Universidad Autónoma de Madrid

Escuela politécnica superior

Trabajo de fin de máster

Desarrollo de una infraestructura para el escaneado y clasificación de documentos mediante dispositivos móviles inteligentes

Máster Universitario en Ingeniería Informática

Autor: Natalia Roales González
Tutor: David Arroyo Guardeño
Fecha: Febrero 2017

Desarrollo de una infraestructura para el escaneado y clasificación de documentos mediante dispositivos móviles inteligentes

AUTOR: Natalia Roales González
TUTOR: David Arroyo Guardeño

Dpto. de Ingeniera Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Febrero 2017

Resumen

Uno de los mayores retos en la actualidad es la adecuada combinación de movilidad y fácil acceso a la información. Dentro de este fácil acceso a la información es de especial relevancia el contar con métodos para la datificación de la información. Dicha datificación ha de partir de la digitalización de contenidos. En este proyecto se presenta una solución para efectuar tal digitalización y favorecer el posterior proceso de datificación. Con objeto de favorecer su usabilidad y adaptabilidad se ha optado por una solución híbrida, esto es, un producto software que permite digitalizar documentos desde cualquier tipo de dispositivo móvil.

Palabras clave
Digitalización, aplicaciones móviles híbridas, OCR, LDA, node.js, ionic.

Abstract

One of the biggest challenges at the moment is the adequate combination of mobility and easy-access to information. Regarding this last concern, it's very important to have some method to conduct datafication. The first step in datatification is document digitalization. This project proposes a solution to carry out such a digitalization in order to foster the latter datafication. For the sake of usability and adaptability, in this project we have adopted an hybrid software solution to enable documents digitalization from any type of mobile device.

Key words
Digitalization, hybrid mobile applications, OCR, LDA, node.js, ionic.

Índice general

Índice de figuras

Índice de tablas

Índice de *listings*

Glosario

Aplicación híbrida Aplicación cuyo desarrollo es capaz de ser ejecutado en distintas plataformas móviles.

Desarrollador full-stack Desarrollador que domina tecnologías de sistemas, backend y frontend, incluyendo las subáreas de cada una de estas tres.

Latent Dirichlet allocation (LDA) Algoritmo que nos permite obtener sobre un texto un conjunto de términos característicos del mismo.

Optical Character Recognition (OCR) Reconocimiento óptico de caracteres.

Test Driven Development (TDD) Desarrollo guiado por pruebas [1].

Capítulo 1

Introducción

El proyecto que vamos a comenzar tiene como objetivo el desarrollo de una aplicación móvil, multiplataforma, que sea capaz de escanear documentos de texto en papel y posteriormente mostrar una lista de posibles temáticas relacionadas con el documento. Abarcará las distintas fases del ciclo de vida del software, salvo la fase de mantenimiento: análisis del problema, diseño de la solución, implementación del código que defina la plataforma y pruebas sobre la misma.

Además del desarrollo de la aplicación mencionada, se persigue otro objetivo muy distinto pero igualmente importante para el ámbito en el que nos encontramos, y es el de poner en práctica todos los conocimientos y competencias adquiridas en el master que acaba de ser cursado. Se intentarán cubrir las siguientes áreas de la disciplina de la informática:

- Ingeniería del Software
- Metodologías de desarrollo
- Desarrollo web
- Minería de datos
- Seguridad
- Persistencia de datos

1.1. Motivación

No es ninguna novedad que cada vez más usuarios utilicen sus **dispositivos smartphone** para realizar tareas que antes desempeñaban en sus equipos de sobremesa. Al fin y al cabo, nuestros teléfonos móviles son pequeños ordenadores que nos acompañan durante todo el día y que caben en el bolsillo de nuestro pantalón. Por ello, el mercado de las aplicaciones móviles es cada vez mayor, y los desarrolladores de este tipo de software son cada vez más demandados por las **empresas**.

Sin embargo, existe hoy una amplia variedad de plataformas para estos dispositivos, distinguiéndose tres principales sistemas operativos: iOS, Android y Windows Phone. Además, en sistemas como Android, contamos con infinidad de ROMs y de versiones actualmente soportadas. Esto plantea un verdadero reto desde el punto de vista del desarrollador, siendo muy difícil satisfacer a todo tipo de usuarios.

Por suerte, existen tecnologías como PhoneGap, Appium o Ionic que nos permiten implementar desarrollos compatibles con múltiples plataformas de manera simultanea. Es por ello que en este proyecto se ha decidido optar por este tipo de aplicaciones, denominándose **aplicaciones híbridas**.

Otra de las principales motivaciones de este proyecto es adquirir conocimientos y habilidades suficientes para desenvolvernos con soltura en el campo del **desarrollo web**. Durante los estudios universitarios cursados a lo largo de todos estos años, el desarrollo web es una disciplina a la que apenas se le otorga importancia, pero que acaba siendo muy importante en el entorno laboral de la mayoría de las empresas que se dedican a la informática.

Finalmente, existe también un motivo por el cual hemos seleccionado para este proyecto la aplicación que vamos a desarrollar. Hoy en día existen diversas herramientas que escanean documentos de texto, teniendo incluso la capacidad de extracción de su contenido. También existen un montón de servicios que nos proporcionan almacenamiento de nuestros contenidos en la nube. Por otro lado, tenemos herramientas que nos permiten mantener un listado de documentos (o de otras cosas), pudiendo añadir a cada entrada del registro cierta información.

En este momento seguro que disponemos de innumerables documentos almacenados en múltiples dispositivos, en muchas ocasiones siendo necesario haberlos digitalizado o escaneado desde un dispositivo para poder visualizarlo desde otro distinto, y requiriendo ser clasificados manualmente. Por tanto, sería muy interesante contar con una aplicación que unificase en un único sistema tanto el procedimiento de escaneo y clasificación de documentos como el almacenamiento de los mismos.

1.2. Estructura del documento

El presente documento se divide en **siete capítulos**. Vamos a describir de forma sintetizada el contenido de cada uno de ellos:

- El primer capítulo, tratándose del actual, nos presenta una introducción al proyecto, exponiéndose las motivaciones que nos llevan a su desempeño y los objetivos del mismo.
- El segundo capítulo, **Estado del arte**, nos proporciona un acercamiento al marco tecnológico que engloba a este proyecto. Forman parte de él las plataformas móviles, la digitalización de documentos y el almacenamiento de contenidos.
- El tercer capítulo, **Análisis del problema**, se enumeran los requisitos funcionales y no funcionales del software que vamos a desarrollar.

- En el cuarto capítulo, **Diseño de la plataforma**, tendremos una vista panorámica de la plataforma tanto a nivel visual como a nivel de arquitectura, necesaria para poder realizar una buena implementación posteriormente.
- En el quinto capítulo, **Implementación de la solución**, conoceremos todos los detalles de cómo se ha implantado la solución diseñada en el capítulo anterior.
- En el sexto capítulo, **Pruebas**, se describirán las pruebas que se han llevado a cabo para validar el sistema, tanto unitarias como funcionales.
- Acabaremos en el séptimo capítulo con unas **Conclusiones**, presentando además una propuesta de trabajo futuro.

Capítulo 2

Estado del arte

De forma previa al análisis del problema que nos ocupa, como es el desarrollo de nuestra plataforma de digitalización y clasificación de documentos, es conveniente informarse sobre el marco tecnológico que podemos encontrarnos en el mercado actual.

2.1. Plataformas móviles

Actualmente el mercado móvil está superando al uso de escritorio para las tareas de navegación y uso de Internet. Según diversos estudios como el realizado por la compañía de investigación StatCounter[1], a finales del año 2016 el 51,3 % de los accesos web en Internet se han efectuado desde dispositivos móviles, siendo el 49,7 % restante desde dispositivos de escritorio, o las estadísticas[2] para la Unión Europea donde en el año 2016, 8 de cada 10 usuarios han accedido a información web a través de un teléfono móvil o smartphone.

Esto demuestra que las plataformas móviles a día de hoy representan la mayoría de dispositivos con los que día a día nos mantenemos conectados. Dentro de los dispositivos actuales, tienen una gran representación 3 sistemas operativos móviles:

- Android
- iOS
- Windows Phone

Durante el último trimestre del año 2016, estas 3 plataformas conforman el 99,6 % de todos los dispositivos móviles[3], estando repartidos de la siguiente forma:

[1] http://gs.statcounter.com/press/mobile-and-tablet-internet-usage-exceeds-desktop-for-first-time-worldwide (Último acceso 12-02-2017)

[2] http://ec.europa.eu/eurostat/documents/2995521/7771139/9-20122016-BP-EN.pdf (Último acceso 12-02-2017)

[3] http://www.idc.com/promo/smartphone-market-share/os;jsessionid=47D25B86F062D59C37005939EAD81A13 (Último acceso 12-02-2017)

Android	iOS	Windows Phone	Otros
86,8 %	12,5 %	0,3 %	0,4 %

Cuadro 2.1: Cuota de mercado en dispositivos móviles (*Q4* de 2016)

Android

Android es el sistema operativo mayoritario en los dispositivos móviles. Se trata de un sistema operativo basado en el núcleo Linux desarrollado inicialmente por la compañía Android Inc. (actualmente propiedad de Google) en el año 2007. Está incorporado en multitud de dispositivos de diversos fabricantes, no estando limitado su uso a uno específico.

iOS

iOS es el sistema operativo desarrollado por la compañía Apple Inc. desde el año 2007. Se trata del sistema operativo de los dispositivos iPhone, iPad, iPod Touch y Apple TV, exclusivos de la compañía. Los dispositivos con iOS destacan por su gran optimización en conjunto con el hardware, al no disponer de una alta variedad de fabricantes que soportan el sistema operativo.

Windows Phone

Sucesor del Windows Mobile, Windows Phone es el sistema operativo desarrollado por la compañía Microsoft. Actualmente, los dispositivos Windows Phone, aunque tienen una pequeña representación en el mercado móvil, su tendencia es a desaparecer. Cuentan con sustituto en el sistema operativo Windows 10 Mobile, lanzado a mediados de 2016 y con muy poca representación por el momento.

Tipología de aplicaciones

Al disponer de distintos sistemas operativos, cada uno con sus plataformas de desarrollo independientes, entran en juego dos tipologías de aplicaciones móviles:

Aplicaciones monoplataforma

Requiere un desarrollo cada tipo: Android, iOS y Windows Phone. Existen plataformas de desarrollo en sus respectivos lenguajes de programación, para desarrollar aplicaciones exclusivas para uno de ellos. De esta forma se consigue además una integración del 100 % entre hardware y software, al disponer de todas las características y funcionalidades que el sistema operativo ofrece, pero si queremos que la aplicación se ejecute en varias plataformas será necesario repetir el desarrollo de forma nativa tantas veces como plataformas deseemos.

