura 59: 10000 peticiones con concurrencia vs tiempo total

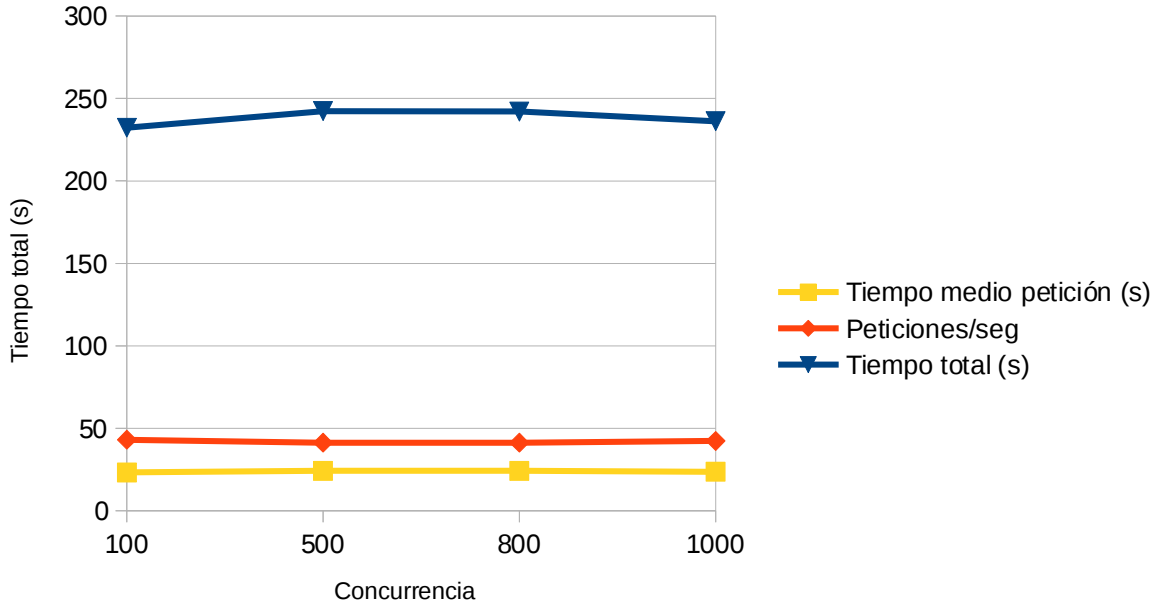


Figura 60: Peticiones concurrentes vs tiempo medio, peticiones/seg y y tiempo total

Observando la primera gráfica, se observa que independientemente de la concurrencia el tiempo total para la ejecución de 10000 peticiones se mantiene en torno a 232 – 242 segundos, mejorando incluso los tiempos sin concurrencia (784,125 segundos). Ello se debe a que, al aumentar el número de peticiones bloqueantes (por ejemplo al acceder a la BD para preguntar sobre la autenticación, o al acceder a disco), Node y Nginx son capaces de distribuir equitativamente la carga entre las mismas.

La misma conclusión obtenemos mirando la segunda gráfica, en la que tanto el tiempo por petición como el número de peticiones por segundo no varía en exceso en función de la concurrencia.

9.2.1 Comparación con servidor prototipo en Python

Cómo se ha indicado en la sección de planificación, en un primer momento se desarrolló un prototipo del servidor web en Python.La arquitectura de dicho prototipo estaba basada en hilos, creando un hilo nuevo para cada petición. Como se verá en las siguientes tablas y gráficas, dicho prototipo quedó descartado por su pobre rendimiento.

Peticiones no concurrentes

Número de peticiones	Concurrencia	Correctas (%)	Tiempo medio/petición (ms)	Peticiones /segundo	Tiempo total (s)
100	1	100 %	44,779	22,33	4,478
500	1	100 %	43,852	22,8	21,296
1000	1	100 %	44,364	22,54	44,364
1000	1	100 %	45,652	21,9	456,524

Tabla 20: Peticiones no concurrentes - Servidor prototipo en Python

Obsérvese que los valores son menores que en el servidor final. Ello se debe a que estas pruebas fueron realizadas con un prototipo del servidor, en el que no se tenían en cuenta los accesos a base de datos o la autenticación.

Peticiones concurrentes

Número de peticiones	Concurrencia	Correctas (%)	Tiempo medio/petición (ms)	Peticiones /segundo	Tiempo total (s)
1000	5	100 %	43,481	23	43,841
1000	10	100 %	45,21	22,12	45,21
1000	20	100 %	44,459	22,49	44,459
1000	30	13,1 %	-	-	

Tabla 21: Peticiones concurrentes - Servidor prototipo en Python

Obsérvese que a partir de las 30 peticiones concurrentes no hay resultados, ello es así porque el servidor daba fallos o se atascaba en cuanto se le pasaba ese número de peticiones.

Gráficas comparativas

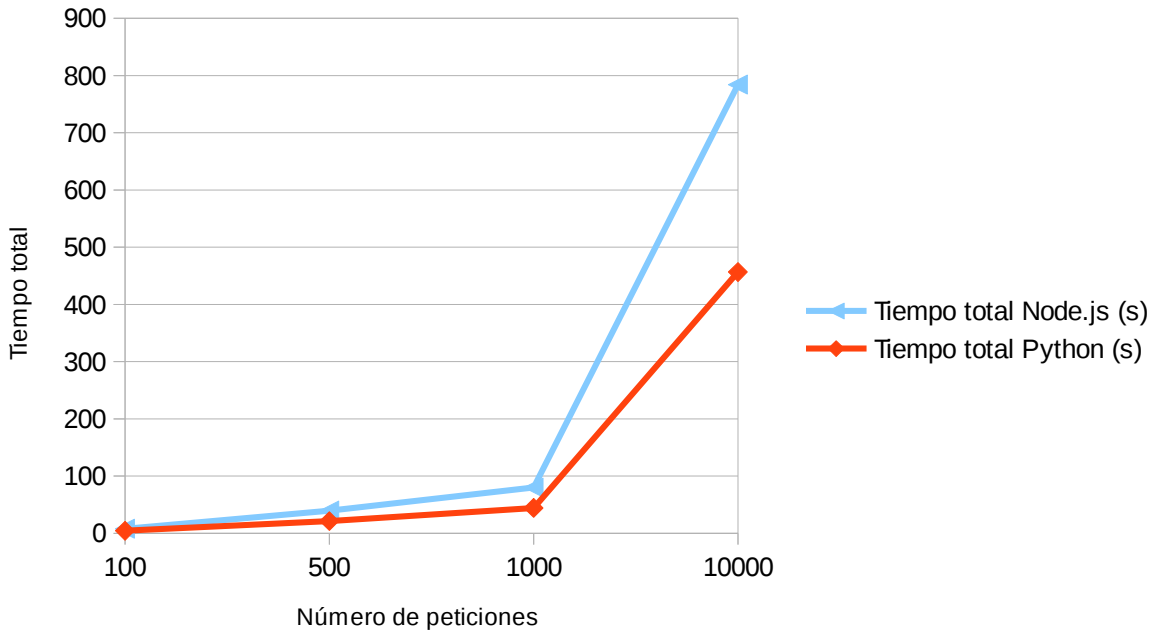


Figura 61: Tiempo total vs número de peticiones – Comparación servidores

Se puede observar que el tiempo total es menor en el caso del servidor Python. Ello se debe, como se ha comentado anteriormente, a que este prototipo no autenticaba las peticiones ni tenía que acceder a la base de datos, al contrario de lo que ocurre con el servidor Node.js/Nginx, cuyo tiempo si incluye estas comprobaciones.

