Universitat Oberta de Catalunya

# *PetFinder*: el buscador de mascotas perdidas

Memoria de Proyecto Final de Máster

**Máster universitario en Desarrollo de Sitios y Aplicaciones Web**

**Autor: Ricard Molina Ferret**

Consultor: Víctor Cuervo

Profesor: Carlos Casado Martínez

Enero de 2021

# Créditos/Copyright

Todo el contenido de este documento de memoria, así como el código fuente y la aplicación desarrollada en ocasión de este proyecto están sujetos a la siguiente licencia.

Esta obra está sujeta a una licencia de Reconocimiento-NoComercial SinObraDerivada 3.0 España de Creative Commons

# Dedicatoria/Cita

*“La grandeza de una nación y su progreso moral puede ser juzgado por la forma en que sus animales son tratados”*. Mahatma Gandhi.

# Abstract

En este proyecto se ha realizado una aplicación –*PetFinder*–, desde la definición de la estructura de datos hasta la aplicación cliente pasando por la aplicación *backend*, con el objetivo de ayudar a gestionar los casos de mascotas perdidas. La aplicación, orientada preferentemente a los dispositivos móviles, pretende ayudar a los usuarios mediante la solidaridad de estos, a encontrar las mascotas que se hayan perdido con el uso de la geolocalización, para ayudar a precisar el ámbito de la búsqueda.

La realización de este proyecto también supone una oportunidad para desarrollar una aplicación moderna basada en tecnologías como Angular, Spring Boot y MongoDB ampliamente usadas a nivel profesional.

Palabras clave: web, web app, animals, pet, searcher, backend, frontend, aplicacion

# Abstract (english version)

In this project an application -PetFinder– has been made, from the definition of the data structure to the client application through the backend application, in order to help manage the cases of lost pets. The application preferably aimed at mobile devices aims to help users through their solidarity to find the pets that have been lost with the use of geolocation, to help specify the scope of the search.

Carrying out this project is also an opportunity to develop a modern application based on technologies such as Angular, Spring Boot and MongoDB widely used at a professional level.

Keywords: web, web app, animals, pet, searcher, backend, frontend, aplicación

# Agradecimientos

Un sentido agradecimiento a las profesoras que he tenido el placer de tener a lo largo de la formación que concluyo con la entrega de este proyecto, así como a todo el personal que conforma la UOC.

El modelo de organización con el que trabajáis, la flexibilidad –sobre todo en estos tiempos complicados por la COVID– que ofrecéis y la reactividad ante las dudas y comentarios en los diferentes foros hacen que sea un auténtico placer estudiar y aprender en la UOC.

# Índice

1. Introducción....................10
2. Descripción....................11
3. Objetivos....................12
  3.1 Principales....................12
  3.2 Secundarios....................12
4. Escenario....................13
5. Contenidos....................16
  5.1 Aplicación Backend....................16
  5.2 Aplicación Frontend....................16
  5.3 Despliegue....................17
6. Metodología....................19
7. Arquitectura de la aplicación....................20
  7.1 Base de datos....................20
  7.2 Aplicación *backend*....................21
  7.3 Aplicación *front-end*....................23
  7.4 Despliegue....................26
8. Plataforma de desarrollo....................29
9. Planificación....................30
11. APIs utilizadas....................32
  11.1 Servicio *Facebook*....................32
  11.2 Servicio *Google*....................35
12. Diagramas UML....................38
  12.1 Casos de Uso....................38
  12.2 Clases....................42
  12.3 Secuencia del Social Login....................43
13. Prototipos....................45
14. Perfiles de usuario....................49
15. Usabilidad/UX....................50
16. Seguridad....................53
17. Tests....................55
18. Versiones de la aplicación/servicio....................56

19. Requisitos de instalación/implantación/uso ........ 57
20. Instrucciones de implantación ........ 58
21. Instrucciones de desarrollo ........ 59
21.1 Aplicación Backend ........ 59
21.2 Aplicación Frontend ........ 60
22. Bugs ........ 61
23. Proyección a futuro ........ 62
24. Presupuesto ........ 63
25. Análisis de mercado ........ 64
26. Marketing y Ventas ........ 65
26.1. Métricas de la aplicación ........ 66
27. Conclusiones ........ 68
Anexo 1. Entregables del proyecto ........ 70
Anexo 2. Libro de estilo ........ 71
Anexo 3. Glosario ........ 72

# 1. Introducción

Decía el escritor y premio Nobel francés Anatole France que todos tenemos un parte de alma que permanece dormida hasta que se ama a un animal. En mi opinión, no se equivocaba para nada.

En España, según datos de la ANFAAC (Asociación Nacional de Fabricantes de Alimentos para Animales de Compañía), el año 2019 se censaron más de 28 millones de mascotas en aproximadamente el 40% de los hogares.

De estos 28 millones de animales de compañía, los diferentes tipos de perros suman 6.733.097 ejemplares y las diferentes especies de gatos suman 3.795.139 censados. Aunque ni los gatos ni los perros son el tipo de mascota más común –según los mismos datos hay más de 8 millones de peces contabilizados–, son estos dos tipos de mascotas en los que nos vamos a centrar por una razón: los perros y los gatos son las mascotas que más a menudo se pierden o se escapan.

A pesar de que ya existen opciones disponibles que tratan la problemática de las mascotas perdidas de forma incluso más detallada que la aplicación que se propone en este proyecto, considero muy interesante la oportunidad de desarrollar de cero una aplicación que persigue el objetivo de ayudar a animales perdidos y, que, al mismo tiempo, me permite trabajar con tecnologías innovadoras como Angular, Spring Boot o MongoDB y, al mismo tiempo, aplicando patrones de arquitectura aplicativa usados a nivel profesional, como el despliegue de aplicaciones en contenedores y el uso de orquestadores (Docker Compose) para gestionar el despliegue desde un solo fichero con una sola instrucción.

# 2. Descripción

En este Proyecto de Final de Máster se propone la concepción de una aplicación –*PetFinder*– para ayudar a encontrar mascotas –perros o gatos– perdidos.

Para realizar esta aplicación se ha optado por un patrón de arquitectura desacoplado, con una aplicación de servidor y una aplicación cliente ligera que se comunicaran entre ellas mediante el consumo de servicios REST.

Esta opción es un poco más laboriosa que la clásica aplicación monolítica que envía directamente contenido HTML al navegador, pero permite separar mejor las diferentes responsabilidades de cada parte de la aplicación (la aplicación cliente presenta la información mientras que la aplicación servidor gestiona la información y la procesa) y permite, al mismo tiempo, escalar de forma independiente cada parte de la aplicación en función de si es o no necesario (con el ahorro de costes que eso conlleva).