Aplicaciones multiplataforma

Otra tipología de aplicación es la **aplicación multiplataforma**, donde hacemos uso de tecnologías web como HTML, JavaScript o CSS para implementar un solo desarrollo que pueda ejecutarse en una gran variedad de plataformas. La desventaja de este tipo de aplicaciones radica en que la integración con el hardware no es tan alta como con las monoplataforma (por ejemplo para acceder a funciones del sistema operativo para la gestión de la memoria), pero por otro lado cuenta con una gran ventajas: le ahorrará tiempo de desarrollo y mantenimiento al desarrollado de forma considerable.

2.2. Digitalización de documentos

La digitalización de documentos es la conversión de documentos físicos e imágenes de cualquier tipo y en cualquier formato, en sus equivalentes digitales. La digitalización permite reducir el espacio físico y permite además una búsqueda mucho más rápida y eficiente de los documentos al poder indexar el contenido en las plataformas digitales.

Además, permite que podamos trabajar con el contenido de los documentos entre distintas aplicaciones y servicios, permitiendo así una mejora de la eficiencia del trabajo tanto en tiempo como en costes.

Actualmente, existe una técnica de digitalización documental llamada Reconocimiento Óptico de Caracteres (OCR por sus siglas en inglés, *Optical Character Recognition*), que permite identificar símbolos y caracteres tomando como origen una imagen. Al tener como origen una imagen, esta técnica no está exenta de algunos inconvenientes:

- Una mala calidad de imagen de entrada generará un mal reconocimiento de texto, bien sea por una mala resolución o por una mala definición o contraste entre los caracteres y el fondo.

- Si los caracteres tienen un formato de fuente alejado del estándar o con los caracteres muy unidos entre sí, dificultará de igual forma el reconocimiento dando como resultado un texto de salida más pobre.

Este tipo de algoritmos se pueden especializar en distintos casos de uso, como por ejemplo el reconocimiento de matrículas de vehículos, donde se tiene claro el patrón buscado, siendo por ejemplo 4 números seguido de 3 letras para las matrículas modernas.

Existen diversos algoritmos de OCR, pero la gran mayoría mantienen una estructura de trabajo similar, pudiendo dividir el algoritmo en las siguientes técnicas:

Preprocesado de la imagen

Dentro de la imagen se necesita aislar el texto del resto del fondo. Para ello, existen distintas técnicas para optimizar el siguiente reconocimiento de caracteres separando texto de imagen. Una de las opciones es reducir el número de colores disponibles en la entrada, dejando una imagen en blanco y negro con pocos cambios en la escala de grises.

Además, se puede identificar si el texto está girado y los algoritmos tratarán de corregir esta alineación para mantener una horizontal o vertical, según el alfabeto a identificar.

Entre otros procesos del preprocesado se incluyen la reducción del ruido de la imagen, la detección de líneas y párrafos o la normalización del escalado de los caracteres.

Reconocimiento de caracteres

Una vez tengamos la imagen preprocesada, el algoritmo tratará de identificar los caracteres del texto. Para ello, primero se tienen que aislar los caracteres de las palabras.

Actualmente, existen dos métodos principales de reconocimiento:

- **Reconocimiento de patrones**: Según una base de conocimiento de distintos caracteres con distintas fuentes tipográficas, se tratará de encontrar una correlación del caracter reconocido en la imagen, con los distintos patrones almacenados. Esta técnica es más eficiente para textos tipografiados.

- **Aprendizaje automático y minería de datos**: Algoritmos de clasificación y regresión también son efectivos a la hora de reconocer texto a partir de una imagen. Se trata de algoritmos que convierten la imagen de entrada en vectores y que en base a un entrenamiento y aprendizaje previos, permitirán clasificar el texto mediante coincidencias con elementos cercanos en base probabilística. Este tipo de algoritmo es eficaz para reconocer texto manuscrito, habiendo entrenado antes al algoritmo con distintos tipos de letra manuscrita.

Postprocesado

Una vez se dispone ya de una salida digital de caracteres, se procede a un postprocesado, donde en base a un diccionario de palabras y de sentencias comunes se procede a una corrección del texto reconocido.

Por ejemplo, si el OCR ha reconocido la palabra *MA**0**RID*, con un dígito *0* entre las letras *MA* y *RID*, será capaz de corregir el carácter *0* por una *D*, ya que la palabra *MADRID* existe, mientras que *MAD0RID* no.

2.3. Almacenamiento de contenidos

Para poder almacenar contenido dentro del software móvil, se dispone a día de hoy de diversas opciones:

- **Almacenamiento local**: Se almacena en el propio dispositivo, a través de una base de datos o bien en ficheros locales, el contenido de la aplicación. La ventaja de esta solución está en que no se necesita conectividad para poder acceder al contenido, la desventaja está en la limitación de almacenamiento del propio dispositivo.
- **Almacenamiento en la nube**: También disponemos de soluciones basadas en la nube, donde la aplicación se conecta a un repositorio online donde poder almacenar el contenido. La ventaja de este tipo de soluciones está en que proporcionan un entorno fiable, con alta disponibilidad y escalable. La desventaja está en que por lo general tiene un coste asociado.

  Existen varias soluciones como Google Cloud Storage, Amazon S3, Microsoft Azure, o bien disponer de un almacenamiento remoto en un servidor hosteado.

Capítulo 3

Análisis del problema

Antes de comenzar a desarrollar nada debemos pasar por una primera fase de análisis, en la cual detectaremos las necesidades que nos surgen tanto a nivel funcional como a nivel técnico. Si al finalizar la implementación del sistema hemos cubierto todas esas necesidades detectadas habremos completado con éxito nuestro proyecto, y podremos decir que hemos realizado un buen desarrollo que cumple con el objetivo marcado.

Estas necesidades de las que hablamos se denominan requisitos, siendo estos de dos tipos: funcionales y no funcionales. Vamos a ver de qué se trata cada uno de ellos y vamos a enumerarlos en concreto para este proyecto.

3.1. Requisitos funcionales

Los requisitos funcionales de la aplicación quedan definidos mediante **historias de usuario** [2]. Una historia de usuario describe una funcionalidad que aportará cierto valor al usuario, y siempre debe redactarse en un lenguaje fácil de comprender.

El formato con el que se suelen escribir las historias de usuario es siempre muy parecido. En ellas se debe indicar el rol para el cual la historia aportará valor y qué se desea obtener con ella. Por ejemplo, la estructura de una historia de usuario podría ser *"Yo como **rol** quiero _____ para poder _____"*.

Además se suele utilizar la abreviatura **YCUQ** para expresar *"Yo como usuario quiero"*, YCDQ para expresar *"Yo como desarrollador quiero"*, YCNQ para expresar *"Yo como negocio quiero"*, y así hasta un largo etcétera.

A continuación se enumeran las historias de usuario definidas para este proyecto, cuyo rol será el usuario final de la aplicación:

RF1 YCUQ poder registrarme.

RF2 YCUQ poder autenticarme en la aplicación.

RF3 YCUQ poder cerrar la sesión.

RF4 YCUQ poder almacenar y etiquetar documentos de texto en la plataforma, de manera que el sistema clasifique el contenido del documento y me sugiera algunos temas.

RF5 YCUQ ver una lista de mis documentos almacenados, filtrados por etiqueta.

RF6 YCUQ ver una lista de todas las etiquetas asignadas a mis documentos.

RF7 YCUQ poder buscar un documento por su título.

RF8 YCUQ poder visualizar el contenido de un documento de texto ya almacenado.

RF9 YCUQ que ningún otro usuario tenga acceso a mis documentos.

3.2. Requisitos no funcionales

Los requisitos no funcionales, al contrario que los requisitos funcionales, no aportan un valor de negocio al usuario. Son propiedades que debe poseer el sistema que vayamos a desarrollar para que el funcionamiento del mismo cumpla con el comportamiento deseado. Podemos encontrarnos requisitos no funcionales que especifiquen criterios de rendimiento, disponibilidad, coste, seguridad o usabilidad, entre otras muchas categorías.

Los requisitos no funcionales definidos para este proyecto son los siguientes:

RNF1 - El sistema deberá contar con una interfaz de usuario para dispositivos móviles, y deberá ser capaz de ejecutarse tanto en un sistema Android como en un iPhone. Es decir, se tratará de una Aplicación híbrida.

RNF2 La aplicación móvil hará uso de la cámara del teléfono para escanear los documentos.

RNF3 El idioma de los textos escaneados será el inglés.

RNF4 El servidor deberá soportar peticiones de varios usuarios a la vez.

RNF5 El tiempo de respuesta de las operaciones realizadas en la aplicación móvil deberá ser inferior a 2 segundos.

RNF6 Los servicios deberán estar securizados a través del protocolo SSL, utilizándose HTTPS para realizar las peticiones al servidor.

RNF7 Deberá hacerse uso de cualquier sistema de seguridad que proteja los contenidos almacenados en la plataforma, siendo accesible cada recurso únicamente por su autor.

RNF8 Las herramientas utilizadas para el desarrollo de la plataforma serán de uso gratuito y libre.

RNF9 El código desarrollado deberá contar con un set de test unitarios que verifiquen la correcta construcción del sistema.

Capítulo 4

Diseño de la plataforma

Una vez hemos realizado el análisis del sistema, es hora de diseñar la plataforma. Nuestra plataforma constará de un conjunto de tecnologías, llamado *stack* tecnológico, a través del cual implantaremos la parte servidora y la parte cliente.

En función a nuestra experiencia con proyectos similares, será necesario seleccionar cuáles son las tecnologías más adecuadas para llevar a cabo nuestro desarrollo, sin olvidarnos de que deberán integrarse unas con otras de forma lo más óptima posible.

En el diseño de la aplicación se definirá también qué aspecto tendrá la interfaz de usuario, o cuáles serán los servicios necesarios para que esta pueda comunicarse con el servidor. Y como el mejor diseño de un sistema es su código fuente, prescindiremos de cualquier diagrama para definir su comportamiento y emplearemos la técnica TDD *(Test Driven Development)*, cuyo principio se basa en la definición de la funcionalidad de un sistema a través de sus test. Profundicemos en el diseño del sistema al completo en los siguientes aparatos.

4.1. El stack tecnológico

Una parte importante del diseño de nuestra plataforma es la elección de las tecnologías sobre las cuales vamos a construir nuestros sistema. En el marco de la programación web, una correcta decisión acerca de las herramientas a seleccionar puede resultar muy complicada, pues tenemos a nuestra disposición infinidad de tecnologías con características muy distintas.

Comenzamos nuestra investigación, pero ante un abanico tecnológico tan amplio como es el actual debemos hacernos una serie de preguntas. Vamos a citar algunas de estas cuestiones:

- ¿Cuáles son los recursos *hardware* que tenemos a nuestro alcance?
- ¿De qué recursos económicos disponemos?
- ¿Sobre qué dispositivos deberá funcionar nuestra plataforma?