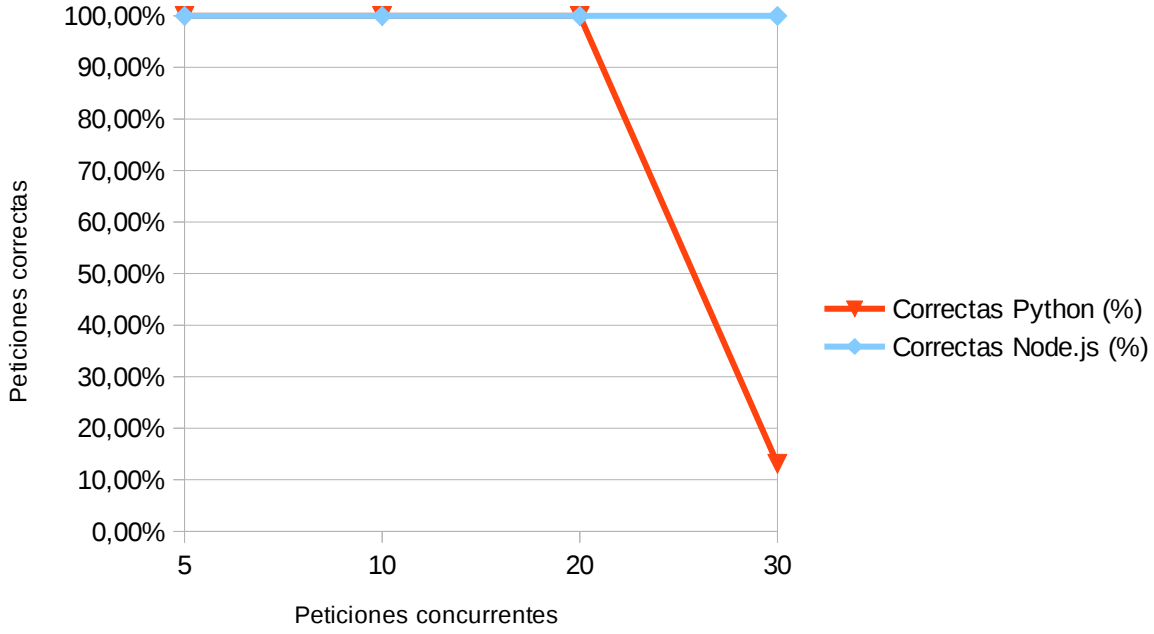


Figura 62: Peticiones correctas vs Peticiones concurrentes - Comparación servidores

Esta gráfica demuestra el mal rendimiento del servidor realizado en Python ante peticiones

concurrentes, no pudiendo superar las 30 peticiones concurrentes sin comenzar a fallar. Por contra, el servidor realizado en Node.js no ofrece ningún tipo de problema, llegando incluso a las 1000 peticiones concurrentes (cómo se puede ver en las tablas de la sección anterior).

La razón de esta diferencia se encuentra en el patrón que sigue cada servidor: Mientras que Python abre un hilo para cada nueva petición y cada hilo queda a la espera (bloqueado) ante operaciones de entrada/salida, Node.js y Nginx se ejecutan en un sólo proceso y no esperan por ninguna petición, pasando a la siguiente disponible lo antes posible. Por otra parte, el modelo del servidor Python provoca un aumento desmesurado del uso de memoria al crear varios hilos para un sólo proceso.

Ante estos datos, en una primera fase del proyecto se ha elegido desechar el prototipo en Python y utilizar Node.js y Nginx, cuyo rendimiento basado en arquitecturas asíncronas o reactivas es superior para el caso de uso de este proyecto.

9.2.2 Conclusiones

Se puede observar que para peticiones no concurrentes el servidor se comporta aceptablemente, quizá ligeramente lento a medida que aumentan las peticiones, problema que puede ser resuelto si el cliente utiliza peticiones condicionales y la caché (no es el caso del programa *ab* en estos tests).

Por otra parte, donde tanto Node como Nginx brillan es el manejo de la concurrencia, donde podemos ver que el servidor se maneja sin problema hasta llegar a 1000 peticiones concurrentes, y manteniendo en todo momento el mismo tiempo de respuesta, lo cual indica la facilidad del servidor para escalar ante un mayor número de conexiones concurrentes, escenario habitual en aplicaciones web como la que se presenta.

Finalmente resaltar que tanto Node como Nginx pueden ser configurados para manejar un volumen mucho mayor que el presentado en estos tests, pudiendo por ejemplo añadir más aplicaciones Node para que Nginx distribuya las peticiones entre las mismas o simplemente ajustando parámetros en la configuración de ambos. Dicha optimización escapa al alcance de este proyecto, reservándose para un hipotético Trabajo Futuro (ver sección *12. Trabajo Futuro*)

10 . Costes

A continuación se desglosan los distintos costes económicos que conforman la elaboración de este proyecto fin de carrera.

Costes de hardware

Descripción	Coste
Ordenador Portátil Asus K53SV	685 €
Ordenador Portátil MacBook Pro	1.178,99 €
Total	1863,99

Tabla 22: Costes de hardware

Costes de despliegue servidor

Descripción	Coste
Dominio ULPGC	0 €
Instancia EC2 Micro	0 € (primer año de uso)
Total	0,00 €

Tabla 23: Costes de despliegue servidor

Costes software

Descripción	Coste
Libre office	0 €
Editor Atom	0 €
Cuenta mailgun	0 € (hasta 10.000 correos por mes)
Licencia sistema operativo Mac OS/X	0 €
Licencia Sistema operativo Windows	135 €
Licencia Sistema operativo GNU/Linux	0 €
Total	135 €

Tabla 24: Costes de software

Costes de desarrollo y recursos humanos

Teniendo en cuenta que el sueldo bruto mínimo de un ingeniero de software es de 7,79 € por hora [WageIndicator, 2015], podemos establecer los siguientes costes de desarrollo:

Descripción	Horas	Total
Desarrollo servidor	411	3201,69 €
Desarrollo cliente web	280	2181,2 €
Desarrollo cliente escritorio	205	1596,95 €
Total	896	6979,84 €

Tabla 25: Costes de desarrollo y recursos humanos

Total

Descripción	Coste
Costes hardware	1863,99 €
Costes software	135 €
Costes de despliegue servidor	0 €
Costes de implementación y recursos humanos	6901,94 €
Total coste proyecto	8900, 93 €

Tabla 26: Costes totales

11 . Conclusiones

En la realización de este trabajo fin de grado hemos llevado a cabo el desarrollo de una solución que permita el dar servicios de almacenamiento en red desde nuestros servidores compartiendo estos datos entre los usuarios.

Para desarrollar el sistema hemos desarrollado se ha desarrollado un sistema de gestión de usuarios (objetivo 1 PFC-1) mediante un sistema de solicitud de cuenta y confirmación a través del correo electrónico. Así mismo también se desarrolló un sistema de recuperación de contraseñas. Cada usuario tiene asociado una zona de almacenamiento en el servidor (objetivo 2) que se comporta como un repositorio de gestión personal, donde puede almacenar sus archivos y descargarlo mediante acceso remoto (objetivo 3). Estos datos pueden ser cifrados por el usuario proporcionando a los usuarios una herramienta que garantice la privacidad de sus datos (objetivo 4). Los usuarios pueden gestionar sus operaciones con un cliente web, acceso mediante navegador al servidor, o a través de un cliente de escritorio, comportándose como una unidad local. Las distintas versiones de cualquier documento que se encuentre en el sistema de almacenamiento, unidades locales o documento compartidos se encontrará sincronizados (objetivos 5 y 6). Las descarga de datos se puede realizar mediante enlaces p2p y la utilización de clientes torrent o magnet (objetivo 7). Los clientes de escritorio se han desarrollado para su utilización en plataformas de sobremesa (objetivo 8)

En definitiva, se ha conseguido realizar una solución integral de almacenamiento en la nube abierta y que a su vez ceda el control de los datos al usuario de la misma, respetando su privacidad.

Desde el punto de vista puramente técnico, se ha demostrado que las arquitecturas asíncronas (como aquellas que siguen el patrón reactor) son idóneas para aplicaciones con una gran carga de peticiones de entrada/salida, como es el caso del presente proyecto. En la misma línea, se ha demostrado que una solución basada en hilos no es adecuada para este caso de uso, teniendo problemas para escalar correctamente ante varios usuarios.