Desde el punto de vista técnico, para la elaboración de la base de datos se ha optado por el uso de MongoDB; para la concepción de la aplicación *backend* o de servidor se ha usado Java con *Spring Boot* y la aplicación cliente se ha construido con las premisas de *mobile-first* y usando el *framework JS* Angular para la realización de una *SPA (Single Page Application)* que funcione en el navegador.

# 3. Objetivos

En este punto se exponen los diferentes objetivos propuestos para la realización de este Proyecto de Fin de Máster.

## 3.1 Principales

En este apartado se exponen los objetivos principales que se persiguen en este proyecto y que deben completarse necesariamente al final del proyecto.

- Analizar e identificar los actores y las necesidades que la aplicación debe resolver.
- Diseñar y realizar una aplicación backend con la lógica necesaria para responder a las necesidades y casos de uso para los que esta aplicación dará servicio.
- Diseñar y desplegar una aplicación *frontend* para consumir los servicios ofrecidos por la aplicación backend.

## 3.2 Secundarios

En este apartado los objetivos secundarios, esto es, objetivos con un menor nivel de importancia y cuya asunción puede ser variable al final del proyecto.

- Aplicar los conocimientos de diseño de interfaces web adquiridos en este Máster.
- Investigar las diferentes opciones de despliegue para cada aplicación y entorno maximizando el ahorro y el costo de despliegue: Uso de contenedores Docker.
- Geoposicionamiento. Uso de las tecnologías espaciales y de geoposicionado para dotar a la aplicación de mayores funcionalidades.
- Notificaciones. Integración de la aplicación con servicios de mensajería de terceros para el envió de notificaciones en tiempo real.
- API de terceros. Integración de la aplicación con APIs de terceros, como Google o Facebook, para el inicio de sesión, la creación de usuarios, el uso de mapas, localizaciones u otras funcionalidades.
- Calidad. Aplicación de pruebas unitarias y funcionales para asegurar el funcionamiento deseado de las partes de las aplicaciones.

# 4. Escenario

Durante estos últimos años –a partir de 2010 en adelante– el diseño de contenido y aplicaciones web ha avanzado muy rápidamente, evolucionando y dejando atrás algunos patrones de arquitectura para profundizar en otros más modernos y que han permitido que, hoy en día, las aplicaciones web estén presentes en casi cualquier dispositivo conectado: ordenadores, tabletas, smartphones, relojes, etc.

A finales de los años 90, las páginas web eran esencialmente aparadores con contenidos estáticos servidos directamente des de los servidores web: no había, a penas, interacción ni contenido personalizable en función del usuario o de otras variables y el único elemento dinámico lo aportaba el uso de tecnologías como JavaScript o Flash. Tecnologías por entonces muy heterogéneas que cada navegador interpretaba de forma diferente.

Este tipo de páginas web (el concepto de página, en sí, ya evoca un alto grado de contenido estático, si lo comparamos con el de sitio o aplicación) evolucionaron para ganar un cierto dinamismo con el auge de las tecnologías de interacción cliente – servidor, fundamentalmente AJAX – *Asynchronous Javascript And XML* por sus siglas en inglés– y a la introducción y mejora constante de un elemento como CSS (Cascading Style Sheets) para definir los estilos de los nodos HTML de un sitio web.

Sin embargo, aun con el uso de AJAX y la comunicación asíncrona entre el cliente y el servidor, las aplicaciones continuaban siendo monolíticas: es decir, el servidor se encargaba de realizar los procesos lógicos y, además, generaba las vistas HTML que enviaba al cliente en cada respuesta del servidor. Este enfoque, si bien sigue siendo válido hoy en día y tiene algunas ventajas como el coste de su mantenimiento con aplicaciones de tamaño pequeño o su facilidad de despliegue, plantea varios inconvenientes que limitan –o encarecen el coste– en caso de que la aplicación crezca.

Las aplicaciones monolíticas, padecen todos los problemas ligados al alto acoplamiento de código: las dificultades para trazar y gestionar errores aumenta conforme aumenta el tamaño de la aplicación, no es posible escalar (aumentar o reducir los recursos asignados a una aplicación) de forma separada diferentes partes de una misma aplicación y, en este modelo el servidor realiza la mayor parte del esfuerzo, puesto que tiene que enviar el código HTML al cliente y cliente se limita a interpretarlo, agregar estilos y los scripts de interacción JS que lleve añadidos. El cliente apenas tiene responsabilidad en este esquema.

Este paradigma evoluciona, entre otros, con la vertiginosa evolución en el tipo y la potencia de procesamiento de los clientes: el aumento de potencia de los ordenadores, la aparición de tabletas y el

cambio de concepto del teléfono, que pasa a ser un *smartphone*, un ordenador en nuestros bolsillos y, consecuentemente, con el aumento exponencial en el uso de los servicios y aplicaciones web que explota en estos años con la llegada de grandes plataformas sociales como *Facebook* (2004) o *Twitter* (2006).

Esta evolución de la forma en que se hacen las aplicaciones web lleva a cambiar el enfoque: se dejan de hacer aplicaciones web de forma monolítica, para pasar a realizar y consumir servicios web. Se popularizan las APIs (*Application Programming Interface,* por sus siglas en inglés). La creación de APIs es un nuevo paradigma en que el producto final no es un documento HTML interpretable por un navegador sino un formato de texto, normalmente JSON (JavaScript Object Notation, por sus siglas en inglés) o XML, y cuyo destinatario son otras aplicaciones o programas que usaran la información que tu API genera.

El protocolo de red sobre el que funcionan las API es el mismo que ya se usaba antes para el envió de páginas web desde el servidor: HTTP. Sin embargo, éste se actualiza para ser más eficaz y seguro. Se refuerza el uso de los verbos (GET, PUT, POST, DELETE, PATCH, etc) para acompañar las peticiones al servidor y saber de este modo que tipo de operación pretende realizar una determinada petición al servidor.

En paralelo al desarrollo del protocolo HTTP, también se ha extendido el uso de certificados para proteger el intercambio de información entre cliente y servidor. Es lo que se conoce como HTTPS. Se trata de un cifrado del contenido enviado en HTTP con el protocolo SSL/TLS consistente, básicamente, en el uso de certificados para proteger la información y, también, para validar que el sitio web que estamos visualizando es el auténtico y que no se está intentando usurpar su identidad o engañar al usuario.

El uso de las API permite separar las aplicaciones según sus responsabilidades (tratamiento de datos, presentación de la información, actualización de usuarios, transacciones bancarias, etc) y aligerar enormemente el tráfico de peticiones y respuestas entre servidores o entre un servidor y un cliente, puesto que se deja de enviar largos y complejos documentos HTML, para enviar texto plano con formato JSON o XML, pero texto, después de todo. El ahorro es notorio.