- ¿Cuál será el máximo de conexiones concurrentes que deberá soportar el sistema?
- ¿Existirá conectividad en red?
- ¿De qué tipología serán las operaciones que se van a realizar y cuál será el nivel de rendimiento requerido para su ejecución?
- ¿Qué datos deberán persistir?
- ¿Es importante la seguridad?

Esto simplemente es una reflexión, pues la respuesta a estas cuestiones puede encontrarse en la definición de los requisitos no funcionales. En segundo lugar debemos identificar qué **componentes** formarán parte del entorno de nuestro sistema, así como la tipología de los mismos. Para el caso de nuestra plataforma, se han detectado los siguientes componentes:

- Un **servidor** remoto, accesible desde Internet.
- Un **cliente**, instalado en uno o varios dispositivos móviles, y conectando con el servidor de forma concurrente.
- Una **base de datos** en la que persistir información sobre el registro de usuarios y los documentos escaneados.
- Conjunto de **ficheros**, posiblemente alojados en el servidor remoto, que almacenarán el contenido de los ficheros de texto.

Ya tenemos acotados tanto los criterios de selección de las herramientas que vayamos a utilizar como los componentes de la plataforma que vamos a tener que construir. Por tanto, solo nos queda decidir cuáles serán las tecnologías que emplearemos para desarrollar el sistema. Vamos a enumerarlas una a una, razonando el por qué de cada elección.

NodeJS y Express

Para la implementación de lo que será el servidor de nuestra aplicación, se ha decidido emplear NodeJS y Express. **NodeJS** es JavaScript del lado del servidor. Proporciona una arquitectura dirigida a **eventos**, con un modelo **asíncrono**, y utilizando por debajo el motor de JavaScript de Google, llamado V8.

Hay muchas razones por las cuales esta tecnología ha sido la primera opción a contemplar. Hace unos años, pensar en JavaScript significaba pensar en aplicaciones pequeñas y poco robustas. Pero hoy en día, el lenguaje ha evolucionado tanto que se ha convertido en una opción tan válida como Java, C++ o PHP a la hora de diseñar un sistema de gran envergadura. NodeJS es un proyecto de **código abierto**, y gracias a ello disponemos de infinidad de módulos que podemos incorporar a nuestro desarrollo, pudiendo cubrir todas las necesidades de nuestro sistema.

Es multiplataforma, rápido y tolerante a un alto nivel de concurrencia, ideal si queremos que puedan acceder al servidor a la vez una cantidad considerable de usuarios desde sus dispositivos móviles.

Otro punto a favor es que iniciar un servidor NodeJS desde cero es bastante rápido. En pocos minutos podemos tener un servidor totalmente operativo, sin necesidad de complicarnos a la hora de configurar nada. Solamente emplearemos nuestro tiempo en instalar los módulos que vayamos a utilizar y en implementar la lógica necesaria para obtener la funcionalidad requerida. Además, no necesitamos ningún entorno de desarrollo especial, ya que podemos desarrollar cualquier aplicación NodeJS simplemente con un editor de notas y desplegarla a través de una terminal.

Por último, NodeJS cuenta con **Express**, un framework que nos ayuda a la hora de desarrollar una aplicación web. Gracias a este módulo podremos trabajar con el protocolo HTTP (en especial con HTTPS, en nuestro caso), facilitándonos la gestión de las rutas, de las peticiones y respuestas, de sus cabeceras, etc.

Ionic y AngularJS

Con la intención de abarcar un mercado lo más amplio posible, se ha decidido realizar un **desarrollo híbrido** para cubrir la parte cliente de nuestro sistema. Dos de las tecnologías más conocidas y experimentadas para desarrollar aplicaciones móviles híbridas son PhoneGap e Ionic, ambas basadas en Apache Cordova.

Una aplicación híbrida **nos ahorrará tiempo** de desarrollo frente a la implementación de una aplicación nativa, ya que solo realizaremos un desarrollo que funcionará en diversas plataformas. Por contra, el rendimiento podría ser ligeramente peor, pero en nuestro caso no es muy relevante porque nuestra aplicación se limitará a enviar y recibir mensajes del servidor.

Aunque PhoneGap proporcione soporte para más plataformas que Ionic y permita usar cualquier *framework* JavaScript y CSS, Ionic nos aporta **simplicidad**. Esta última tecnología trabaja con AngularJS, similar a NodeJS pero en el lado del cliente. AngularJS nos ayuda a realizar aplicaciones web integrando HTML, CSS y JavaScript, apoyándose en el patrón Modelo-Vista-Controlador (MVC). Por tanto, no habría que realizar ninguna integración entre las dos herramientas, pues una trabaja por defecto con la otra.

Ionic tiene soporte total para las **plataformas móviles con más cuota de mercado**, que son Android, iOS Windows Phone. Si la aplicación compila sobre una plataforma, compilará sobre todas. Además, Ionic posee **diseños bastante refinados y agradables** para nuestras vistas , lo cual será de agradecer por parte del usuario final. Con todo esto, nos decantamos por Ionic y AngularJS para llevar a cabo nuestro desarrollo.

MongoDB

Para persistir la información sobre los documentos escaneados y sobre los usuaarios registrados en el sistema, hemos decidido utilizar MongoDB, una base de datos no relacional orientada

a documentos.

Ahora mismo existen muchos gestores de bases de datos en el mercado, pero principalmente podemos distinguir dos tipos: bases de datos relacionales o SQL, y bases de datos no relacionales o NoSQL. Una base de datos relacional estructura la información en distintas tablas, cada una de ellas con ciertos campos que definimos al crearlas, y con un conjunto de relaciones que consiguen enlazar la información de unas con otras. Además, podemos establecer una serie de claves o restricciones en cada una de las tablas, indicando por ejemplo si un campo es identificativo o si depende de otro campo situado en otra tabla. Cada tabla almacena la información en forma de registros.

Sin embargo, las bases de datos no relaciones funcionan de manera muy distinta. No estructuran la información en tablas, y dependiendo del tipo de base de datos NoSQL la estructura será una u otra. Las hay que persisten la información en forma de columnas, otras que lo hacen en forma de grafo, algunas que definen una estructura clave-valor, y como MongoDB, las hay que estructuran los datos en forma de documentos. En total, podemos encontrarnos con más de 150 sistemas de bases de datos no relacionales.

Pero más allá de la estructura de los datos almacenados, existen algunas características y virtudes que nos hacen decantarnos por MongoDB. Este sistema es ideal para almacenar grandes conjuntos de información, y en especial información con una estructura variante a lo largo del tiempo. En un futuro se pretende que en nuestro servidor existan una gran cantidad de documentos escaneados, e incluso una gran cantidad de usuarios registrados. También se pretende ir mejorando el algoritmo de clasificación de nuevos documentos, por lo que la estructura con la que se almacena la información cambiará, habiendo posteriormente más campos o cambiando los que ya existen.

Frente a sistemas de base de datos relacionales, como puede ser MySQL, MongoDB es muchísimo más rápido realizando operaciones de búsqueda. Esto nos beneficia porque la mayoría de las operaciones que hagamos durante el uso de la aplicación serán búsquedas de información.

Por otro lado, MongoDB se integra a la perfección con NodeJS, y la forma de almacenar o recuperar información encaja a la perfección con la tipología de nuestros datos. Y por último, se trata de un sistema de código abierto.

4.2. La API de servicios

Para que la interfaz de usuario se integre con el lado del servidor hemos definido una API de servicios REST, los cuales se encargarán de autenticar al usuario en la plataforma, recuperar los datos almacenados, actualizarlos cuando sea necesario y cargar nuevos documentos.

A continuación se muestra la especificación de cada uno de los servicios definidos en el diseño de la plataforma, describiéndose brevemente el comportamiento de cada uno de ellos, e indicándose cuáles son sus entradas y salidas.

```
{
  "token": "eyJ0eXAiOiJKV1ToLCJubAciOeJIUzI2NiJ7.eyJdzWIeOuJ1c5
     VyMSUsstlpd
            CI1MTQRNDQyNjE9MiwiZXhwIjoxNDg2NjM1NzgyfQ.EPgVVJQmL8zzN
               3POQD3SH
            ngHcAPcbqkK96LvayAic9p"
}
```

Code Listing 4.1: Respuesta OK del servicio /login.

```
{
  "error": "Bad credentials."
}
```

Code Listing 4.2: Respuesta errónea del servicio /login.

POST - /login

- **Resumen**: Permite al usuario autenticarse en la aplicación.
- **Descripción**: El servicio comprueba en primer lugar que el nombre de usuario introducido existe en el sistema, y si es así verifica que la contraseña introducida es correcta. En caso de que los dos datos coincidan, el proceso de autenticación finalizará correctamente y el servicio devolverá el token de autenticación que corresponda al usuario, y si algún dato es erróneo se retornará un código de error.

  Para autorizar al usuario el acceso al resto de los servicios, el token devuelto será almacenado posteriormente en el dispositivo.
- **Parámetros de entrada**: En el cuerpo de la petición nos encontraremos los siguientes parámetros de entrada:
  - **user:** Nombre de usuario.
  - **password:** Contraseña del usuario, ya encriptada para evitar ataques tipo *man in the middle*.
- **Respuesta**: Si el funcionamiento es correcto, el servicio devolverá un **código 200 OK** y la respuesta tendrá la estructura que vemos en la figura Code Listing 4.1. Si hay un error en las credenciales, el servicio devolverá un **código 401 de error** y la respuesta será la que vemos en la figura Code Listing4.2.

GET - /tags

- **Resumen**: Devuelve todas las etiquetas correspondientes al usuario autenticado en la aplicación.

```
{
  "tags": [ "cats", "cheese", "evolution", "physic" ]
}
```

Code Listing 4.3: Respuesta OK del servicio /tags.

```
{
  "error": "Internal server error: Mensaje de error"
}
```

Code Listing 4.4: Error interno del servidor.

- **Descripción**: El servicio realiza una consulta a base de datos en la que se recuperan todos los documentos cuyo propietario es el usuario autenticado. Posteriormente se obtiene la etiqueta de cada uno de los documentos recuperados, y se aplica el filtro *distinct* sobre la lista de etiquetas para quedarnos con las que son distintas entre sí.

  Si todo ha ido bien el servicio devolverá una lista de etiquetas, y si no retornará un código de error.

- **Parámetros de entrada**: No existen parámetros de entrada para este servicio.

- **Respuesta**: Si el comportamiento es el esperado, el servicio devolverá un **código 200 OK** y la respuesta tendrá la estructura que vemos en la figura Code Listing 4.3.