Por otra parte, el uso de Javascript en el proyecto ha permitido desarrollar una solución en el mismo lenguaje en todos los niveles de la aplicación (*frontend*, *backend* y cliente de escritorio), lo cual demuestra la madurez del lenguaje a la hora de desarrollar aplicaciones usando tecnologías web. Asimismo, la elección de este lenguaje ha permitido el uso de herramientas web, como puede ser AngularJS o Node.js, que han facilitado y clarificado el desarrollo de la solución implementada.

Finalmente, la realización de una solución que comprende todas las etapas de una aplicación web ha permitido una mayor comprensión y entendimiento las tecnologías que forman cada capa de la aplicación, teniendo que mejorar aspectos tan diversos como la usabilidad del usuario, la representación de los datos en la base de datos o la correcta configuración de los parámetros del servidor. Esta visión y comprensión es muy útil de cara a evaluar otras soluciones reales similares y sin ninguna duda será usada en proyectos futuros.

12 . Trabajo futuro

A continuación se proponen una serie de aspectos que podrían ser aplicados al presente proyecto para mejorar la solución desarrollada:

- **Control de versiones**

  Se podría desarrollar un control de versiones en el servidor, de forma que cada vez que se modifique o borre un recurso, el servidor guarde una copia del mismo y la base de datos guarde las referencias a todas las versiones. Ello permitiría restablecer archivos borrados por error o volver a versiones anteriores de un mismo recurso.

- **Aplicación móvil**

  Si bien el cliente web realizado adapta su contenido a la pantalla del dispositivo que esté viendo la web, se podría desarrollar una aplicación móvil nativa para acceder al servidor y sincronizar archivos del móvil con el resto de la plataforma.

- **Sincronización incremental**

  En la solución implementada cada vez que hay un cambio en un archivo se actualiza el archivo completo en el repositorio. Una mejora sobre este algoritmo sería subir sólo aquellas partes que han cambiado ahorrando ancho de banda y ofreciendo una sincronización más rápida. Para ello habría que utilizar algoritmos *delta* [Mogul, 2002] y de diferencia entre archivos, o *diff* [Hunt, 1976].

- **Optimización servidor**

  Existen multitud de optimizaciones que se le pueden hacer al servidor web, tanto a la aplicación Node.js como al servidor Nginx. Para abordar las mismas, en primer lugar habría que examinar o hacer *profiling* para averiguar qué tareas dificultan el rendimiento servidor, y en segundo lugar aplicar medidas para mejorar esas tareas. Una posible optimización a realizar sería crear una instancia de Node.js para las tareas más costosas en tiempo (como comprimir una carpeta o subir un archivo de forma parcial) y que Nginx distribuya las peticiones a dichas tareas, aliviando la carga del servidor principal.

13 . Anexos

13.1 Lista de dependencias servidor

La lista de dependencias o librerarías necesarias para el funcionamiento del servidor Node.js se encuentra en el archivo *package.json.* Este archivo indica al manejador de paquetes de Node (NPM) las librerías que éste debe instalar o actualizar para que funcione la aplicacón (en este caso el servidor web Node.js) al que hace referencia.

El contenido de este archivo es el siguiente:

```
{
  "name": "dandelion-backend",
  "version": "1.0.0",
  "description": "\"Backend for Dandelion\"",
  "private": true,
  "main": "cluster.js",
  "scripts": {
    "start": "node server/cluster.js",
    "test": "node server/util/mongodb.js"
},

  "author": "Yarilo Villanueva",
  "license": "GNU-GPL",
  "dependencies": {
    "acl": "^0.4.4",
    "async": "^1.4.0",
    "bcrypt": "^0.8.1",
    "body-parser": "^1.13.3",
    "clone": "^1.0.2",
    "connect-multiparty": "^1.2.5",
    "connect-restreamer": "^1.0.2",
    "create-torrent": "^3.9.0",
    "express": "^4.10.2",
    "file-encryptor": "^0.1.0",
    "fs-extra": "^0.23.1",
    "http-proxy": "^1.7.0",
    "jsonwebtoken": "^5.0.5",
    "magnet-link": "^1.0.3",
    "mailgun-js": "^0.7.1",
    "mongodb": "^1.4.22",
    "morgan": "^1.5.0",
    "node-acl": "0.0.2",
```

```
    "nodemailer": "^1.4.0",
    "nodemailer-mailgun-transport": "^1.0.1",
    "passport": "^0.2.1",
    "passport-local": "^1.0.0",
    "prototypes": "^0.3.2",
    "readdirp": "^1.4.0",
    "request": "^2.60.0",
    "tar-fs": "^1.8.1",
    "testing": "^0.2.2",
    "webtorrent": "^0.27.2"
  }
}
```

13.2 Lista de dependencias cliente web

La lista de dependencias o librerías de las que depende el cliente web para su correcto funcionamiento se encuentra en el archivo *bower.json* e indican al gestor *Bower* que librerías son necesarias para el funcionamietno del cliente web. Su contenido es el siguiente:

```
{
  "name": "Dandelion",
  "version": "1.0.0",
  "homepage": "https://github.com/Yarilo/dandelion-backend",
  "authors": [
    "Yarilo <error500.dos@gmail.com>"
  ],
  "license": "GPL",
  "dependencies": {
    "angucomplete-alt": "~1.1.0",
    "angular-route": "~1.3.0",
    "angular-bootstrap": "~0.11.2",
    "ng-context-menu": "~1.0.1",
    "ng-flow": "~2",
    "bootstrap": "~3.3.4",
    "angular-cookies": "~1.4.1"
  },
  "resolutions": {
    "angucomplete-alt": "~1.1.0"
  }
}
```

13.3 Lista de dependencias cliente de escritorio

El cliente de escritorio guarda una lista de sus dependencias en el archivo *package.json* (al igual que el servidor Web). A partir de este archivo, NPM instalará y actualizará convenientemente las dependencias. El contenido de *package.json* es el siguiente:

```
{
  "name": "dandelion-desktop-client",
  "description": "\"Desktop client for Dandelion\"",
  "version": "1.0.0",
  "main": "main.js",
  "author": "Yarilo Villanueva",
```

```
  "license": "GNU-GPL",
  "scripts": {

    "clean": "rm -rf ../dist",
        "clean:linux": "rm -rf ../dist/linux",
    "clean:osx": "rm -rf ../dist/osx",
    "clean:win": "rm -rf ../dist/win",

    "pack": "npm run clean && npm run pack:linux && npm run pack:osx && npm run pack:win",
        "pack:linux": "npm run clean:linux && electron-packager ./ dandelion --out=../dist/linux --platform=linux
--arch=x64 --version=0.33.7 --asar=true --icon=assets/tray_logo.ico",
    "pack:osx": "npm run clean:osx && electron-packager ./ dandelion --out=../dist/osx --platform=darwin --arch=x64
--version=0.33.7 --asar=true --icon=assets/tray_logo.ico",
    "pack:win": "npm run clean:win && electron-packager ./ dandelion --out=../dist/win --platform=win32 --arch=x64
--version=0.33.7 --asar=true --icon=assets/tray_logo.ico",

    "build": "npm run build:osx && npm run build:win",
    "build:osx": "npm run pack:osx && electron-builder ../dist/osx/dandelion-darwin-x64 --platform=osx --out=../dist/osx/
--config=config.json",
    "build:win": "npm run pack:win && electron-builder ../dist/win/dandelion-win32-x64 --platform=win --out=../dist/win/
--config=config.json"
  },
  "dependencies": {
    "async": "^1.4.2",
    "chokidar": "^1.0.3",
    "dotenv": "^1.2.0",
    "electron-builder": "^2.0.1",
    "fs-extra": "^0.24.0",
    "line-reader": "^0.2.4",
    "prototypes": "^0.4.1",
    "readdirp": "^2.0.0",
    "request": "^2.58.0",
    "requests": "^0.1.6",
    "underscore": "^1.8.3",
    "upath": "^0.1.6"
  },
  "devDependencies": {
    "electron-packager": "^5.1.0"
  }
}
```