Siguiendo la tendencia de las API, se puede llegar a tener aplicaciones extremadamente desacopladas, en que cada dominio de la aplicación (gestión de usuarios, transacciones, generación de documentos, etc.) es, en realidad, una aplicación independiente que se comunica con las otras mediante el uso de APIs u otras tecnologías de mensajería. Este tipo de arquitectura se conoce como arquitectura de microservicios y, aunque permite mucha flexibilidad a la hora de escalar las aplicaciones y responder a picos de demanda específicos, también presenta muchos retos en lo concerniente a mantenibilidad y

despliegue, puesto que no gestionamos una aplicación que ofrece una API sino que se trata de un ecosistema de aplicaciones.

Para la realización de la aplicación *PetFinder*, adoptaremos el paradigma de API, por el que una aplicación de servidor –*backend*– ofrecerá unos servicios mediante una API y una aplicación que se ejecutará en el navegador del cliente (ordenador, tableta, smartphone, reloj, etc.) consumirá y se encargará de presentar la información al usuario final.

Otra de las ventajas de este tipo de arquitectura es que una API puede ser consumida por varios clientes: otras aplicaciones web que quieran usar o mostrar la información que tu generas, o otras aplicaciones para otros dispositivos, como aplicaciones nativas para smartphone, relojes inteligentes o incluso para aplicaciones de escritorio.

# 5. Contenidos

Los contenidos sobre los que se va a trabajar y que se van a producir durante este Proyecto son tres: las dos aplicaciones que, juntas, conformaran la aplicación web *PetFinder* y el procedimiento de despliegue de la aplicación, para que las dos aplicaciones trabajen juntas.

## 5.1 Aplicación Backend

Para la gestión de toda la lógica de la aplicación: gestión de usuarios, seguridad, gestión de las mascotas, notificaciones, etc se ha realizado una aplicación usando la tecnología Spring Boot (Java) que se ejecutara sobre un servidor web Tomcat autocontenido en la aplicación. Esta aplicación constituye el *backend* de *PetFinder* y se encargara, entre otros, de:

- **Seguridad**. Verificación de usuarios y roles para autorizar la realización de acciones en la aplicación. La seguridad es una responsabilidad muy importante en toda aplicación y, por ello, es también compartida con la aplicación cliente.
- **Gestión de usuarios.** Creación de usuarios en la aplicación, ya sea a través de terceros o a partir del registro de usuarios de la aplicación. Gestión de usuarios registrados con sus mascotas.
- **Mascotas.** Gestión de mascotas mediante acciones. Perdida del animal. Animal avistado. Animal encontrado, etc.
- **Notificaciones.** Envio de notificaciones a los usuarios en los casos que sea necesario.

El resultado final será una aplicación Java compilada en formato .jar con un servidor web Tomcat autocontenido para facilitar el arranque de la aplicación. Es lo que se llama un *fat jar.*

## 5.2 Aplicación Frontend

Para encargarse de la presentación de la información al usuario, se ha concebido una aplicación frontend, creada bajo las premisas del diseño *mobile first* y usando Bootstrap para lograr fácilmente comportamientos responsivos adaptados a las diferentes medidas de pantalla de los dispositivos más utilizados.

Se ha usado la versión 10 del *framework* Angular para la creación de una *SPA (Single Page Application*, por sus siglas en inglés*)* que gestione y presente la información y, en caso de interaccionar con la aplicación, muestre los cambios realizados por los usuarios.

El producto final de esta aplicación es un conjunto de ficheros HTML, CSS y JS que deben ser alojados en un servidor de contenido web (como Apache o NGINX) para poder ser consumidos. Es necesario apuntar que, para el desarrollo de la aplicación se usan tecnologías como Typescript o SCSS que, antes de generar los ficheros de la aplicación de producción son precompilados en JavaScript y CSS –respectivamente– y, por ello, no hay ficheros de estas extensiones en los ficheros finales.

## 5.3 Despliegue

Una vez, se han generado las versiones finales, o de producción, de cada una de las dos aplicaciones que conforman *PetFinder* es necesario pensar en como realizar el despliegue de ambas aplicaciones y sus dependencias (bases de datos, sistemas de mensajería, etc) de forma orquestada y coordinada procurando automatizar lo máximo posible el despliegue. Para ello, en este proyecto se ha propuesto usar Docker.

Docker es una tecnología muy poderosa para la virtualización y la creación de contenedores ideal para desplegar rápidamente contenedores creados previamente, lo que permite desplegar en producción contenedores de forma rápida y segura.

Con el objetivo de poder crear contenedores Docker para las aplicaciones anteriores, crearemos un fichero Dockerfile en la carpeta raíz de cada proyecto con las instrucciones necesarias para crear las imágenes y subirlas al repositorio de imágenes oficiales de Docker, DockerHub.

Una vez las imágenes se han cargado en el repositorio remoto de Docker, ya podríamos instanciarlas, crear un contenedor y ejecutarlo para probar nuestra aplicación. Sin embargo, para asegurar el despliegue correcto de la aplicación es necesario crear otro fichero llamado docker-compose.yml. Este fichero se encargará, cuando se ejecuta el comando "*docker-compose up*", de ejecutar de forma coordinada y secuencial los contenedores de las dependencias de nuestra aplicación backend, nuestra aplicación backend y, en último lugar, nuestra aplicación frontend (ya con su servidor web configurado para servir los ficheros de la aplicación).

Tras haber realizado estos pasos y creado los diferentes ficheros, será posible arrancar toda la aplicación *PetFinder*, desde la base de datos hasta el servidor de la aplicación frontend, con un solo comando de Docker y con la única condición que la maquina en que se quiera ejecutar la aplicación

tenga instalado Docker y disponga de conexión a Internet (necesaria, entre otras cosas, para recuperar las imágenes de los diferentes contenedores del repositorio remoto).

# 6. Metodología

Para la elaboración de este proyecto se han seguido o tenido en cuenta varias metodologías diferentes en función de la fase del proyecto.

En primer lugar, para la elaboración general del proyecto y su memoria se ha realizado un seguimiento continuo del estado del proyecto. Esta metodología permite la detección temprana de errores y permite al consultor o tutor orientar al alumno y guiarlo en ciertos pasos de la elaboración del proyecto. De otro modo, si el alumno no tuviera –o no usara– esta forma de trabajo, el resultado final del proyecto podría diferir bastante de lo esperado por el tribunal que lo valora y atiende a la presentación final.