  Si existen errores internos en el servidor, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

GET - /tag/{tagId}/documents

- **Resumen**: Devuelve todos los documentos a los que se le asigna la etiqueta especificada en la url del servicio, correspondientes al usuario autenticado.

- **Descripción**: Este servicio es similar al anterior. Realiza una consulta a base de datos en la que se recuperen todos los documentos cuyo propietario es el usuario autenticado, y cuya etiqueta es el valor del campo **tagId** que se especifica en la url.

  Si los datos se recuperan correctamente el servicio devolverá una lista de objetos de tipo Documento, cuyos atributos son el título, la fecha de creación, la etiqueta asociada y el usuario propietario. En caso contrario, se devolverá un código de error.

- **Parámetros de entrada**: El único parámetro de entrada para este servicio es **tagId**, y su valor debe enviarse a través de la url.

- **Respuesta**: Si el funcionamiento es correcto, el servicio devolverá un **código 200 OK** y la respuesta tendrá la estructura de la figura Code Listing 4.5.

  Si hay algún error interno en el servidor, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

```
{
  "documents":
      [ {
            "title": "Uncertainty principle",
            "creation": "2016-07-10 14:53:20.397Z",
            "tag": "physic",
            "user": "natalia"
        },
        {
            "title": "Momentum",
            "creation": "2016-10-12 17:36:08.687Z",
            "tag": "physic",
            "user": "natalia"
        }
      ]
}
```

Code Listing 4.5: Respuesta OK del servicio /tag/tagId/documents.

GET - /document/{documentId}

- **Resumen**: Devuelve el contenido del documento cuyo identificador se especifique en la url del servicio, siempre que el usuario autenticado tenga permisos para acceder al mismo.
- **Descripción**: Dado el identificador que aparezca en la url (definico como **documentId**), el servicio leerá el fichero de texto cuyo nombre sea dicho identificador acompañado de la extensión '.txt', devolviendo su contenido.

  En caso de no poder leerse el fichero de manera correcta se retornará un código de error.
- **Parámetros de entrada**: El único parámetro de entrada para este servicio es **documentId**, y su valor debe enviarse a través de la url.
- **Respuesta**: Si no se han producido errores en el proceso, el servicio devolverá un **código 200 OK** y la respuesta tendrá la estructura de la figura Code Listing 4.6

  Si por el contrario existe algún error interno en el servidor, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

POST - /document

- **Resumen**: Recibe una imagen y la procesa obteniendo el texto que aparece en ella, devolviendo un conjunto de etiquetas asociadas.
- **Descripción**: En primer lugar el servidor recibe una imagen en formato Base64, a la cual aplica el OCR Tesseract y extrae el texto contenido. Posteriormente se obtienen los tokens que componen el texto, sobre los que se aplica el algoritmo LDA resultando de la operación un conjunto de *topics* o temas.

```
{
  "title": "Uncertainty principle",
  "content": "The uncertainty principle is not readily apparent on
     the macroscopic scales of everyday experience. So it is
     helpful to demonstrate how it applies to more easily
     understood physical situations. Two alternative frameworks for
      quantum physics offer different explanations for the
     uncertainty principle. The wave mechanics picture of the
     uncertainty principle is more visually intuitive, but the more
      abstract matrix mechanics picture formulates it in a way that
      generalizes more easily."
}
```

Code Listing 4.6: Respuesta OK del servicio /tag/tagId/documents.

Después se crea un documento en base de datos cuyos atributos son el título, la etiqueta, la fecha de creación y el usuario. El título y la etiqueta estarán en blanco hasta que el propietario los defina.

Se almacenan en el servidor la imagen escaneada y un fichero de texto con el contenido del documento, cuyo título será el identificador del documento creado en base de datos seguido de la extensión correspondiente ('.jpg' para la imagen y '.txt' para el fichero de texto).

Por último devolvemos la lista de etiquetas obtenidas mediante el algoritmo LDA, que corresponderán al término con más probabilidad de cada *topic* o tema asignado al texto.

- **Parámetros de entrada**: En el cuerpo de la petición nos encontraremos los siguientes parámetros de entrada:
  - **image:** Cadena cuyo contenido es la imagen del documento en formato Base64.
  - **nTerms:** Número de términos que conformen cada *topic* o tema.
  - **nTopics:** Número de *topics* o temas que queremos obtener a partir del texto, los cuales se le mostrarán más tarde al usuario para que seleccione uno de ellos.
- **Respuesta**: Si el funcionamiento es correcto, el servicio devolverá un **código 201 OK** y una respuesta vacía. Si existen errores internos en el servidor, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

PUT - /document/tag

- **Resumen**: Establece sobre el último documento escaneado la etiqueta recibida como parámetro.
- **Descripción**: El servicio actualiza en base de datos el último documento almacenado, modificando su etiqueta por el valor del parámetro **tagId** que reciba en el cuerpo de la petición.

- **Parámetros de entrada**: En el cuerpo de la petición nos encontraremos los siguientes parámetros de entrada:
  - **tag:** Etiqueta que asignaremos al documento correspondiente.
- **Respuesta**: Si el funcionamiento es correcto, el servicio devolverá un **código 204 OK** y una respuesta vacía. Si no, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

PUT - /document/rename

- **Resumen**: Renombra el último documento escaneado con el título especificado en la url del servicio.
- **Descripción**: El servicio actualiza en base de datos el último documento almacenado, modificando el título por el valor del parámetro **title** que reciba en el cuerpo de la petición.
- **Parámetros de entrada**: En el cuerpo de la petición nos encontraremos los siguientes parámetros de entrada:
  - **title:** Título que asignaremos al documento correspondiente.
- **Respuesta**: Si el funcionamiento es correcto, el servicio devolverá un **código 204 OK** y una respuesta vacía. Si hay un error interno en el servidor, el servicio devolverá un **código 500 de error**, retornando el código de la figura Code Listing 4.4.

4.3. La base de datos

Para persistir datos como la información sobre los documentos escaneados, las etiquetas generadas y los usuarios registrados en la plataforma, se ha decidido utilizar una base de datos no relacional, en concreto se ha optado por el uso de **MongoDB**.

En primer lugar, utilizamos una base de datos **no relacional** porque el rendimiento en cuanto a búsquedas frente a una base de datos relacional es mucho mayor. Se desea que en un futuro la aplicación almacene miles de documentos, y por tanto será requerida una gran velocidad en las búsquedas.

Por otro lado, entre todas las bases de datos no relacionales, hemos seleccionado MongoDB por ser una base de datos **orientada a documentos**, que es con lo que vamos a trabajar nosotros. Además es de las más conocidas y utilizadas en los proyectos más actuales, lo cual nos proporciona cierta **estabilidad y facilidad** de integración con cualquier otra pieza del sistema.

MongoDB trabaja con objetos en formato **JSON**, lo cual no entraña una gran dificultad. Aunque en realidad esto no es del todo cierto, pues tenemos que ser un poco más concretos. El gestor de base de datos almacena los documentos en formato **BSON**, una representación binaria de objetos JSON, aunque a nuestros ojos estos dos formatos son idénticos.

En cuanto a nuestro esquema de base de datos, tendremos dos colecciones: documentos escaneados y usuarios registrados. Los objetos a almacenar en cada una de las dos colecciones anteriores se caracterizarán por tener la siguiente estructura.

Documents

- **_id** (de tipo *ObjectId*): Identificador que MongoDB asigna al objeto al ser almacenado en una colección.
- **title** (de tipo *String*): Título con el que el usuario nombra el documento escaneado.
- **tag** (de tipo *String*): Etiqueta seleccionada por el autor del documento para su clasificación, resultado del algoritmo *LDA*.
- **user** (de tipo *String*): Nombre del autor del documento escaneado.
- **creation** (de tipo *Date*): Fecha en la cual se escaneó el documento.

Users

- **_id** (de tipo *ObjectId*): Identificador que MongoDB asigna al objeto al ser almacenado en una colección.
- **user** (de tipo *String*): Nombre que el usuario eligió al registrarse.
- **password** (de tipo *String*): Contraseña cifrada del usuario registrado.

Se muestra en la figura Code Listing 4.7 un ejemplo de cada uno de los dos tipos de objeto. En el ejemplo del documento de texto vemos que aparece un campo __v. Esta propiedad, generada por MongoDB, se llama versionKey y contiene una revisión interna del documento. Por el momento no haremos uso de la misma.

4.4. La interfaz de usuario

Cuando en un proyecto software se requiere desarrollar una interfaz gráfica a través de la que manejar la aplicación, es una buena práctica esbozar una maqueta de la misma, por simple que sea. No solo le sirve al desarrollador como referencia a la hora de trazar las vistas, también es útil para confirmar que la idea de la interfaz que tienen en mente tanto el desarrollador como el usuario final coinciden. Es mucho más fácil rectificar una maqueta si no refleja lo que el usuario final desea encontrarse en su aplicación que corregir un desarrollo ya finalizado.

Siguiendo este principio, hemos trazado una maqueta sencilla que muestra cómo será nuestra aplicación móvil una vez esté terminada. En ella podemos observar el siguiente flujo:

1. En primer lugar, al acceder a la aplicación nos aparecerá la vista de inicio de sesión (Figura A.1).

```
{
    "_id" : ObjectId("578261600f000b2e09c2feda"),
    "title" : "Teaching Evolutions in Schools",
    "tag" : "evolution",
    "user" : "natalia",
    "creation" : ISODate("2016-07-10T14:53:20.397Z"),
    "__v" : 0
}

{
    "_id" : ObjectId("5782696f96c2e5ad4b43ac22"),
    "user" : "natalia",
    "password" : "482c811da5d5b4bc6d497ffa98491e38"
}
```

Code Listing 4.7: Ejemplo de objetos BSON.

2. Tras iniciar sesión, podremos ver la lista de etiquetas asignadas a todos nuestros documentos (Figura A.2).

3. Si pulsamos en cualquiera de las etiquetas, veremos los documentos etiquetados con ella (Figura A.3).

4. Si pulsamos en uno de los documentos que aparecen, podremos leer su contenido en pantalla (Figura A.4).

5. Desde la vista de Etiquetas podemos acceder al menú de la aplicación, pulsando en el icono que aparece en la parte superior izquierda de la pantalla (Figura A.5).

6. Si accedemos a la opción 'Nuevo documento' la aplicación abrirá la cámara de fotos del teléfono. Buscaremos un documento que fotografiar y pulsaremos en 'Capturar' para escanearlo (Figura A.6).

7. La aplicación nos sugerirá una serie de etiquetas detectadas por el algoritmo de etiquetado, acompañadas de un porcentaje numérico calculado por el mismo para cada una de ellas (Figura A.7).

8. Por último, tras seleccionar una etiqueta entre las sugeridas por la aplicación, introduciremos un título para el nuevo documento y pulsaremos en Guardar (Figura A.8).