Cabe destacar la sección *scripts* en la que se definen scripts para que NPM los ejecute de la siguiente forma:

```
npm run nombre_script
```

Se puede observar que se han configurado varios scripts para empaquetar la aplicación (*pack)* así como para crear instaladores (*build*). De esta forma, se pueden crear paquetes de la aplicación ejecutando:

```
npm run pack
```

O instaladores (para Mac OS X y Windows) ejecutando:

```
npm run build
```

13.4 Configuración Nginx

En este anexo se presentan los ficheros de configuración usados a la hora de ajustar los parámetros de Nginx. Los ficheros se encuentran en la carpeta *server/nginx/* y se llaman *nginx.conf* y *dandelion.conf*

nginx.conf

```
user  yarilo staff; #Debe ser el usuario que ejecuta nginx
worker_processes  auto;

events {
   worker_connections  1024;
}


http {
   include       mime.types;
   default_type  application/octet-stream;

   sendfile        on;
   #tcp_nopush     on;

   keepalive_timeout  70;

           include /usr/local/etc/nginx/sites-enabled/*;
}
```

dandelion.conf

```
client_max_body_size 0;
#Caching
proxy_cache_path  /var/cache/nginx levels=1:2 keys_zone=one:8m max_size=3000m inactive=600m;
proxy_temp_path /var/tmp;

#Opciones de compresión
gzip on;
gzip_comp_level 6;
gzip_vary on;
gzip_proxied any;
gzip_buffers 16 8k;

#Dirección aplicació Node.js
upstream dandelion_backend {
           server 127.0.0.1:3000;
}

server {

       server_name localhost;
       listen 80;
             #SSL
       listen 443 ssl default_server;

       ssl_session_cache   shared:SSL:20m;
       ssl_session_timeout 30m;
       ssl_certificate /Users/yarilolisure/PFC/dandelion-backend/server/assets/server.crt;
       ssl_certificate_key /Users/yarilolisure/PFC/dandelion-backend/server/assets/server.key;

       ssl_session_tickets on;

       ssl_protocols       SSLv3 TLSv1 TLSv1.1 TLSv1.2;
       ssl_prefer_server_ciphers on;
       ssl_ciphers ECDH+AESGCM:ECDH+AES256:ECDH+AES128:DH+3DES:!ADH:!AECDH:!MD5;
       ssl_dhparam /Users/yarilolisure/PFC/dandelion-backend/server/assets/dhparam.pem;


       location / { #Ruta para servir archivos de AngularJS
          add_header Access-Control-Allow-Origin "$http_origin";
          add_header Access-Control-Allow-Methods "GET, POST, OPTIONS, PUT, DELETE";
          add_header Access-Control-Allow-Headers 'Authorization,Content-Type,Accept,Origin,User-Agent,DNT,Cache-
Control,X-Mx-ReqToken,Keep-Alive,X-Requested-With,If-Modified-Since';
          add_header Access-Control-Allow-Credentials "true";
          root /Users/yarilolisure/PFC/dandelion-backend/client/app;
          expires max;
```

```
        proxy_cache one;
        proxy_cache_key sfs$request_uri$scheme;
      }

      #Todas las peticiones que contengan /api/ serán redirigidas a la aplicación de Node.js
      location /api {
         add_header Access-Control-Allow-Origin "$http_origin";
         add_header Access-Control-Allow-Methods "GET, POST, OPTIONS, PUT, DELETE";
         add_header Access-Control-Allow-Headers 'Authorization,Content-Type,Accept,Origin,User-Agent,DNT,Cache-
Control,X-Mx-ReqToken,Keep-Alive,X-Requested-With,If-Modified-Since';
         add_header Access-Control-Allow-Credentials "true";
         client_body_temp_path /Users/yarilolisure/PFC/dandelion-backend/data/tmp;
         proxy_redirect off;
         proxy_set_header Host $host;
         proxy_pass_request_body on;
         proxy_pass_request_headers on;
         proxy_pass http://dandelion_backend;
                              proxy_read_timeout 1800s; #30 min
                              proxy_send_timeout 1800s;
      }
          #Ruta de uso interno para descargar, sólo Node.js puede realizar peticiones a esta ruta.
      location /authorizated_download{
              internal;
              alias /Users/yarilolisure/PFC/dandelion-backend/data;
              expires 48h;
              proxy_cache one;
              proxy_cache_key sfs$request_uri$scheme;
              add_header Cache-Control "public";
      }
          #Ruta de uso interno para descargar archivos comprimidos, sólo Node.js puede realizar   #peticiones a esta ruta.
      location /authorizated_compressed{
              internal;
              alias /Users/yarilolisure/PFC/dandelion-backend/data_compressed;
              expires 48h;
              proxy_cache one;
              proxy_cache_key sfs$request_uri$scheme;
              add_header Cache-Control "public";
      }
          #Ruta de uso interno sólo Node.js puede realizar peticiones a esta ruta.
      location /authorizated {
        allow 127.0.0.1;
        deny all;
        alias /Users/yarilolisure/PFC/dandelion-backend/data;
        client_body_temp_path /Users/yarilolisure/PFC/dandelion-backend/data/tmp;
        dav_methods PUT DELETE MKCOL COPY MOVE;
        dav_ext_methods PROPFIND OPTIONS;
        expires 48h;
        proxy_cache one;
        proxy_cache_key sfs$request_uri$scheme;
      }

   }
```

13.5 Scripts de despliegue

Para desplegar rápidamente el servidor y la aplicación web en un entorno Unix, se han desarrollado una serie de scripts de despliegue que se encuentra bajo la carpeta *deployment.* El script principal, y encargado de llamar al resto se denomina *deploy.sh* y su contenido se puede ver a continuación:

```
#!/usr/bin/env bash

cd ..
parentdir=$(pwd)
cd deployment

echo "Please type the system you are running this script (1/2/3)"
```

```
echo "1) RedHat/CentOS  2)Ubuntu  3)Mac OS X"
read system

#Linux
if [ $system -eq 1 ]
then
   sh "deploy-red-hat.sh"
#Ubuntu
elif [ $system -eq 2 ]
then
   sh "deploy-ubuntu.sh"
#Mac OSX
elif [ $system -eq 3 ]
then
   sh "deploy-mac.sh"
fi

##### Common Setup ######

#Nginx setup
echo "Downloading nginx"
wget -c http://nginx.org/download/nginx-1.7.10.tar.gz
tar -xvf nginx-1.7.10.tar.gz

echo "Compiling nginx"
cd nginx-1.7.10
./configure --sbin-path=/usr/local/sbin --with-http_dav_module --with-http_ssl_module --add-module="$parentdir/nginx-dav-
ext-module-master"
make
sudo make install

cd "$parentdir/deployment"
echo "Please tell me the absolute location of the data folder of your server"
read data_folder
sudo chmod 777 $data_folder

echo "Please go to server/nginx folder and ensure the configurations (data folders and so on) are ok"
echo "Then press ENTER to continue"
read continue

sudo cp "$parentdir/server/nginx/nginx.conf" /usr/local/nginx/conf/nginx.conf
sudo mkdir /usr/local/nginx/sites-enabled
sudo ln -fs "$parentdir/server/nginx/dandelion.conf" /usr/local/nginx/sites-enabled/dandelion.conf
sudo mkdir /var/cache/

mkdir "$parentdir/server/assets/lists"

sudo npm install -g bower
cd "$parentdir"
echo "Firing up services..."
npm install
bower install
sudo nginx
npm start &

#Admin user
sh create-admin.sh

echo "Cleaning folders.."
rm -rf $parentdir/deployment/nginx-1.7.10
rm -rf $parentdir/deployment/nginx-17.10.tar.gz
rm -rf $parentdir/deployment/node
```