Para la obtención y depuración de la idea de aplicación para este proyecto, se ha realizado un *brainstorming*, una lluvia de ideas con el objetivo de arrojar posibles necesidades que pudieran ser cubiertas con una aplicación web o con un sitio web. Ideas como realizar una aplicación web para monitorear el estado de la pandemia de COVID-19 o realizar una aplicación web que se integrara con la plataforma deportiva *Strava* para procesar sus datos y ofrecer otro tipo de presentación de los datos de cada usuario en esa plataforma aparecieron, aunque al final prevaleció la idea de crear una aplicación para mascotas perdidas, *PetFinder.*

La aplicación del seguimiento continuado del proyecto me ha permitido, en cierto modo, implementar una metodología basada en iteraciones en que cada entrega se entiende como una iteración, al final de la cual se entrega un producto y se reciben los errores y los puntos de mejora por parte del tutor. Dentro del desarrollo de la aplicación y, en general del proyecto, los errores comentados en una entrega son corregidos durante la iteración siguiente para reforzar la aplicación y el proyecto. Con ello, a medida que pasa cada iteración (o ciclo de trabajo) la aplicación se refuerza y el contenido del proyecto y de su memoria también.

# 7. Arquitectura de la aplicación

A continuación, se va a explicar la arquitectura seguida en los diferentes niveles de la aplicación, desde la base de datos hasta la aplicación frontend.

## 7.1 Base de datos

Como ya se ha indicado anteriormente, la base de datos elegida para el almacenamiento de datos es MongoDB, uno de los sistemas gestores de bases de datos no relacionales más utilizados por la industria y que basa su funcionamiento en el uso de objetos JSON para la transferencia de información (y, consecuentemente, para hacer las consultas).

En MongoDB, la unidad de información; lo que en sistemas relacionales conocemos como registro o línea en una tabla, se conoce como documento. Los documentos son de formato libre: no existe la noción de columna; sin embargo, todos los documentos almacenados en una colección –así es como se conoce las agrupaciones de documentos en Mongo – deben contener un campo único que sirva como clave primaria para indexar el contenido. En caso de no insertarlo, la base de datos crea automáticamente un campo *_id* y lo alimenta con un valor alfanumérico único.

Aunque MongoDB permite la creación de clústeres para que varias instancias remotas de MongoDB puedan trabajar conjuntamente, para las necesidades de este proyecto se ha considerado suficiente el uso de una sola instancia de base de datos MongoDB, con 5 colecciones.

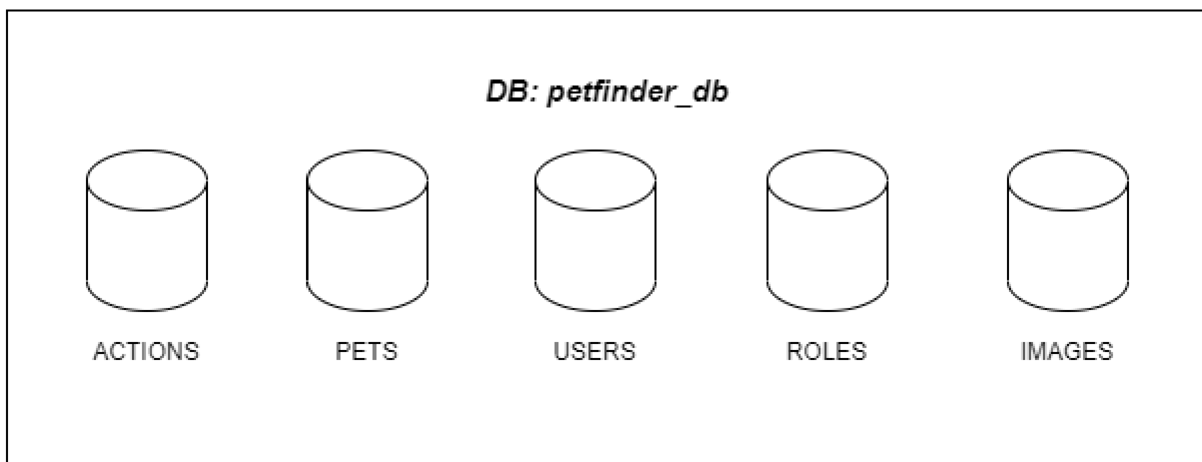



*Imagen: Colecciones en petfinder_db*

La colección “ACTIONS” sirve para almacenar las diferentes acciones realizadas sobre un animal: perdida (LOST), avistamiento (SIGHTING) y hallazgo (FOUND) con las informaciones respectivas.

La colección “PETS” actúa como piedra angular del esquema de datos y almacena la información ligada a la mascota. Esta colección contiene documentos (o, mejor dicho, referencias de documentos) de otras 3 colecciones: “ACTIONS”, “USERS” y “IMAGES”.

La colección “USERS” contiene los usuarios registrados en la aplicación con su información, así como el rol que se le atribuye (incluyendo documentos de la colección “ROLES”).

La colección “ROLES” es una colección muy pequeña que contiene los roles que gestiona la aplicación. Aunque, dado que solo gestionamos dos roles actualmente, la decisión de crear una colección se ha tomado pensando en futuras evoluciones de la aplicación en que puedan crearse nuevos roles.

Por último, la colección “IMAGES”, contiene la información de las imágenes de las mascotas que los usuarios suben a la aplicación. Es importante remarcar que las imágenes, como tal, no son almacenadas en la base de datos, sino que se procesan y se almacenan en el servidor de almacenaje en la nube *MinIO*.

En el diagrama de clases que se incluye en los capítulos siguientes se aprecia más claramente el esquema de datos y clases (mapeadas a colecciones de la base de datos) usado, así como las relaciones existentes entre ellas.

## 7.2 Aplicación *backend*

Para la realización de la aplicación backend, se ha utilizado Java, con *Spring Framework*. Aunque considero que Java no requiere de presentación puesto que es actualmente uno de los lenguajes de programación más utilizados en el mundo (sino el que más), sí que resulta más interesante explicar rápidamente que es *Spring Framework* y, más concretamente, *Spring Boot*, las librerías que se han utilizado para dar forma a la aplicación.

*Spring Framework* es un conjunto de librerías que tienen por objetivo abstraer el desarrollo de aplicaciones empresariales (cualquiera que sea el tipo: web, de servicios REST, microservicios, de escritorio, etc.) de las configuraciones de base y de entorno que son necesarias si se parte de Java

puro. Esta premisa permite que el desarrollo se centre en la implementación de la lógica de negocio de la aplicación, en lugar de pasar tiempo abordando cuestiones técnicas.