4.5. Desarrollo guiado por pruebas: TDD

Como metodología de diseño de software utilizaremos Test Driven Development (TDD) [1]. Esta técnica consiste en desarrollar los test de forma previa al código que implementa la funcionalidad de la aplicación. Desarrollaremos en primer lugar un test que falle y que pruebe cierta funcionalidad, y posteriormente desarrollaremos el código necesario para que el test funcione (es decir, que pase de rojo a verde), llegando así a implementar la funcionalidad requerida.

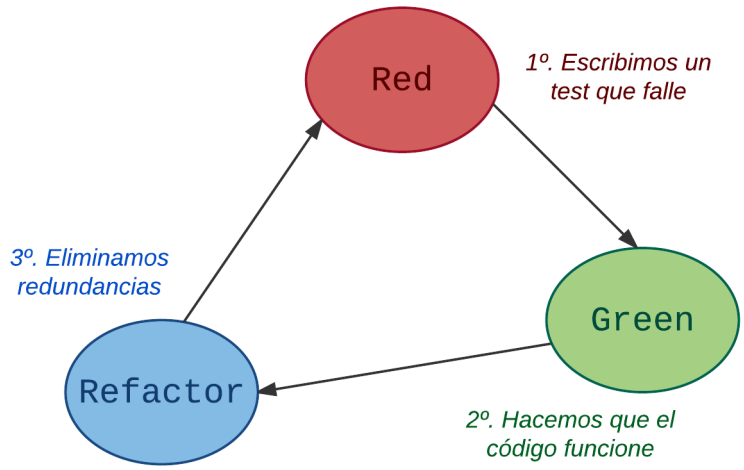

Figura 4.1: Ciclo del desarrollo guiado por pruebas.

Contando con una buena batería de test que sean lo suficientemente descriptivos, evitaremos tener que crear y mantener documentación adicional, pues el diseño de la aplicación se encontrará en el mismo código. Además, la implementación del sistema será muchísimo más robusta y mantenible, ya que si se produce cualquier fallo los test fallarán y podremos detectar rápidamente la causa del problema. En la Figura 4.1 podemos ver un diagrama correspondiente al ciclo de desarrollo de TDD.

Capítulo 5

Implementación de la solución

Ya tenemos claro cuál va a ser el diseño del sistema que vamos a desarrollar. Acabamos de decidir qué tecnologías conformarán nuestra arquitectura, hemos esbozado una maqueta sencilla que refleja la apariencia de la aplicación que el usuario final manejará, hemos definido una API de servicios, el esquema de la base de datos, e incluso la técnica mediante la cual llevaremos a cabo el desarrollo. Con todo esto, procedamos a implementar la solución final.

5.1. El lado del servidor o *Backend*

En el sistema a implementar podemos distinguir claramente dos extremos: el lado del servidor o *Backend*, y el lado del cliente o *Frontend*. En primer lugar, veamos cómo está estructurado nuestro servidor NodeJS. En la Figura 5.1 podemos ver un esquema en el que aparecen todos los módulos que conforman la arquitectura del servidor (y a qué módulo accede cada uno, representado con una flecha dirigida), los cuales se describen a continuación:

- **server.js**: En él definimos el servidor NodeJS, estableciendo el protocolo SSL, la conexión con la base de datos MongoDB, y el fichero de rutas correspondientes a los servicios REST.
- **routes.js**: Contiene la definición de la API de servicios REST de la aplicación, y la lógica de todos ellos salvo la del servicio de escaneado de documentos.
- **documentProcessor.js**: Define toda la lógica de digitalización y etiquetado de documentos.
- **user.js**: En él se define el modelo de datos User, necesario para gestionar los usuarios en base de datos.
- **document.js**: Define el modelo de datos Document, necesario para gestionar los documentos en base de datos. Se utilizará tanto para recuperar los documentos que estén almacenados como para almacenar los nuevos que escaneemos.

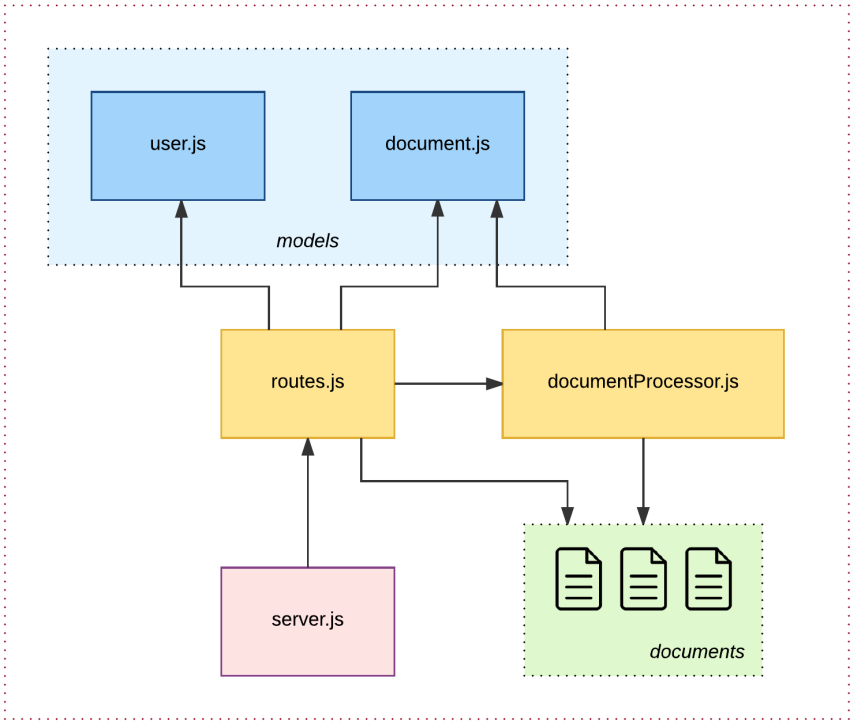

Figura 5.1: Arquitectura del servidor NodeJS.

En las siguientes secciones vamos a detallar la implementación de la lógica del servidor. Primero veremos cómo se definen las rutas, luego entraremos en detalle en el servicio de digitalización y etiquetado de contenidos, y posteriormente veremos cómo se realiza la integración con MongoDB, pues el resto de servicios se basan principalmente en consultas a base de datos.

5.2. Definición de las rutas de los servicios

En el fichero **routes.js** se definen las rutas a todos los servicios de la API. *Express* nos facilita la gestión de las rutas, proporcionándonos un *Router* o enrutador con los métodos **GET**, **POST**, **PUT** y **DELETE**. Para definir una ruta en *Express*, especificaremos el tipo de método (por ejemplo GET), la ruta correspondiente a la URL, y los middleware que se ejecutarán. Estas funciones middleware realizan cambios en los objetos de petición (req) y respuesta (res) e invocan a la siguiente función middleware de la pila a través de la función ***next***.

Podemos ver un ejemplo de definición de rutas en la figura Code Listing 5.1, el cual se trata de un fragmento de código correspondiente al módulo routes.js de nuestra arquitectura. En este fragmento definimos dos rutas: la primera pertenece al servicio POST de login, y la segunda pertenece al servicio GET de recuperación de etiquetas. Las funciones 'login', 'isAuthenticated' y 'findAllTags' se tratan de los middleware que hemos descrito anteriormente.

```
var router = express.Router();

module.exports = function(app) {

  router.post('/login', login);

  router.get('/tags', isAuthenticated, findAllTags);

  app.use('/', router);
}
```

Code Listing 5.1: Modelo de datos Document.

5.3. Digitalización y etiquetado de contenidos

El proceso de digitalización y etiquetado de contenidos es uno de los más complejos e importantes de la lógica de la aplicación. Este proceso corresponde a todo lo que ocurre en el sistema desde que escaneamos un documento de texto con la cámara del móvil hasta que obtenemos las etiquetas que lo clasifican. De un vistazo, podemos ver en la Figura 5.2 en qué consiste cada una de sus fases.

Desde la aplicación móvil fotografiaremos el documento que queramos clasificar, y tras haber tomado la captura se codificará la imagen en *Base64* para ser enviada al servidor. El servidor recibirá la imagen en formato *Base64* y la decodificará, extrayendo posteriormente su contenido a través de una herramienta de tipo *Optical Character Recognition (OCR)*. Se ha utilizado el paquete *Tesseract*[1] de NodeJS para ello.

Después aplicamos un tokenizador, en especial ***WordTokenizer*** del módulo *natural*[2] de NodeJS. En este paso obtendremos el conjunto de términos que componen el texto, para en último lugar obtener las etiquetas que lo clasifiquen. Por último, aplicaremos el algoritmo *Latent Dirichlet allocation (LDA)* al conjunto de términos obtenidos previamente.

LDA establece una serie de métricas sobre cada término del texto, entre las cuales se encuentra la frecuencia de aparición en el mismo, y selecciona uno o varios términos en función a un valor calculado a partir de dichas métricas. Cuando mayor sea el valor calculado, mayor probabilidad tendrá un término de identificar el texto analizado, y por tanto será seleccionado como posible categoría o etiqueta.

Hemos elegido este algoritmo como método de categorización como primera aproximación a un modelo de clasificación, ya que es el mecanismo estándar y además el resultado es directo, incorporándolo en nuestro desarrollo a través del paquete LDA[3] de NodeJS. Como última aclaración, tanto el número de categorías asignadas al texto como el número de términos en los que se basa cada una, será configurable en la llamada al módulo.

[1] https://www.npmjs.com/package/tesseract (Último acceso 12-02-2017)
[2] https://www.npmjs.com/package/natural (Último acceso 12-02-2017)
[3] https://www.npmjs.com/package/lda (Último acceso 12-02-2017)

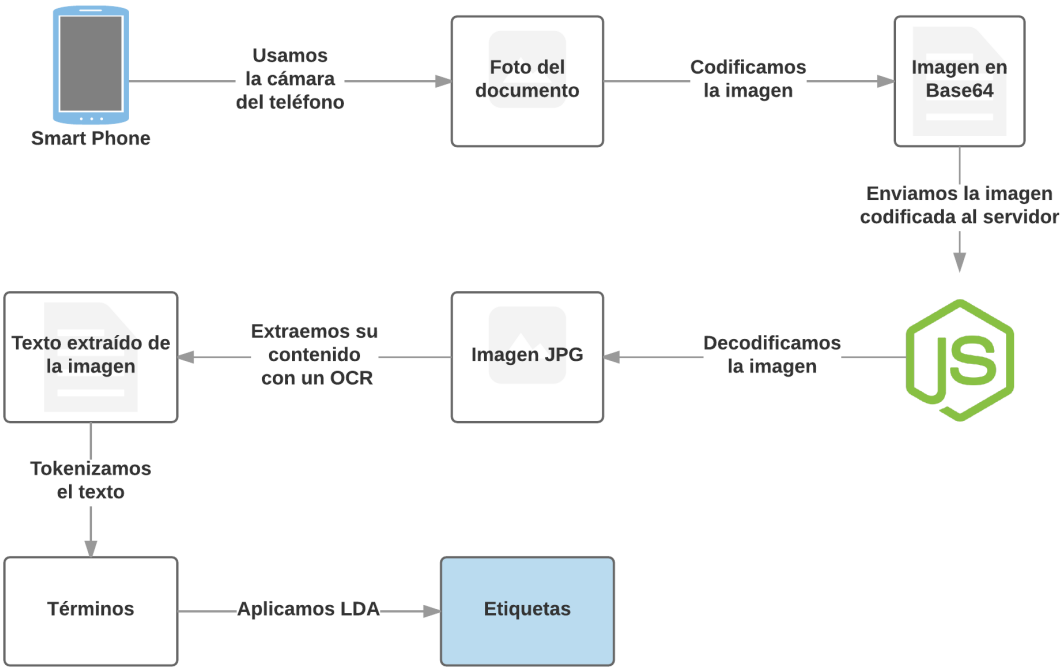

Figura 5.2: Proceso de digitalización y etiquetado de documentos.