Se puede apreciar que este script llama a otros tres scripts distintos en función del sistema operativo: *deploy-red-hat.sh, deploy-ubuntu.sh y deploy-mac.sh* y El contenido de los mismos es el siguiente:

deploy-red-hat.sh,

```
#!/usr/bin/env bash
#Deployment script for Red Hat based distributions
sudo yum install -y mongodb-org #http://docs.mongodb.org/manual/tutorial/install-mongodb-on-red-hat/

#Node
curl -sL https://rpm.nodesource.com/setup | bash -
yum install -y nodejs
```

deploy-ubuntu.sh,

```
#!/usr/bin/env bash
# Deployment script for Ubuntu based distributions

sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install libpcre3 libpcre3-dev
sudo apt-get install libexpat1-dev
sudo apt-get install zlib1g-dev
sudo apt-get install libssl-dev
sudo apt-get install python3-pip
sudo pip3 install watchdog
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
echo "deb http://repo.mongodb.org/apt/ubuntu "$(lsb_release -sc)"/mongodb-org/3.0 multiverse" | sudo tee
/etc/apt/sources.list.d/mongodb-org-3.0.list
sudo apt-get update
sudo apt-get install -y mongodb-org

#PPA by nodesource
curl -sL https://deb.nodesource.com/setup_0.12 | sudo bash -
sudo apt-get install -y nodejs
```

deploy-mac.sh,

```
#!/usr/bin/env bash
#Deployment script for Mac OS X
brew update
brew install mongodb
brew install node
```

Finalmente el script de deployment principal ejecuta un pequeño script para crear el primer usuario de la aplicación, *create-user.sh,* cuyo contenido es:

```
echo "Insert username for the first user"
read user

echo "Insert password"
read password

echo "Creating first user"
curl -X POST "http://localhost/api/users/$user?newpass=$password&initial_token=9K7oTnWKWVD3BX06380l74J2c8w857Lf”
```

13.6 Tests base de de datos

De cara a garantizar el correcto funcionamiento de la base de datos, se han realizado una serie de tests unitarios que aseguran que las operaciones básicas de la base de datos (creación, actualización y borrado de usuarios o recursos) funcionan correctamente, así como aseguran que los datos almacenados son coherentes (no hay duplicados ni entradas incorrectas). Los test se encuentran en el mismo archivo

en el que se ha creado el código de la base de datos, esto es, en *server/util/mongodb.js*. Pueden ejecutarse de dos maneras:

```
node server/util/mongodb.js # Opción 1
npm test #Opción 2
```

A continuación se muestra el código de estos tests, así como la salida de la ejecución de los mismos:

```
var testId = "55c78e4f8edc26b9c878ab20";
var testUser = "batman";
function testCreateUsers(callback)
{
	var batman =
	{
		name: "batman",
		password: "test"
	};
	var joker =
	{
		name: "joker",
		password: "test"
	};
	var harley =
	{
		name: "harley",
		password: "test"
	};
	exports.createUser(batman, function(error)
	{
		testing.check(error,'Error creating user', callback);
		exports.createUser(joker, function(error)
		{
			testing.check(error,'Error creating user', callback);
			exports.createUser(harley, function(error)
			{
				testing.check(error,'Error creating user', callback);
				testing.success(callback);
			});
		});
	});
}
function testCreateResource(callback)
{
   var resource =
   {
      '_id': mongodb.ObjectID(testId),
      'name': 'test',
      'resource_kind': 'file',
      'owner': "batman",
      'mtime': Date.now(),
      'size': "test",
      'parent':"",
      'edit':['batman'],
      'locations':['sample/location/1','sample/location/2']
   };
   exports.createResource(resource,testUser, function(error)
   {
      testing.check(error,'Error creating resource', callback);
      testing.success(callback);
   });
}
function testUpdateResource(callback)
{
   var modifications =
   {
      'name': 'testmodificado',
```

```
        'size': "flejote",
    };
    exports.updateResource(testId,testUser, modifications, function(error)
    {
        testing.check(error,'Error updating resource', callback);
        testing.success(callback);
    });
}

function testAddPermission(callback)
{
    exports.addPermission(testId,testUser, "edit", ["joker", "harley", "gordon"], function(error, result)
    {
        testing.check(error, 'Error adding permission resource', callback);
        testing.success(callback);

    });
}
function testDeletePermission(callback)
{
    exports.deletePermission(testId,testUser, "edit", ["harley","gordon"], function(error, result)
    {
        testing.check(error, 'Error adding permission resource', callback);
        testing.success(callback);
    });
}

function testAddLocation(callback)
{
    exports.addLocations(testId,testUser, ['/another/location/1', '/another/location/2'], function(error, result)
    {
        testing.check(error, 'Error adding permission resource', callback);
        testing.success(callback);
    });
}
function testDeleteLocation(callback)
{
    exports.deleteLocations(testId,testUser, ['sample/location/1'], function(error, result)
    {
        testing.check(error, 'Error adding permission resource', callback);
        testing.success(callback);
    });
}

function testGetResourceId(callback)
{
    exports.getResourceId("sample/location/2", testUser, function(error, result)
    {
        testing.check(error,'Error getting resource ID', callback);
        testing.assertEquals(result,testId,'No resources found1', callback);
        exports.getResourceId([ 'sample/location/2','/another/location/1','/another/location/2' ],testUser, function(error, result)
        {
            testing.check(error,'Error getting resource ID', callback);
            testing.assertEquals(result,testId,'No resources found2', callback);
            //If we pass and array, it must match completely all locations, otherwise should fail.
            exports.getResourceId([ 'sample/location/2','/another/location/1'], testUser, function(error, result)
            {
                testing.check(error,'Error getting resource ID', callback);
                testing.assertNotEquals(result,testId,'No resources found3', callback);
                testing.success(callback);
            });
        });
    });
}


function testGetUserResources(callback)
{
    exports.getAllUserResources(testUser, function(error, result)
    {
        testing.check(error,'Error getting all resources', callback);
```

```
        testing.assert(result,'No resources found', callback);
        testing.success(callback);
    });
}

function testGetAllResources(callback)
{
    exports.getAllResources({},function(error, result)
    {
        testing.check(error,'Error getting all resources', callback);
        testing.assert(result,'No resources found', callback);
        testing.success(callback);
    });
}

function testRemoveResource(callback)
{
    exports.removeResource({_id: mongodb.ObjectID(testId)},testUser, function(error, result)
    {
        testing.check(error, 'Error removing resource', callback);
        testing.assertEquals(result, null, 'Should have removed one element', callback);
        testing.success(callback);
    });
}

function testCan(callback)
{
    exports.can({_id:mongodb.ObjectID(testId)},"joker","edit", function(error, result)
    {
        testing.check(error, 'Error checking can edit', callback);
        testing.assertEquals(result,false,"Joker can edit fails",callback);
                exports.can({_id:mongodb.ObjectID(testId)},"harley","edit", function(error, result)
            {
                testing.check(error, 'Error checking can edit', callback);
                testing.assertEquals(result,false,"harley can edit fails", callback);
                        exports.can({_id:mongodb.ObjectID(testId)},"batman","view", function(error, result)
                        {
                                testing.check(error, 'Error checking can view', callback);
                                testing.assertEquals(result,true, "batman can view fails", callback);
                                exports.can({_id:mongodb.ObjectID(testId)},"harley","view", function(error, result)
                                {
                                        testing.check(error, 'Error checking can view', callback);
                                        testing.assertEquals(result,false,"harley can view fails", callback);
                                        testing.success(callback);
                                });
                        });
            });
    });
}
function testDeleteUsers(callback)
{
        exports.deleteUser("batman", function(error)
        {
                testing.check(error,'Error deleting user', callback);
                exports.deleteUser("joker", function(error)
                {
                        testing.check(error,'Error deleting user', callback);
                        exports.deleteUser("harley", function(error)
                        {
                                testing.check(error,'Error deleting user', callback);
                                testing.success(callback);
                        });
                });
        });
}
function testDeleteResourcesUserCollection(callback)
{
        exports.deleteResourcesUserCollection("batman", function(error)
        {
                testing.check(error,'Error deleting user resources collection', callback);
                exports.deleteResourcesUserCollection("joker", function(error)
```

```
                {
                        testing.check(error,'Error deleting user resources collection', callback);
                        exports.deleteResourcesUserCollection("harley", function(error)
                        {
                                testing.check(error,'Error deleting user resources collection', callback);
                                testing.success(callback);
                        });
                });
        });
}
exports.test = function(callback)
{
   var tests = [
                testCreateUsers,
      testCreateResource,
      testUpdateResource,
      testAddPermission,
      testDeletePermission,
      testAddLocation,
      testDeleteLocation,
      testGetResourceId,
      testGetUserResources,
      testGetAllResources,
      testCan,
      testRemoveResource,
                testDeleteUsers,
                testDeleteResourcesUserCollection,
   ];
   exports.init(function(error)
   {
                if(error)
                {
                        return callback(error);
                }
      testing.run(tests,callback);
   });
};

//Execute tests if running directly
if (__filename == process.argv[1])
{
        exports.test(function(error)
        {
                if(error)
                {
                        console.log("Error executing tests", error);
                }
                else
                {
                        console.log("Database tests OK");
                }
        });
}
```