Esto se ha traducido en la creación de librerías dentro de Spring que simplifican y agilizan la implementación de funcionalidades de una aplicación: de este modo, por ejemplo, *Spring Security*, permite configurar y proteger una aplicación rápidamente y sin tener que implementar todos los controles que, de forma clásica se implementan. En el caso de PetFinder, tiene un peso especial *Spring Web* y *Spring Data Mongo*: la primera simplifica el enrutado de las peticiones hacia controladores e incluye un servidor Tomcat en la aplicación, y la segunda librería facilita el mapeo de entidades entre la base de datos y la aplicación, así como la realización de consultas a la base de datos mediante el uso de clases repositorio autogeneradas.

Para llevar al extremo el objetivo principal de Spring, se ha desarrollado *Spring Boot*. Esta librería se compone básicamente de clases de autoconfiguración que permiten lanzar y autoconfigurar muchas de las librerías que componen *Spring Framework*, de modo que el desarrollador, al crear una aplicación Spring Boot, obtiene una aplicación con un conjunto de configuraciones por defecto que permiten que la aplicación funcione. De este modo, el tiempo que se debe invertir en configurar la aplicación para que arranque es mínimo y el desarrollo puede centrarse en implementar la lógica de la aplicación.

En lo concerniente al patrón de arquitectura seguido para construir la aplicación backend, se ha tomado como referencia el modelo de arquitectura en 3 capas (*three-tier architecture* en inglés).

Este patrón clasifica todos los componentes –clases– de la aplicación en 3 grupos distintos: los componentes que generan y exponen los servicios REST –controladores–, los que implementan la lógica de la aplicación –servicios– y aquellos que se ocupan del acceso a la base de datos –típicamente, las entidades y los repositorios–.

Partiendo de este patrón, para desarrollar la aplicación se han realizado algunas adaptaciones para dar cabida a las dependencias externas de la aplicación.

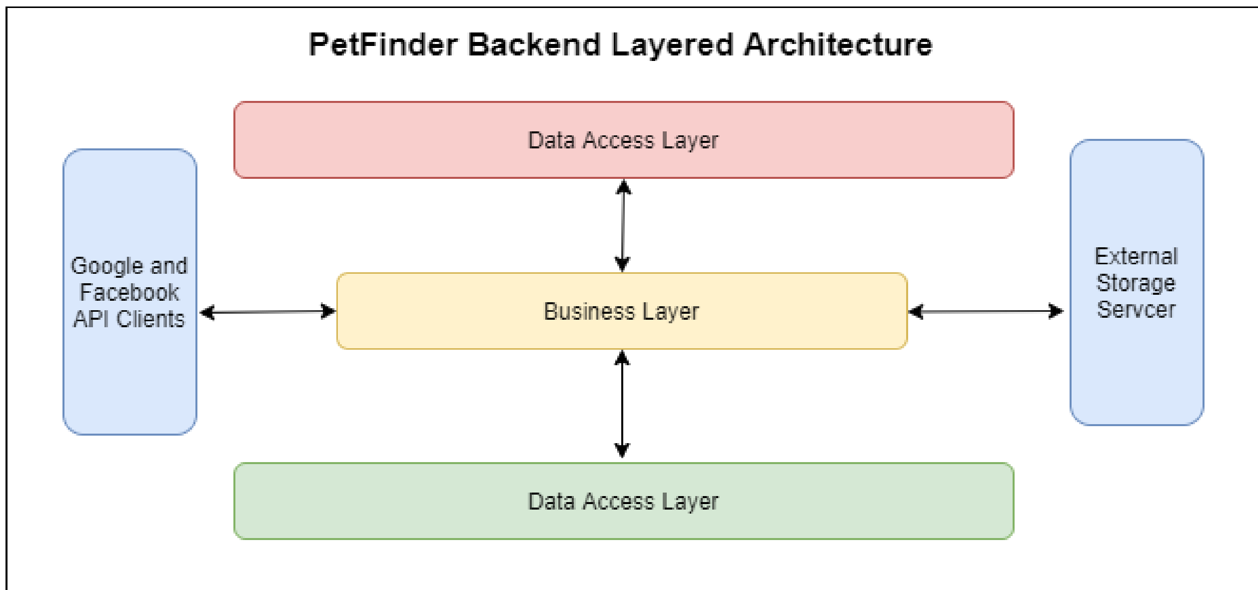



*Imagen: Arquitectura multi capa en PetFinder*

Como se ve en la imagen, al modelo de 3 capas se le han agregado dos elementos que podríamos llamar periféricos y que gestionan la conexión de las aplicaciones de terceros: Google y Facebook para el identificar al usuario que se ha autenticado mediante las redes sociales y el servidor de almacenaje en la nube (MinIO).

## 7.3 Aplicación *front-end*

Para crear la aplicación frontend de PetFinder se ha elegido la librería Angular en su versión 9 juntamente con la librería CSS *Bootrstrap 4* y la librería de componentes NGX-Bootstrap.

Angular es una librería para el desarrollo de aplicaciones web SPA (*Single Page Application*, por siglas en ingles) en TypeScript creado por Google en 2016 –y mantenido desde entonces– y que se encuentra, en el momento de redactar este documento, en la versión 10.1.4. Se trata con *ReactJS* de las dos opciones más utilizadas hoy en día para la creación de aplicaciones frontend en JavaScript.

En lo que respecta a *Bootstrap*, se trata de la librería de componentes CSS y JS más utilizada del mercado. Fue creada por *Twitter* en 2011 y hoy en día es una librería con licencia *open source* que se encuentra en su versión 4.3.1. Aunque *Bootstrap* es una librería que incluye contenido y dependencias en Javascript (*jQuery y PooperJS*) para su funcionamiento normal, en este proyecto se ha optado por usar únicamente los elementos CSS de la librería y combinarlos con los componentes propuestos por NGX-Bootstrap que se han concebido especialmente para ser usados

en aplicaciones Angular y, lo más importante, que no contienen como dependencias librerías como jQuery, cuyo uso contraviene los principios originales de los framework de aplicaciones JS modernos.

En términos de arquitectura, Angular se basa en componentes para la creación de aplicación y no sigue estrictamente ningún patrón de arquitectura convencional. No obstante, tienen ciertos elementos que se podrían entender como una adaptación del patrón MVC (modelo-vista-controlador).