5.4. Integrando el servidor con base de datos

Para integrar nuestro servidor NodeJS con MongoDB se ha hecho uso del paquete Mongoose. Mongoose es una herramienta de modelado de objetos de MongoDB en un entorno asíncrono como NodeJS. Nos facilitará el trabajo a la hora de conectarnos a la base de datos, definir los modelos y lanzar consultas para almacenar, consultar o actualizar información en el sistema de persistencia no relacional.

En nuestro caso vamos a realizar consultas sobre usuarios y documentos, además de crear nuevos documentos y actualizarlos. Para llevar a cabo todo esto, en primer lugar, tendremos que definir la conexión a base de datos. A continuación, tendremos que crear un Schema para cada modelo de datos, a través de los cuales manejaremos los objetos de cada tipo de colección. Por último definiremos las *queries* sobre los modelos de datos creados, utilizando los métodos *find*, *findOne* y *save*.

Una vez aclarado todo esto, procedemos a implementar la lógica correspondiente en el lado del servidor.

Instalación de Mongoose

Instalar Mongoose en nuestro servidor es muy fácil, basta con ejecutar el siguiente comando sobre una terminal:

```
$ npm install mongoose
```

Definiendo los modelos de datos

Para definir un nuevo modelo de datos crearemos un nuevo *Schema*. En la definición de un *Schema* será necesario indicar su nombre, los campos que tendrá el modelo de datos, el tipo de cada uno de ellos, si son obligatorios o no, e información adicional.

Nosotros hemos definidos dos modelos de datos distintos: 'User', que corresponde a los usuarios registrados en el sistema, y 'Document', que corresponde a los documentos de texto escaneados por los usuarios. Estos modelos se han definido en los ficheros 'document.js' y 'user.js' respectivamente, agrupándose estos en un directorio 'models'.

Podemos ver el contenido del fichero 'document.js' en la figura Code Listing 5.2.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var Document = new Schema({
    title:       { type: String },
    creation:    { type: Date, default: Date.now },
    tag:         { type: String },
    user:        { type: String, require:true }
});

Document.path('title').validate(function (v) {
    return ((v != "") && (v != null));
});

module.exports = mongoose.model('Document', Document);
```

Code Listing 5.2: Modelo de datos Document.

Como podemos observar, los atributos del esquema coinciden con los campos de los objetos almacenados en la colección 'Documents'. Al crear una nueva instancia de este esquema el campo user es obligatorio y deberá ser completado, y el campo 'creation' (fecha de creación del documento) por defecto se rellenará con la fecha actual. Además, se ha añadido una restricción sobre el campo 'title' (título del documento), cuyo valor no podrá ser ni una cadena vacía ni *null*.

Podemos ver el contenido del fichero 'user.js' en la figura Code Listing 5.3. En este caso, los atributos coinciden con los campos de los objetos almacenados en la colección 'Users'. Todos los

campos son obligatorios y no hay ninguna restricción sobre los datos.

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;

var User = new Schema({
    name:     { type: String, require: true },
    password: { type: String, require: true }
});

module.exports = mongoose.model('User', User);
```

Code Listing 5.3: Modelo de datos User.

Implementando las *queries*

Ya hemos definido los modelos de datos que nos harán falta para realizar cualquier operación sobre las colecciones existentes en nuestra base de datos. Principalmente vamos a tener tres tipos de operaciones distintas: *find*, *findOne* y *save*.

La operación ***find*** nos permite recuperar una lista de objetos como resultado de una consulta sobre base de datos. Un caso de este tipo de operación en nuestro código es la recuperación de todos los documentos que han sido etiquetados con cierto término y creados por un usuario en concreto, siendo este el usuario que inició sesión en la aplicación.

Para implementar este caso hemos definido la función 'findDocumentsByTagId', que devolverá la lista de documentos recuperados por la operación *find* de Mongoose. Esta operación se invocará sobre el modelo de datos 'Document', y recibirá un objeto JSON con el valor de los parámetros a filtrar en la búsqueda, seguido de una función *callback*. Podemos ver el código de la implementación en la figura Code Listing 5.4.

Si recordamos nuestra API REST de servicios, existe uno de ellos que nos devuelve la información que 'findDocumentsByTagId' nos proporciona. Los parámetros que nosotros enviemos en la url del servicio se encontrarán almacenados en req.params, y por ello nuestra consulta extrae el valor de la etiqueta o del campo 'tag' de la variable 'req.params.tagId'. Por otro lado, en algún momento del flujo de datos almacenamos el nombre del usuario autenticado en la variable 'req.user', siendo ahora cuando lo extraemos.

En cuanto a la función *callback*, define qué código HTTP y qué valores serán devueltos en cada caso. Si no se produce ningún error en el servidor el servicio devolverá un código OK 200 con la lista de documentos, y un código de error 404 en caso de no existir documentos para dicha búsqueda. Si por el contrario se produce un error en el servidor, el servicio devolverá un código de error 500.

Nuestro segundo tipo de operación, ***findOne***, funciona de forma muy similar a la operación *find*, pero a diferencia de esta devuelve un solo objeto en vez de una lista. En el proceso de au-

```
var Document = require('../models/document.js');
...

findDocumentsByTagId = function(req, res) {

   return Document.find(
      {'tag': req.params.tagId,
       'user': req.user},

      function(err, documents) {
         if(!err) {

            if(!documents) {
               return res
                  .status(404)
                  .send({error: 'Tag not found.'});
            }
            return res
               .status(200)
               .send({documents:documents});

         } else {
            return res
               .status(500)
               .send({error: 'Internal server error: ' + err.
                  message});
         }
      }
   );
};
```

Code Listing 5.4: Implementación de la función findDocumentsByTag.

tenticación de un usuario debemos buscar en base de datos un usuario cuyo nombre y contraseña introducidos coincidan, lo cual puede resolverse con *findOne*.

Hemos desarrollado la función 'login' para implementar este procedimiento, invocando sobre el modelo de datos 'User' la operación ya mencionada. Podemos ver el código en la figura Code Listing 5.5, que sigue una estructura muy parecida a la de la función anterior.

En este caso estamos extrayendo los valores que asignamos a 'userName' y 'password' de los parámetros que enviamos al servicio de autenticación, y solo filtraremos resultados por nombre de usuario. En la consulta que ejecutamos a través de Mongoose podríamos filtrar resultados por nombre de usuario y por contraseña, así y evitarnos un caso de error dentro de la función callback. Según hemos desarrollado el servicio podemos distinguir cuándo el usuario introduce un nombre inexistente y cuándo introduce una contraseña errónea, aunque por el momento devolveremos el mismo código de error para no dar un exceso de información a posibles atacantes, pero para

depurar nos será útil.

Por último, la operación ***save***, nos permite crear nuevos objetos en base de datos o actualizar los que ya existen. Para almacenar un nuevo objeto en MongoDB desde el código NodeJS, crearemos una nueva instancia del modelo correspondiente rellenando los campos que deseemos persistir, e invocaremos a la función save sobre la misma. Tras almacenar el nuevo documento tendremos en el campo '_id' el identificador que MongoDB le haya asignado.

En la figura Code Listing 5.6 podemos ver el código de la función 'storeDocument', en la que se utiliza la operación *save* para almacenar en base de datos la información sobre un nuevo documento de texto. Esta función es invocada desde otro sitio y por eso no devuelve ningún código 200 OK de retorno, pero almacena en el campo 'documentId' de la respuesta el identificador que MongoDB haya asignado al nuevo documento.

Creando la conexión entre el servidor NodeJS y el servidor MongoDB

Lo único que nos queda es conectar nuestro servidor con la base de datos, nuevamente a través de Mongoose. Será tan fácil como invocar a la función 'connect, pasando como parámetro la dirección ip, el puerto y el nombre de la base de datos. Esto lo haremos añadiendo al fichero server.js el código de la figura Code Listing 5.7.

5.5. Seguridad

Para conseguir un mínimo de robustez [3] frente a usuarios no legítimos en la red cuyo objetivo sea atacar nuestro sistema a través de cualquier agujero de seguridad, se han empleado algunos mecanismos que nos proporcionarán un poco de protección frente a este tipo de individuos.

Las intenciones de los atacantes pueden ser muy dispares, como suplantar la identidad de un usuario, alterar la integridad de los mensajes que se envíen o reciban o interceptar cierta información, ya sea por diversión o por motivos algo más oscuros.

5.5.1. Secure Socket Layers (SSL)

Por un lado, para garantizar la integridad y confidencialidad de los datos intercambiados entre el servidor y el cliente, se utiliza el sistema **SSL** *(Secure Socket Layers)* acompañado del protocolo **HTTPS**. Las comunicaciones a través de este protocolo son más seguras que por HTTP, ya que la información que se envía y se recibe lo hace de manera cifrada, en concreto con un cifrado de **clave pública**. Este tipo de cifrado no es simétrico, pues cuenta con dos tipos de clave: la clave pública, utilizada para cifrar la información, y la clave privada, utilizada para descifrarla.

Otro elemento importante en SSL es el **certificado digital**, imprescindible para asegurar la vinculación entre nuestra entidad (es decir, nosotros como propietarios del sistema) y la clave

pública asociada. Los certificados digitales contienen información sobre la entidad propietaria, como su nombre, su dirección y correo, la autoridad certificadora que generó el certificado, su firma digital cifrada mediante clave privada, y la clave pública de la entidad.