La salida de la ejecución de estos tests es la siguiente:

```
> dandelion-backend@1.0.0 test /Users/yarilolisure/PFC/dandelion-backend
> node server/util/mongodb.js

Connection to database stablished
✓ testCreateUsers: true
✓ testCreateResource: true
✓ testUpdateResource: true
✓ testAddPermission: true
✓ testDeletePermission: true
✓ testAddLocation: true
✓ testDeleteLocation: true
✓ testGetResourceId: true
✓ testGetUserResources: true
✓ testGetAllResources: true
✓ testCan: true
✓ testRemoveResource: true
✓ testDeleteUsers: true
✓ testDeleteResourcesUserCollection: true
Finished test main: ✓ main
Database tests OK
```

Figura 63: Salida tests base de datos

13.7 API REST Servidor Node.js

A continuación se expone la API del servidor REST, usada tanto por el cliente web como por el cliente de escritorio. El formato en el que se expone es el siguiente:

VERBO_HTTP URI

Seguido por los posibles parámetros de la URL o query necesarios. Se indica asimismo si es necesario enviar algún valor especial en el cuerpo o la cabecera de la petición.

A menos que se indique lo contrario, la respuesta seguirá el formato especificado en el estándar HTTP [IETF, 1999], con códigos 2xx para una respuesta satisfactoria, 4xx para un fallo del cliente y 5xx para un fallo del servidor.

POST /api/login

Para autenticar usuarios, esta llamada debe ser la primera que un cliente no autenticado realice. Nótese que el usuario puede autenticarse bien mediante su nombre de usuario bien usando su correo electrónico.

Body:

```
{ 'user': nombre_de_usuario o correo electrónico ,
 'password': contraseña_de_usuario};
```

Respuesta

```
{ 'username': Nombre de usuario,
 'token': JWT token};
```

*DELETE /api/resources/**

Para borrar un recurso del sistema.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre de recurso que intenta borrarse

Tabla 27: Parámetros URL DELETE /api/resources/*

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado
token	string/jwt	Token JWT de autenticación
id	objectID	ID del recurso en la base de datos
Kind	string	Tipo de recurso: “directory” o “file” (directorio o archivo)

Tabla 28: Parámetros query DELETE /api/resources/*

Respuesta

*GET /api/resources/**

Para obtener un recurso en concreto de la URL

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre del recurso que intenta obtenerse

Tabla 29: Parámetros URL GET /api/resources/*

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado
token	string/jwt	Token JWT de autenticación
id	objectID	ID del recurso en la base de datos
kind	string	Tipo de recurso: “directory” o “file” (directorio o archivo)

Tabla 30: Parámetros query GET /api/resources/*

Respuesta

Recurso en concreto, en el body de la response. Si se trata de una carpeta, ésta se ofrecerá como un archivo comprimido.

*PUT /api/resources/dirs/**

Para crear un directorio.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de directorio	string	Nombre del directorio que intenta crearse

Tabla 31: Parámetros URL PUT /api/resources/dirs/*

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado
token	string/jwt	Token JWT de autenticación

Tabla 32: Parámetros query PUT /api/resources/dirs/*

Headers

```
{'resource_kind': 'directory',
'owner': //Dueño del archivo
'mtime': Date.now(), //Fecha de creación del archivo según cliente (ajustada en el servidor)
'size':'0', //Calculado por el servidor
'parent':”” //Padre en la jerarquía del árbol de directorios “” para root.};
```

*PUT /api/resources/**

Para subir crear un archivo en el servidor.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de archivo	string	Nombre del archivo que intenta crearse

Tabla 33: Parámetros URL PUT /api/resources

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado
token	string/jwt	Token JWT de autenticación

Tabla 34: Parámetros query PUT /api/resources

Headers

```
{'resource_kind': 'file',
'owner': //Dueño del archivo
'mtime': Date.now(), //Fecha de creación del archivo según cliente (ajustada en el servidor)
'size':'0', //Calculado por el servidor
'parent':"" //Padre en la jerarquía del árbol de directorios "" para root.};
```

Body

Datos del archivo que se quiere subir

*MOVE /api/resources/**

Para mover un archivo de un lugar a otro. No disponible para carpetas. Parte del estándar WebDAV [IETF, 2007].

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de archivo	string	Nombre del archivo que intenta crearse

Tabla 35: Parámetros URL MOVE /api/resources/*

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado
token	string/jwt	Token JWT de autenticación

Tabla 36: Parámetros query MOVE /api/resources/*

Headers

```
{'host': //Nombre del servidor que realiza la peticón
'destination': //Destino a donde se quiere mover el archivo};
```

PUT /api/users/share/:id

Para compartir un recurso con otro usuario.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de usuario	string	Nombre del usuario al que se quiere compartir el recurso

Tabla 37: Parámetros URL PUT /api/users/share/:id

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario autenticado, dueño del recurso original.
token	string/jwt	Token JWT de autenticación
id	objectID	Id del recurso en base de datos
resource	string	Nombre del recurso que se quiere compartir
permissions	array de strings	Permisos que se le quieren dar al usuario con el que se comparte sobre el archivo. Valores posibles : “edit”, “view”.

Tabla 38: Parámetros query PUT/api/users/share/:id

Headers

```
{'kind': //Tipo de recurso a compartir: “directory” o “file” (directorio o archivo).
}
```

PUT /api/users/password/forgot/

Petición realizada para intentar recuperar la contraseña de un usuario que la ha olvidado. Al realizar esta llamada, el servidor enviará un email con instrucciones al correo indicado en el body de la petición.

Body

```
{'email':email}
```

POST /api/users/password/reset/:token

Petición realizada para, dado un token de reseteo de contraseña válido y enviado por correo, cambiar la misma por un usuario que la ha olvidado.

Parámetros de la URL

Parámetro	Tipo	Descripción
Token	string	Token de reseteo de contraseña, válido durante una hora. Cada token se identifica unívocamente con un usuario durante el tiempo que es válido.

Tabla 39: Parámetros URL POST /api/users/password/reset/:token

Parámetros de la query

Parámetro	Tipo	Descripción
password	string	Nueva contraseña.

Tabla 40: Parámetros query POST /api/users/password/reset/:token

PUT /api/users/password/:id

Para cambiar la contraseña de un usuario. Esta acción sólo es posible si el usuario envía su contraseña actual correctamente.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de usuario	string	Nombre de usuario que intenta cambiar la contraseña.

Tabla 41: Parámetros URL POST /api/users/password/:id

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario que intenta cambiar la contraseña.
password	string	Contraseña actual del usuario.
newpass	string	Nueva contraseña del usuario.

Tabla 42: Parámetros query POST /api/users/password/:id

PUT /api/users/validate/:token

Para validar un usuario creado.

Parámetros de la URL

Parámetro	Tipo	Descripción
Token	string	Token de validación de cuenta, válido durante una hora. Cada token se identifica unívocamente con un usuario durante el tiempo que es válido.