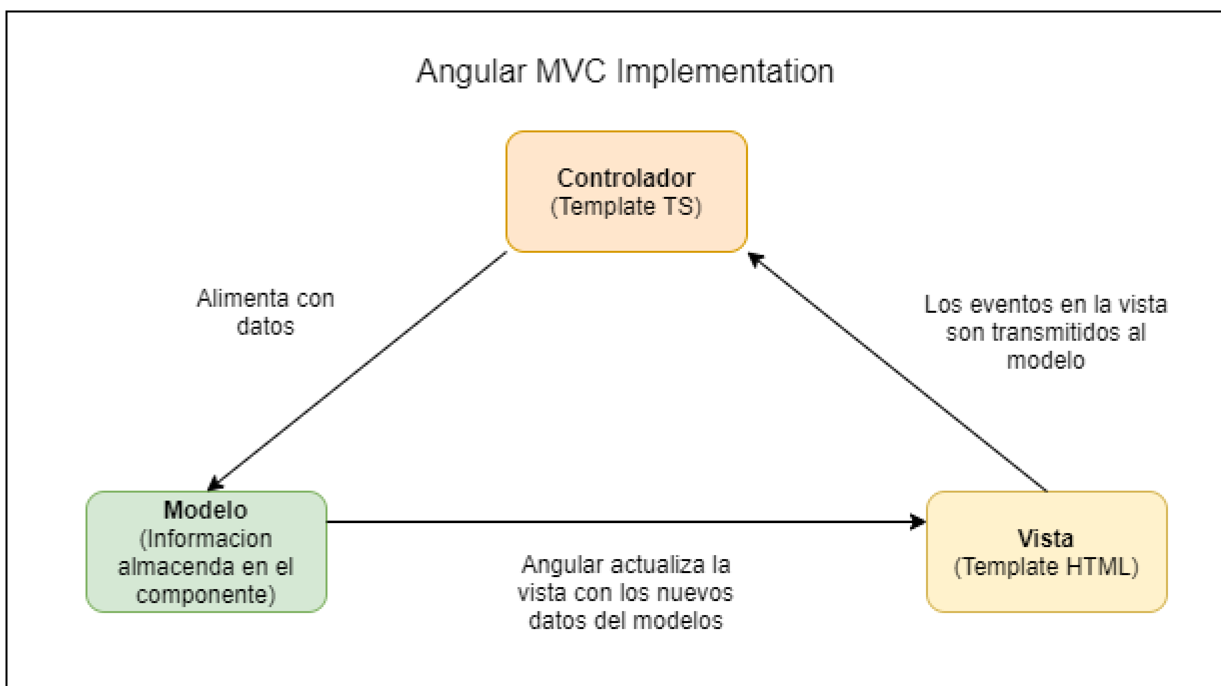



*Imagen: Implementación de MVC en Angular*

En relación con la arquitectura implementada en la aplicación cliente, se basa en el uso de servicios y los elementos reactivos de la librería RxJS para la gestión de la información, descartando, de este modo, el uso de Redux, puesto que se trata de una aplicación pequeña y la su uso añadiría complejidad innecesariamente al desarrollo.

A continuación, se muestra el diagrama de módulos que componen la aplicación Frontend.

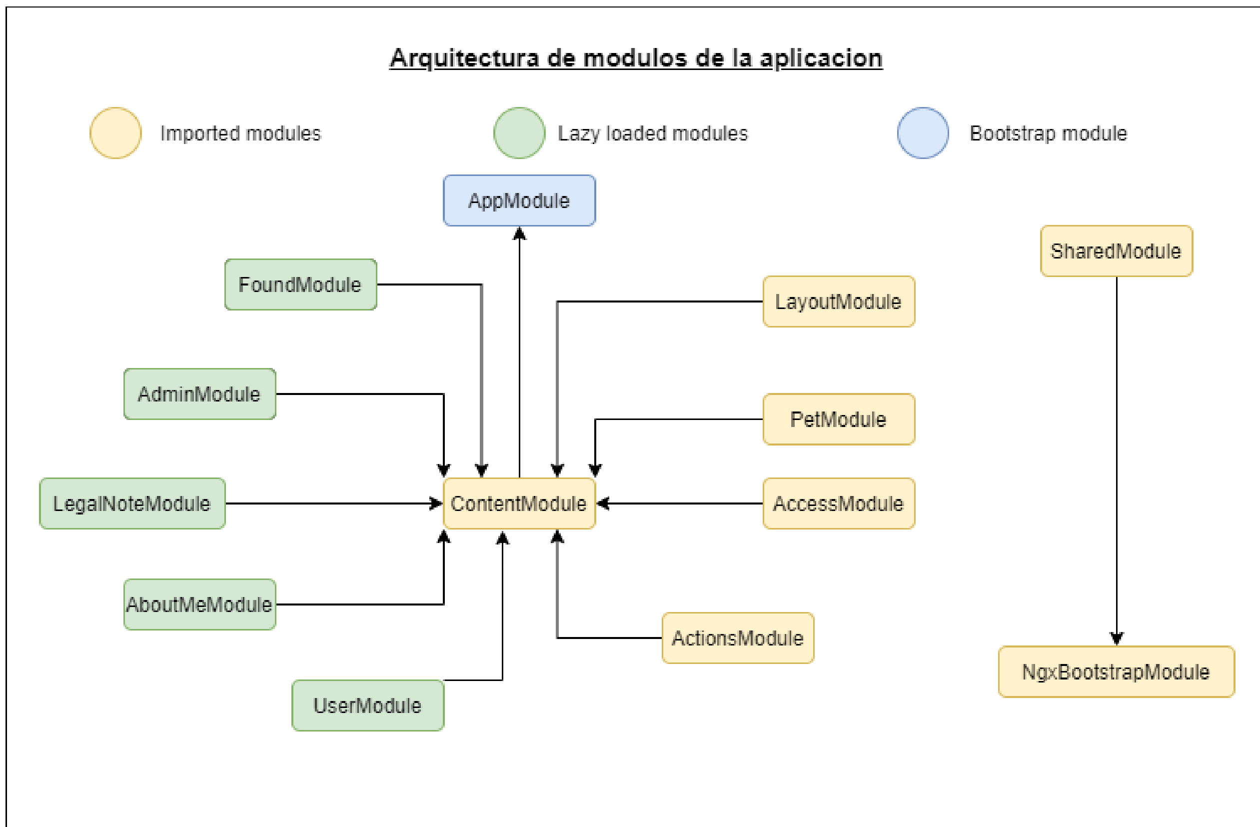



*Imagen: Arquitectura de módulos de la aplicación Angular*

En este diagrama se aprecia que el módulo raíz de la aplicación delega en *ContentModule* la proyección del contenido normal de la aplicación, así como de los elementos de estructura, como los menús o el footer (contenidos en *LayoutModule*). En caso de necesitar mostrar elementos externos al funcionamiento de la aplicación, como puede ser una ventana de sitio en mantenimiento, este contenido se proyectaría directamente en el *router-outlet* del AppModule.