Su **funcionamiento** es transparente al usuario. Si nos centramos en el caso de nuestra aplicación, podría sintetizarse en estos cuatro pasos:

1. El cliente, o sea la aplicación móvil, solicita identificación de nuestro servidor.
2. Como respuesta, el servidor muestra al cliente el certificado digital.
3. El cliente verifica la validez del certificado con una serie de comprobaciones, que consisten en verificar la vigencia, la integridad y el emisor del mismo. Para comprobar que la integridad del certificado no se ha alterado, se descifra la firma digital a través de la clave pública de la autoridad certificadora, y se compara con la firma generada en el certificado.
4. Si el certificado digital es válido, el cliente confiará en el servidor. Finalmente se establecerá una comunicación segura cifrada.

En la figura Code Listing 5.8 se muestra la implementación de SSL en el servidor. Como requisito para establecer el uso del protocolo, es necesario disponer de dos ficheros: uno con la clave privada y otro con el certificado, que contendrá la clave pública. Ambos deberán ser generados por una entidad certificadora de confianza, aunque en este caso nosotros seremos la entidad.

Como vemos en la figura Code Listing 5.8, para **implementar SSL** especificamos en primer lugar el puerto en el que escuchará el servidor https. Después, inicializamos una variable 'options' con el contenido del fichero de la clave privada ('key') y el del certificado ('cert'). A continuación creamos el servidor HTTPS utilizando el paquete 'https' de NodeJS, pasando como parámetros la variable 'app' creada con *Express* y la variable 'options', y por último hacemos que el servidor escuche en el puerto especificado.

5.5.2. Token de autenticación

HTTP es un protocolo sin estado, por lo que es necesario implantar algún mecanismo de control de sesión en nuestro sistema. Para ello hemos utilizado la especificación JSON Web Token (JWT)[4]. En este caso, el control de sesión se realiza mediante un token de autenticación, que se genera en el momento en que el usuario inicia sesión en la aplicación, y se comprueba su validez cada vez que el usuario intenta realizar una petición a cualquier servicio.

Existe un módulo en NodeJS que realiza la codificación y decodificación del token, llamado jwt-simple[5], y su uso es muy sencillo. Por un lado, para generar el token, creamos una variable llamada payload para la cual especificamos el nombre del usuario, el momento de creación y el momento de caducidad, y se la pasamos como parámetro a la función ***encode*** de jwt-simple.

[4]https://tools.ietf.org/html/rfc7523 (Último acceso 12-02-2017)
[5]https://www.npmjs.com/package/jwt-simple (Último acceso 12-02-2017)

Podemos verlo en la figura Code Listing 5.9. También pasaremos como parámetro un token de secreto, almacenado en el servidor en este caso.

El servidor enviará al usuario el token generado como respuesta del servicio login, y este lo almacenará en la variable localStorage del dispositivo móvil. Al realizar una petición a cualquier servicio, ese token será decodificado mediante el método ***decode*** de jwt-simple para comprobar su validez. Si el token no ha expirado aún, el servidor extraerá el nombre de usuario del payload decodificado para que el usuario autenticado pueda acceder a sus documentos. En caso de haber caducado, se deberá generar un nuevo token.

5.5.3. Sanitización de entradas

Otra medida de seguridad que hemos adoptado es la de sanitizar los datos de entrada de las peticiones a nuestros servicios. Si no filtramos las entradas somos propensos a ataques de inyección de código[6], muy fáciles de realizar pero también de evitar.

Este tipo de ataques consisten en la inserción de un pequeño fragmento de código en uno de los parámetros de entrada del servicio, el cual será ejecutado por el servidor. Una práctica muy común es realizar inyección de código para recuperar información almacenada en base de datos, como pueden ser las contraseñas de todos los usuarios registrados.

Para filtrar en el servidor los datos recibidos por las, se ha utilizado el paquete **sanitizer** de NodeJS[7]. Utilizamos el módulo como se indica en la figura Code Listing 5.10:

De esta forma "escapamos" todo tipo de caracteres y términos que puedan terminar en la ejecución de cualquier código no deseado.

5.6. Programación en el lado del cliente o Frontend

Para termina, hablemos de la arquitectura del cliente, implantado a través de la tecnología AngularJS. Podemos encontrarnos tres agrupaciones de módulos, los cuales vamos a describir a continuación:

- **Vista principal**: Implementada en el fichero index.html. Se encarga de compilar la aplicación, incorporando para ello la ejecución de algunos scripts pertenecientes a Cordova, el módulo de los controladores, las rutas de navegación de la aplicación y el resto de vistas (llamadas Templates).
- **Templates**: Cada ventana o vista de la aplicación se desarrolla en un fichero html distinto. Distinguimos las siguientes:
    - Menú lateral

[6]https://www.owasp.org/index.php/Inyecci%C3%B3n_de_C%C3%B3digo (Último acceso 12-02-2017)
[7]https://www.npmjs.com/package/sanitizer (Último acceso 12-02-2017)

    - Login
    - Lista de etiquetas
    - Lista de documentos
    - Contenido de un documento
    - Cámara del dispositivo
    - Selección de una etiqueta entre las sugerencias de LDA
    - Selección de un nuevo título para un documento

- **Controladores**: Desarrollados en el módulo controller.js. Hemos implementado uno por vista, y cada uno de ellos constará de una serie de variables que se inicializarán al cargar la vista y de un conjunto de funciones que podremos invocar en el ámbito de la misma vista.
- **Rutas**: El fichero app.js define para cada ruta de navegación una única vista html y un controlador que se ejecutará en el momento de cargar una vista.

La arquitectura sobre la que está construida la aplicación móvil incorpora muchísimos más ficheros, como ficheros de estilo css, módulos internos de Cordova, archivos de gestión de dependencias bower, etc. Pero como estamos desarrollando el cliente a través de los *frameworks* AngularJS e Ionic, no tenemos que preocuparnos por generar ninguno de ellos. Tan solo hemos tenido que implementar el contenido de cada vista y su funcionamiento.

Ionic generará también los ficheros binarios necesarios para que la aplicación funcione sobre la plataforma móvil en la que deseemos ejecutar el cliente. Para generar una apk, por ejemplo, conectaremos al ordenador el dispositivo Android sobre el que queremos ejecutar la aplicación y lanzaremos el comando build de Ionic. Automáticamente tendremos nuestra aplicación funcionando.

```
var User = require('../models/user.js');
...

login = function(req, res) {

   var userName = req.body.user;
   var password = req.body.password;

   User.findOne({user: userName}, function(err, user) {
      if(!user){
         return res
            .status(401)
            .send({error: 'Bad credentials.'});
      }

      if(!err) {
         if(user.password == password) {
            return res
               .status(200)
               .send({token: createToken(userName)});
         } else {
            return res
               .status(401)
               .send({error: 'Bad credentials.'});
         }

      } else {
         return res
            .status(500)
            .send({error: 'Internal server error: ' + err.message})
               ;
      }
   });
};
```

Code Listing 5.5: Implementación de la función login.

```
storeDocument = function(req, res, next) {
  ...

  var document = new Document({
    creation:   new Date(),
    user:       req.user
  });

  document.save(function(err) {
    if(err) {
      if(err.name == 'ValidationError') {
        return res
          .status(500)
          .send({error: 'Validation error: ' + err.message});
      } else {
        return res
          .status(500)
          .send({error: 'Internal server error: ' + err.message});
      }
    }
  });

  res.documentId = document._id;

}
```

Code Listing 5.6: Implementación de la función storeDocument.

```
mongoose.connect('mongodb://<ip>:27017/tfm', function(err, res) {
  if(err) {
    console.log('ERROR: connecting to Database. ' + err);
  } else {
    console.log('Connected to Database');
  }
});
```

Code Listing 5.7: Conexión con la base de datos Mongoose.

```
var app   = express(),
        https = require('https');
...

var port = process.env.PORT || 8000;

var options = {
    key:    fs.readFileSync('ssl/tfm.key'),
    cert:   fs.readFileSync('ssl/tfm.crt')
};

var httpsServer = https.createServer(options, app);
httpsServer.listen(port);
```

Code Listing 5.8: Implementación de SSL en el servidor

```
var jwt = require('jwt-simple');
...

createToken = function(user) {
  var payload = {
    sub: user
    ,iat: moment().unix()
    ,exp: moment().add(14, "days").unix()
  };
  return jwt.encode(payload, tokenSecret);
};
```

Code Listing 5.9: Generación del token de autenticación

```
var sanitizer = require('sanitizer');
...

sanitizer.escape('input');
```

Code Listing 5.10: Sanitización de entradas.

Capítulo 6

Pruebas

Al haber empleado la metodología de desarrollo TDD, no existe una fase de pruebas unitarias posterior a la fase de implementación del código, pues para finalizar con éxito el desarrollo del mismo hemos tenido que desarrollar una serie de test que finalmente acaben pasando a verde. Para implementar una batería de test unitarios se han utilizado dos herramientas, llamadas **QUnit** y **Karma**.

QUnit se trata de un *framework* que nos ayuda a **diseñar nuestros test unitarios**, de la misma forma que JUnit lo hace para el lenguaje de programación Java. Como siempre, existe un módulo para integra la herramienta con el servidor NodeJS, llamado qunit[1].

Para **ejecutar los test unitarios** que hemos desarrollado con QUnit utilizamos Karma, una herramienta denominada como *test runner*. Cada vez que desarrollemos un test guardemos los cambios, Karma chequeará si los test unitarios han pasado con éxito. Nos indicará como SUCCESS (en verde) los que funciona, y como FAIL (en rojo) los que han fallado. Karma se integra con NodeJS a través del módulo karma[2].

En cuanto a las **pruebas funcionales**, se han realizado únicamente las necesarias para verificar que la aplicación funciona correctamente para casos de usos normales, y no para casos extremos. Los casos de prueba que se han contemplado han sido los siguientes:

- Inicio y cierre de sesión con dos usuarios distintos.
- Inicio de sesión introduciendo datos erróneos.
- Recuperación de etiquetas con dos usuarios distintos.
- Recuperación de documentos asignados a una etiqueta, a través de la sesión de dos usuarios distintos.
- Recuperación del contenido de un documento de texto, desde la sesión de dos usuarios distintos.

[1] https://www.npmjs.com/package/qunit (Último acceso 13-02-2017)
[2] https://www.npmjs.com/package/karma (Último acceso 13-02-2017)

- Escaneado de documentos para dos usuarios distintos.
- Verificación de la persistencia de los datos una vez cerramos sesión y volvemos a iniciarla.
- Acceso a los servicios una vez el token de autenticación ha sido expirado.

Capítulo 7

Conclusiones

Para finalizar la elaboración de este proyecto, vamos a terminar realizando una serie de conclusiones, comentando qué aporta nuestro sistema, cuál es mi valoración personal y algunas dificultades encontradas. Finalizaremos dichas conclusiones proponiendo una serie de puntos a desarrollar como trabajo futuro.