Tabla 43: Parámetros URL PUT /api/users/validate/:token

Parámetros de la query

Parámetro	Tipo	Descripción
initial_token	string	Valor especial alfanumérico para indicar al servidor que esta petición no necesita ser autenticada.

Tabla 44: Parámetros query PUT /api/users/validate/:token

GET /api/users/

Obtiene la lista de usuarios del sistema. Necesario para poder compartir usuarios desde el cliente web.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre del usuario que realiza la petición
token	string/jwt	Token jwt de autenticación

Tabla 45: Parámetros GET /api/users/

Respuesta

Lista de nombres de usuarios del sistema

PUT /api/users/:id

Crea un usuario nuevo.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de usuario	string	Nombre de usuario a crear

Tabla 46: Parámetros URL PUT /api/users/:id

Parámetros de la query

Parámetro	Tipo	Descripción
initial_token	string	Valor especial alfanumérico para indicar al servidor que esta petición no necesita ser autenticada.

Tabla 47: Parámetros query PUT /api/users/:id

Body

```
{
        name: nombre_de_usuario,
        email: email,
        password:contraseña,
};
```

POST /api/users/:id

Actualiza un usuario existente. Se incluirán en el cuerpo de la petición aquellos campos que deseen ser actualizados (excepto la contraseña).

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de usuario	string	Nombre de usuario a actualizar

Tabla 48: Parámetros URL POST /api/users/:id

Parámetros de la query

Parámetro	Tipo	Descripción
token	sring/jwt	Token JWT de autenticación

Tabla 49: Parámetros URL query /api/users/:id

Body

```
{
        name: nombre_de_usuario,
        email: email,
};
```

DELETE /api/users/:id

Borra un usuario existente. Esta petición borrará también los recursos del usuario, incluyendo los recursos compartidos de los que el usuario sea dueño.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de usuario	string	Nombre de usuario a actualizar

Tabla 50: Parámetros URL DELETE /api/users/:id

Parámetros de la query

Parámetro	Tipo	Descripción
token	string/jtw	Token JWT de autenticacón

Tabla 51: Parámetros query DELETE /api/users/:id

GET /api/p2p/magnet

Petición para crear un enlace tipo *magnet* a partir de un archivo determinado. No disponible para directorios. Adicionalmente, esta llamada crea un archivo *torrent* en el mismo directorio que el archivo original.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre del archivo del que obtener el enlace *magnet*

Tabla 52: Parámetros URL GET /api/p2p/magnet

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticacón
id	string	Id del recurso en la base de datos

Tabla 53: Parámetros query GET /api/p2p/magnet

Respuesta

Cadena con el enlace magnet, del tipo:

magnet:?xt=urn:btih:cf5e11ca8f6b59eadba1410181526ddf49bf5af1

GET /api/p2p/torrent

Para crear un archivo torrent a partir de un archivo determinado. El nuevo archivo torrent es creado en el mismo directorio que el archivo original. No disponible para directorios.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre de recurso del que obtener el enlace magnet

Tabla 54: Parámetros URL GET /api/p2p/torrent

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticacón
id	string	Id del recurso en la base de dato

Tabla 55: Parámetros query GET /api/p2p/torrent

GET /api/state

Esta petición provoca que el servidor examine la carpeta de recursos del usuario, actualice la base de datos con los nuevos recursos o modificaciones encontradas y devuelva la lista de recursos.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación

Tabla 56: Parámetros query GET /api/state

Respuesta

Array de objetos, en el que cada objeto representa un recurso y tiene el formato indicado en *8.1.6 Representación de datos.*

GET /api/state/list

Devuelve la lista en base de datos de recursos del usuario, sin examinar físicamente la carpeta del mismo.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación

Tabla 57: Parámetros query GET /api/state/list

Respuesta

Array de objetos, en el que cada objeto representa un recurso y tiene el formato indicado en *8.1.6 Representación de datos.*

GET /api/public-token

Obtiene un link público al archivo. No disponible para directorios.

Parámetros de la URL

Parámetro	Tipo	Descripción
Id de recurso	objectID	Id del recurso en la base de datos del cual se intenta obtener un link público.

Tabla 58: Parámetros URL GET /api/public-token

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación

Tabla 59: Parámetros query GET /api/public-token

Respuesta

El link generado es del tipo:

http://DIRECCIÓN_SERVIDOR/api/resources/nombreArchivo?user=usuario&public_token=f474e7b15bc3a325bc9792ae7c0081933afa31f8&id=56114b45fa3d924d7be419d8

Donde *public_token* es el *token* que identifica unívocamente al archivo.

GET /api/status/encrypted

Obtiene el estado de encriptación de un archivo consultando si existe el parámetro *encrypted* en su documento de la base de datos.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre de recurso del que se quiere averiguar el estado de encriptación.

Tabla 60: Parámetros URL GET /api/status/encrypted

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticacón
id	string	Id del recurso en la base de dato

Tabla 61: Parámetros query GET /api/status/encrypted

Respuesta

Devuelve *true* si el archivo está encriptado y *false* si no lo está.

PUT /api/encrypt

Encripta un archivo usando el algoritmo AES 256 [Daemen, 2002]. Asimismo, marca el archivo en base de datos con "*encrypted: true"*. El archivo generado tiene extension *.dat*. No disponible para directorios.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre de recurso que se quiere encriptar

Tabla 62: Parámetros URL PUT /api/encrypt

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticacón
id	string	Id del recurso en la base de datos

Tabla 63: Parámetros query PUT /api/encrypt

PUT /api/decrypt

Desencripta un archivo previamente encriptado y elimina de la base datos la propiedad, *encrypted,* que indica que el archivo esta encriptado. No disponible para directorios.

Parámetros de la URL

Parámetro	Tipo	Descripción
Nombre de recurso	string	Nombre de recurso que se quiere desencriptar

Tabla 64: Parámetros URL PUT /api/decrypt

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación
id	string	Id del recurso en la base de datos

Tabla 65: Parámetros query /api/decrypt

POST /api/chunks

Usado en subidas de archivos parciales. Sube un trozo (*chunk*) del archivo y devuelve el estado de la subida.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación

Tabla 66: Parámetros URL POST /api/chunks

Body

Trozo del archivo que se intenta subir.

Respuesta

"*partly_done*" si quedan más trozos por subir "*done*" si es el último trozo.

OPTIONS /api/chunks

Usado en subidas de archivos parciales para ver si el servidor permite CORS (*cross-domain requests*). Si el servidor no responde a esta petición, no se intentará subir el archivo.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación

Tabla 67: Parámetros de la query OPTIONS /api/chunks

GET /api/chunks/download/:identifier

Para obtener el identificador de un trozo determinado del archivo que se quiere subir.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jwt	Token JWT de autenticación

Tabla 68: Parámetros de la query GET /api/chunks/download/:identifier

GET /api/chunks/

Usado en subidas parciales para ver si un determinado trozo está o no en el servidor.

Parámetros de la query

Parámetro	Tipo	Descripción
user	string	Nombre de usuario
token	string/jtw	Token JWT de autenticación
flowChunkNumber	integer	Índice del trozo que se está subiendo en esta petición. Necesario para ordenar los trozos una vez se hayan subido todos
flowChunkSize	integer	Tamaño del trozo en bytes
flowTotalSize	integer	Tamaño total del archivo
flowRelativePath	string	Directorio en el que guardar el archivo, añadido al nombre del mismo
flowTotalChunks	integer	Número de trozos totales del archivo
flowIdentifier	string	Identificador del trozo que se está subiendo

Tabla 69: Parámetros de la query GET /api/chunks/

13.8 Interfaz SO / Servidor cliente de escritorio

El cliente de escritorio posee una interfaz para acceder, tanto al servidor (usando la API REST descrita en el *Anexo 13.7*) como al sistema operativo y modificar el sistema de ficheros.