El módulo *ContentModule* es la base para mostrar el contenido de la aplicación: en él, hay una serie de módulos que se importan desde el principio y otros que se cargan de forma tardía cuando la aplicación lo solicita al servidor. Esta técnica, llama *lazy loading modules* en inglés, permite reducir significativamente el peso de la aplicación cargada inicialmente en el navegador y optimizar así la carga de la aplicación en el navegador.

En el diseño de la aplicación, se parte de la premisa que, en caso de no a ver una razón especial para importar un módulo en otro, los módulos serán cargados de forma perezosa. Siguiendo esta norma, los módulos *LayoutModule* y *PetModule* se cargan de inicio (mediante importación) porque son usados en la pantalla inicial de la aplicación (*HomeComponente*). Los otros dos módulos,

*AccessModule* y *ActionsModule* contienen modales que se pueden invocar en diferentes puntos de la aplicación sin navegación, son lo que es más sencillo cargarlos al arrancar la aplicación.

En relación con los módulos de carga perezosa, es necesario distinguir dos tipos de módulos; los de página, que contienen una página y sus subcomponentes y que se cargaran cuando el usuario navegue a una ruta especifica (es el caso de *FoundModule*, *LegalNoteModule* y *AboutMeModule*). Por otro lado, están los módulos que agrupan funcionalidades reservadas, en el caso de la aplicación, a un tipo de usuario especifico. De este modo, en la aplicación aparecen dos módulos de carga perezosa; uno englobando las funcionalidades como “Mis datos” o “Mis mascotas” que tiene todo usuario identificado en la aplicación y otro (*AdminModule*) que, además, añade las opciones de Administración para controlar las mascotas y los usuarios publicados. Se han usado *guards* (implementando su método *canLoad*) para asegurar que los módulos solo se cargan cuando las condiciones para ello se cumplen: cuando el usuario que intenta acceder esta autenticado y cuando es administrador.

Por último, el módulo *SharedModule* contiene los elementos comunes usados en toda la aplicación, como, por ejemplo, el componente *spinner* que aparece en la aplicación mientras se realiza un proceso asíncrono se inserta en todos los módulos de la aplicación. Como parte de este módulo, se encuentra *NgxBootstrapModule*, un módulo cuya función es agrupar el uso de módulos de la librería NGX Bootstrap que se usan en PetFinder.

## 7.4 Despliegue

Como se ha comentado en apartados, el despliegue es una parte importante del desarrollo de la aplicación, puesto que se pretende crear una aplicación que sea fácilmente desplegable y escalable, en caso de ser necesario.

Para lograr los objetivos marcados, se ha escogido *Docker Compose* como herramienta para desplegar el stack de componentes (contenedores en el lenguaje de Docker) que componen la aplicación.

Aunque es cierto que hay otras tecnologías como *Docker Swarm* o *Kubernetes* que abordan la orquestación de contenedores de forma mucho más detallada que *Docker Compose y* ofrecen muchas más funcionalidades, como el auto escalado de aplicaciones o el uso de *Ingress* en Kubernetes para gestionar de forma centralizada y segura los accesos desde internet a una aplicación, Docker Compose permite desplegar un stack tecnológico de forma relativamente simple

(un solo fichero *docker-compose.yml* es suficiente) en muy poco tiempo en un servidor real y eso es lo que se necesita para desplegar la aplicación en producción.

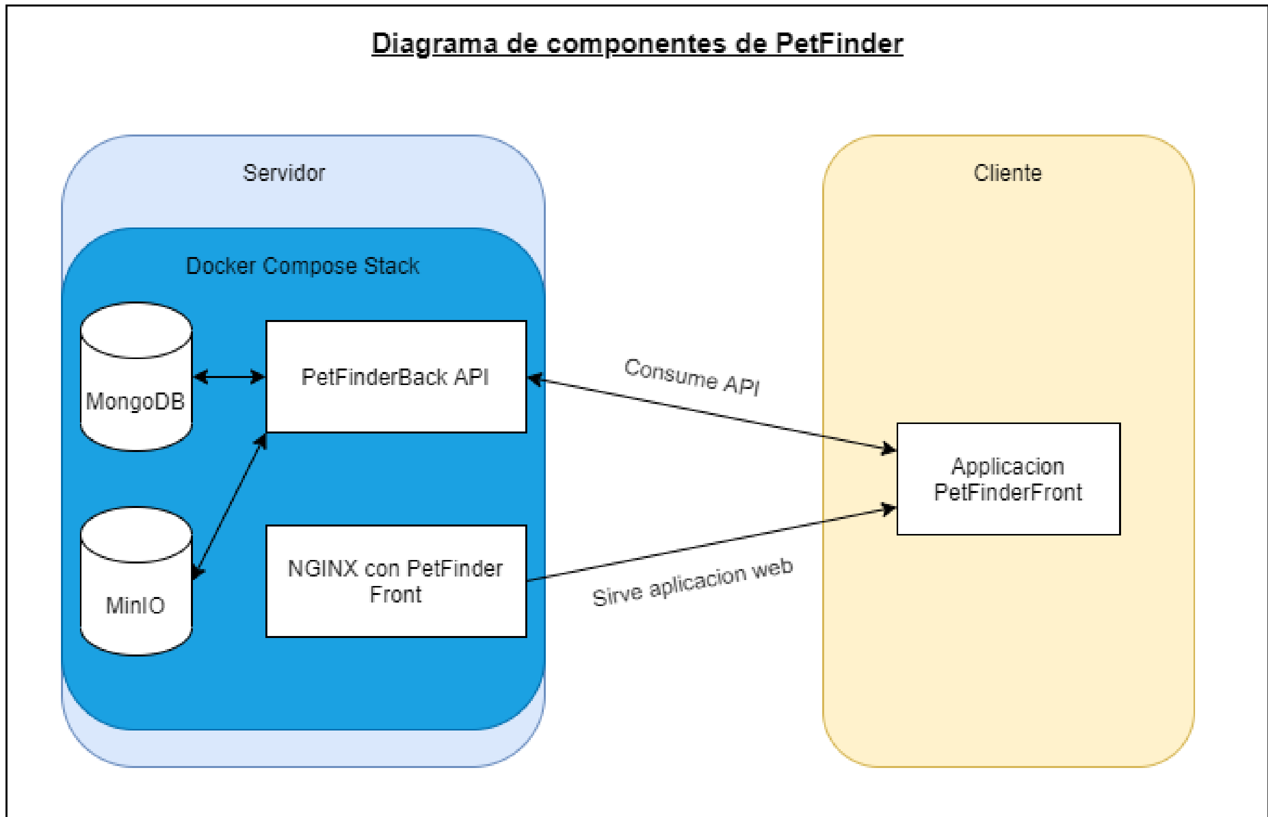



*Imagen: Diagrama de componentes del sistema PetFinder*

En este esquema se muestran los 4 contenedores desplegados con el fichero *docker-compose.yml* y como interactúan los contenedores como la aplicación JavaScript que se ejecuta en el cliente. A continuación, se explican los componentes del stack:

- **MongoDB**. El contendor de MongoDB se usa como base de datos. No se mapea ningún puerto con lo que no es posible conectarse a la base de datos desde el exterior de la red creada por Docker. En interno, la aplicación backend se comunica con la base de datos mediante el puerto 27017.
- **MinIO**. El contenedor de MinIO almacena las imágenes de las mascotas que los usuarios suben a la aplicación. De igual forma que con MongoDB, no se mapean puerto al exterior, con lo que este contenedor no es accesible desde el exterior. En interno, se expone el puerto 9000 para la comunicación con la aplicación backend.