7.1. Aportaciones

Como aportaciones, acabamos de desarrollar un sistema que unifica varias fases de un procedimiento que normalmente desempeñamos a través de distintas aplicaciones. Nuestra aplicación, en un mismo flujo, es capaz de escanear un documento, extraer su contenido, almacenarlo en *la nube* y finalmente mostrar una lista de posibles temáticas relacionadas con el mismo. Y gracias a su soporte multiplataforma, podremos realizar dicha operación desde cualquier dispositivo móvil que llevemos encima, estemos donde estemos.

7.2. Valoración personal y dificultades encontradas

Como valoración personal, este trabajo me ha abierto las puertas a un nuevo paradigma de desarrollo, que es el de la programación web a través de JavaScript. Gracias a ello, he mejorado una serie de habilidades como *Desarrollador full-stack*, que de otra forma o en otro entorno de trabajo no hubiese sido posible. Esto también ha supuesto una gran dificultad para mí, ya que las diferencias de este paradigma con respecto a la programación orientada a objetos son considerables.

Por otro lado, desarrollar un proyecto de principio a fin, pasando por todas y cada una de sus fases, ha sido muy enriquecedor. Y como conclusión final, considero que he adquirido una serie de competencias muy valiosas para enfrentarme a proyectos de este tipo en un entorno laboral.

7.3. Trabajo futuro

Como trabajo futuro, se propone un entrenamiento del sistema de clasificación. En este punto tenemos la infraestructura de digitalización y clasificación de contenidos desarrollada, realizándose la clasificación a través de LDA como primera aproximación. Queda pendiente alimentar al sistema con la digitalización de múltiples documentos, y la clasificación de los mismos de forma manual pero asistida por LDA, es decir, seleccionando por el autor la etiqueta que más se ajuste a un documento entre una lista de sugerencias dadas por el algoritmo.

Deberán realizarse pruebas variando los parámetros de entrada del algoritmo, observándose cuáles son las cantidades óptimas para el número de categorías a sugerir y el número de términos en los que se basa cada una. Investigaremos también qué alternativas hay al algoritmo LDA, comparándolas con este en función a los resultados de ambos.

Y en cuanto al diseño del sistema, queda pendiente desarrollar servicios que mejoren la funcionalidad, como por ejemplo un servicio de registro de usuarios.

Referencias

[1] D. Astels, Test Driven Development: A practical guide, Prentice Hall Professional Technical Reference, 2003. VII, 21

[2] M. Cohn, User stories applied: For agile software development, Addison-Wesley Professional, 2004. 9

[3] T. OWASP, 10 2010, The Ten Most Critical Web Application Security Risks. 30

[4] S. Xanthopoulos, S. Xinogalos, A comparative analysis of cross-platform development approaches for mobile applications, in: Proceedings of the 6th Balkan Conference in Informatics, ACM, 2013, pp. 213–220.

Apéndice A

Interfaz de usuario

Figura A.1: Formulario de inicio de sesión.

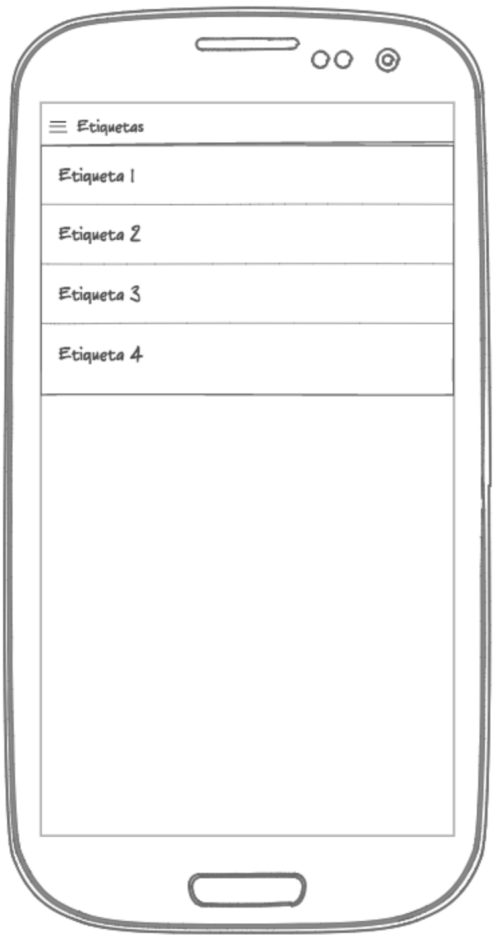

Figura A.2: Lista de las etiquetas asignadas a todos los documentos.

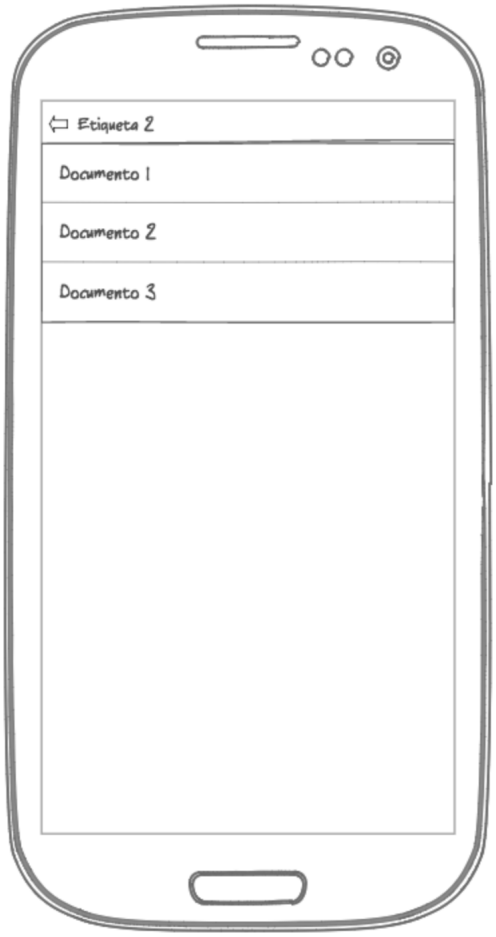

Figura A.3: Lista de documento correspondientes a la ‘Etiqueta 2’.

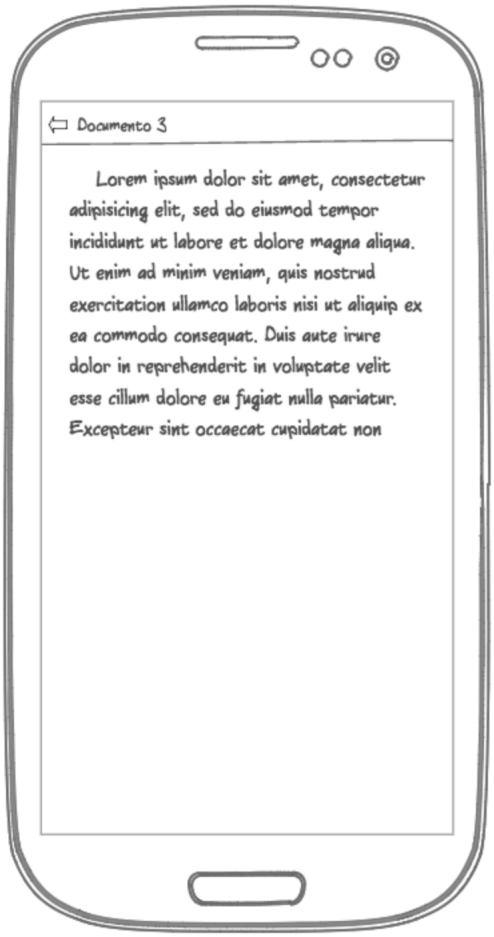

Figura A.4: Contenido de 'Documento 3'

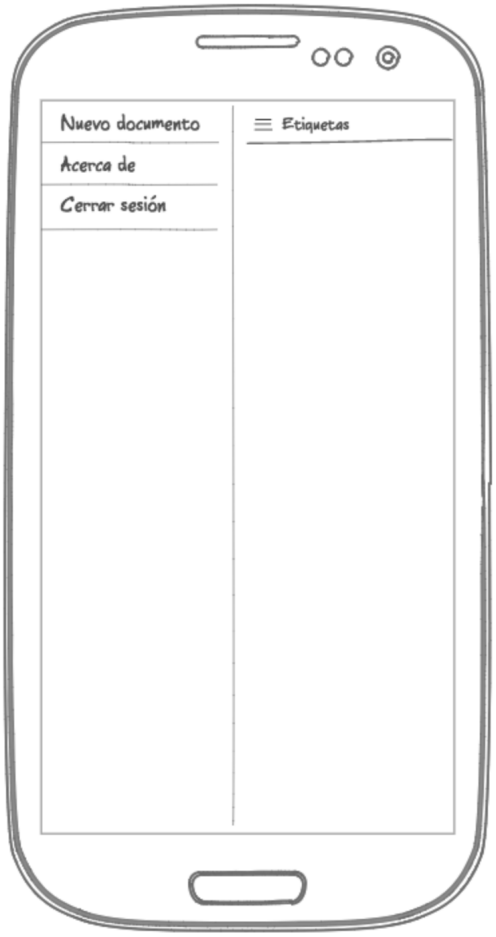

Figura A.5: Menú de la aplicación.

Figura A.6: Vista de cámara.

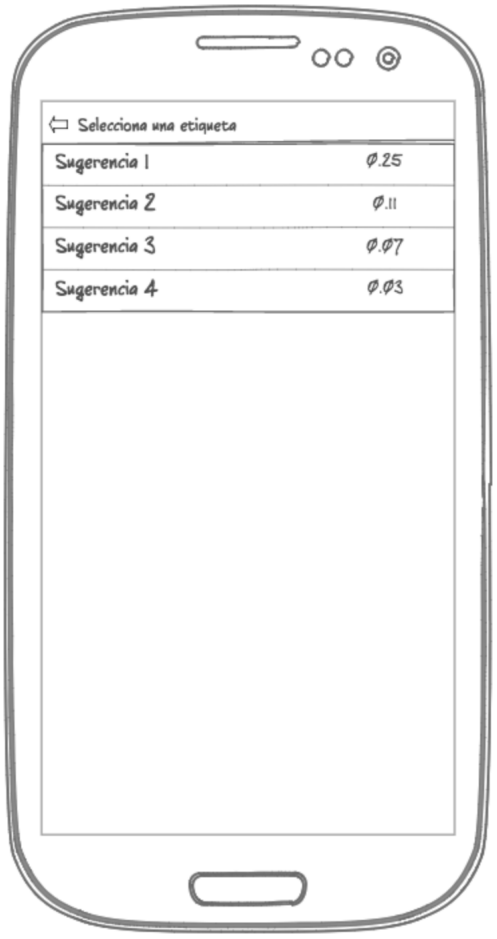

Figura A.7: Lista de etiquetas sugeridas por la función de etiquetado.

Figura A.8: Selección de título.