A continuación se exponen, simplificadas por claridad, las llamadas en Javascript que conforman dicha interfaz. El código completo puede consultarse en el archivo *api.js*.

```
 //Llamadas al servidor
exports.login = function(callback){};
exports.getServerFileList = function(callback){};
exports.uploadFile = function(path, callback){};
exports.getFile = function(path, callback){};
exports.deleteFile = function(path, callback){};
exports.addDirectory = function(path, callback){};
exports.deleteDirectory = function(path, callback){};

//Llamadas al sistema operativo
exports.deleteLocalFile = function (path, callback){};
exports.deleteLocalDirectory = function (path, callback){};
exports.createLocalDirectory = function(path, callback){};
exports.copyResource = function(origin, destination, callback) {}; //Para crear copias de recursos conflictivos
```

13.9 Archivo de estilo JSHintrc

JSHint es una herramiente de Javascript que permite definir una serie reglas de estilo para programar en el mismo. Algunos editores de programación permiten definir un archivo llamado .j*shintrc* definiendo estas reglas. Posteriormente, a la hora de progarmar, el editor alertará ante la ruptura de las misma.

El editor usado, Atom, soporta este tipo de archivos, por lo que en este proyecto se ha usado el siguiente archivo *.jshintrc:*

```
/**
 * JSHint Dandelion Configuration File
 * See http://jshint.com/docs/ for more details
 * Examples at : https://github.com/jshint/jshint/blob/master/examples/.jshintrc
 **/

{
        "node": true,       // NodeJS JavaScript files
        "strict": true,     // Require "use strict" statement
        "white": true,      // Check whitespaces between operators
        "camelcase": false, // Force camelcase for vars
        "curly": true,      // Require curly braces in every scope
        "unused": true,     // Do not allow unused vars
        "funcscope": false, // Do not allow using a var out of its virtual scope
        "-W065": true,                  // Do not require radix as we use a safe parseInt
        "loopfunc": true,   // Allow functions to be declared in loops
        "quotmark": false,              // Do not warn about mixing ' and "
        "trailing": true,     // Warn about trailing spaces
        "undef": true,       // Force vars declaration before using them
        "smarttabs": true // Force mixign of tabs and spaces evaluation
}
```

13.10 Archivo de electron-builder config.json

A continuación se expone el archivo *config.json,* usado por electron-builder para crear instaladores en Mac OS X y Windows:

```
{
 "osx" : {
   "title": "Dandelion",
   "background": "assets/installer.png",
   "icon": "assets/tray_logo.icns",
   "icon-size": 80,
   "contents": [
     { "x": 438, "y": 240, "type": "link", "path": "/Applications" },
     { "x": 192, "y": 240, "type": "file" }
   ]
 },
 "win" : {
   "title" : "Dandelion",
   "icon" : "assets/tray_logo.ico"
 }
}
```

14. Bibliografía

Modulus, 2015: 2015, File-encryptor: Encrypts files using Node's built-in Cipher class.,

Aboukhadijeh, 2015: Aboukhadijeh, F., create-torrent: Create .torrent files, 2015

Alexeev, 2008: Alexeev, A., The Architecture Of Open Source Applications, Volume II, 2008

Amazon, 2015: Amazon, Amazon Elastic Compute Cloud, EC2, 2015

Apache, 2015: Apache, ab - Apache HTTP server benchmarking tool, 2015

Apple, 2015: Apple, Requisitos del sistema para iCloud, 2015

Apple, 2011: Apple, Apple Introduces iCloud, 2011

Auth0, 2015: Auth0, JSON Web Tokens , 2015

Beck, 2001: Beck K. et al, Manifesto for Agile Software Development, 2001

Boem, 1986: Boem B., A Spiral Model of Software Development and Enhancement ,

Bower, 2015: Bower, Bower: A package manager for the web, 2015

Daemen, 2002: Damen, J. , Rijmen, V., The Design of Rijndael: AES - The Advanced Encryption Standard, 2002

Dropbox,2009: Dropbox, Ayuda de Dropbox, ¿Qué es una copia en conflicto?, 2009

Ecma, 2011: Ecma International, Standard ECMA-262 5.1 Edition, 2011

Github, 2015: Github, Atom: A hackable text editor for the 21st Century, 2015

Github, 2008: Github, Github, 2008

Github, 2014: Github, Electron, 2015

Glenn, 1988: Glenn, E., Stephen, T., A Cookbook for Using View-Controller User Interface Paradigm in Smalltalk-80, 1988

Goldman, 2015: Goldman, N. , magnet- link: Get a magnet link from a torrent file, 2015

Google, 2012: Google, Introducing Google Drive... yes, really, 2012

Google, 2014: Google, Material Design, 2014

Google, 2010: Google, AngularJS API Docs, 2010

Google, 2015: Google, V8 JavaScript Engine, 2015

Gossman, 2005: Gossman, J., Introduction to Model/View/ViewModel pattern for building WPF apps, 2005

Hunt, 1976: Hunt J, McIlroy, M., An Algorithm for Differential File Comparison, 1976

IETF, 1999: IETF, Hypertext Transfer Protocol -- HTTP/1.1, 1999

IETF, 2007: IETF, HTTP Extensions for Web Distributed Authoring and Versioning (WebDAV), 2007

Jgraph, 2015: Jgraph, Draw.io, 2015

Jones, 2015: Jones, M. et al., JSON Web Token (JWT), 2015

Kahn, 2013: Kahn, J., Apple at Q3 call: iCloud reaches 320M accounts, 900B iMessages, 125B photo uploads, 2013

Koorapati, 2014: Koorapati, N., Streaming File Synchronization, 2014

Loeb, 2012: Loeb, S., , 2012

Lorenz, 2015: Lorenz, T, Readdirp: Recursive version of fs.readdir with streaming api, 2015

Microsoft, 2013: Microsoft, A file copy operation fails when files or folders have long paths in Windows Explorer, 2013

Miller, 2015: Miller, P. , chokidar: A neat wrapper around node.js fs.watch / fs.watchFile, 2015

Mogul, 2002: Mogul, J. ,Compaq WRL et. al., Delta encoding in HTTP, 2002

MongoDB, 2008: MongoDB, Model Tree Structures with Materialized Paths, 2008

MongoDB, 2015: MongoDB, The MongoDB 3.0 Manual, 2015

Mozilla, 2015: Mozilla, HTTP access control (CORS),

Nathan, 2011: Nathan, W., Asynchronous Callbacks, 2011

Nginx, 2015: Nginx, nginx documentation, 2015

Node, 2015: Node, Node Package Manager, 2015

Node.js, 2015: Node.js, Node.js v4.1.1 Documentation, 2015

Nodejitsu, 2015: Nodejitsu, node-http-proxy: A full-featured http proxy for node.js, 2015

Ogden, 2015: Ogden, M., electron-packager: package and distribute your electron app in OS executables (.app, .exe etc) via JS or CLI, 2015

Park, 2015: Park, T., Paper: Material is the metaphor, 2015

Provos, 1998: Provos, N. , Mazieres, D., Bcrypt Algorithm, 1998

Pydio, 2013: Pydio, Pydio, 2013

Rackspace, 2015: Rackspace, Mailgun: The Email Service for Developers, 2015

Request, 2015: Request, Request: Simplified HTTP request client, 2015

Richardson, 2015: Richardson, node-fs-extra: Node.js: extra methods for the fs object like copy(), remove(), mkdirs(), 2015

Schmidt, 2000: Schmidt C., An Object Behavioral Pattern forDemultiplexing and Dispatching Handles for Synchronous Events, 2000

SpiderOak, 2015: SpiderOak, Zero Knowledge, 2015

Strongloop, 2015: Strongloop, Express - API Reference, 2015

The Document Foundation, 2015: The Document Foundation, LibreOffice: Documentación, 2015

Torvalds, 2009: Torvalds, L., Git, 2009

Twitter, 2011: Twitter, Bootstrap, 2011

Unitech, 2015: Unitech, Production process manager, PM2, 2015

WageIndicator, 2015: WageIndicator, Tusaliario.es - Salario bruto mensual para Ingeniero de software informático,