- **PetFinderBack API**. Este contenedor ejecuta la aplicación de Spring Boot que sirve de backend. En este caso, se ha mapeado el puerto 8080 para que la aplicación frontend pueda conectarse a la API (hay que tener en cuenta que la aplicación frontend consume la API desde el cliente a través de internet). Se asigna el subdominio siguiente a la API: https://petfinder-api.ricardmolinaferret.com/.
- **NGINX con PetFinderFront.** Este contenedor es un servidor web, NGINX, configurado para servir la aplicación frontend compilada. Como es un servidor web, se mapea el puerto 80 y se le asigna el dominio de la aplicación: https://petfinder.ricardmolinaferret.com/. Esta es la URL que un usuario debe introducir en el navegador para acceder a PetFinder.

```yaml
version: "3"

services:
  mongodb:
    image: mongo
    restart: always
    expose:
      - 27017
  minio:
    image: minio/minio:latest
    restart: always
    expose:
      - 9000
    environment:
      MINIO_ACCESS_KEY: AKIAIOSFODNN7EXAMPLE
      MINIO_SECRET_KEY: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
    command: server minio/export
  petfinder-back:
    image: junker52/petfinder-back:1.0.0-SNAPSHOT
    restart: always
    ports:
      - 9091:8080
    environment:
      SPRING.PROFILES.ACTIVE: docker
      MINIO_ACCESS_KEY: AKIAIOSFODNN7EXAMPLE
      MINIO_SECRET_KEY: wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
    depends_on:
      - mongodb
      - minio
  petfinder-front:
    image: junker52/petfinder-front:1.0.0
    restart: always
    ports:
      - 8081:80
```

*Imagen: Fichero docker-compose.yml de la aplicación.*

Aunque el fichero definitivo está disponible junto con el código fuente de la aplicación, esta imagen contiene una versión estable del fichero *docker-compose.yml* que permite lanzar la aplicación en producción. Se puede apreciar que con poco más de 30 líneas de código se puede desplegar una aplicación completa.

# 8. Plataforma de desarrollo

Para el desarrollo de la aplicación PetFinder se han utilizado los siguientes recursos tecnológicos. En relación con el hardware utilizado, se han utilizado los recursos siguientes:

- **Ordenador personal**. Se trata de un portátil Acer Aspite V Nitro equipado con un procesador Intel i7 6400-HQ (4 nucleaos y 8 hilos de procesamiento) con una memoria RAM de 16 GB y un disco duro SSD de 256 GB. El sistema operativo es Windows 10 Pro en su versión de 64 bits.
- **Servidor remoto**. Se trata de un servidor personal equipado con procesador Intel Celeron Apollo Lake J3455 de 4 nucleos y 4 hilos de procesamiento. El servidor tiene una memoria RAM de 4 GB y un disco duro de 1 TB. El sistema operativo es Ubuntu Server 18.04 con Docker, en su versión 19.03.11, instalado y configurado (el mismo que se ha usado para el desarrollo).

En lo que respecta al apartado de software, es necesario señalar que para las aplicaciones o dependencias de terceros (bases de datos, servidores de almacenamiento, etc), se ha optado siempre por el uso de contenedores Docker con la dependencia instalada y operativa, en lugar de instalarlas directamente en el puesto de desarrollo.

- **Base de datos**. Se ha usado la imagen de Docker oficial de MongoDB en su versión 4.0.20 (la más reciente en el momento de realizar el proyecto).

- **Servidor de almacenaje de ficheros**. Se ha usado la imagen de Docker oficial de MinIO en su versión más reciente.

- **Dependencias de base para el desarrollo:** Se ha trabajado con el SDK de Java en su versión 1.8.0_211, con el gestor de dependencias Maven (versión 3.6.3), el entorno de ejecución de Javascript Node, en su versión 14.8.0, la librería NPM Typescript (v4.0.3) y, por último, la librería Angular CLI, en su versión 9.1.12.

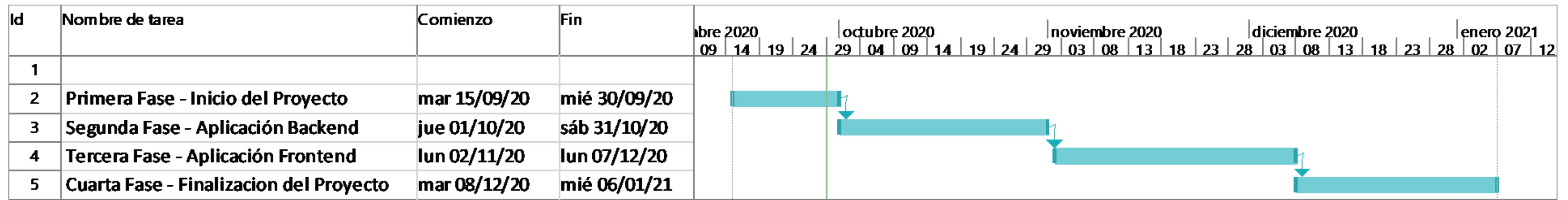
- **IDEs** (Entornos Integrados de Desarrollo, por sus siglas en inglés): Para el desarrollo de la aplicación backend, se ha usado IntelliJ IDEA 2020 2.2. Para la aplicación frontend, a su vez, se ha usado Visual Studio Code 1.50.1.

# 9. Planificación

Enlazando con la metodología basada en iteraciones que se ha explicado anteriormente, se han identificado 4 o iteraciones en que se van a realizar las diferentes partes del proyecto y se van a corregir los aspectos necesarios de las iteraciones anteriores.

1) **Primera fase.** Concepción y planificación de la aplicación *PetFinder*. Se deciden las principales necesidades que serán resueltas por la aplicación. En paralelo se empieza a preparar la memoria del proyecto y se definen aspectos como los objetivos del proyecto o la metodología a seguir durante la realización.
2) **Segunda fase.** Se define la estructura de datos de la aplicación y la mayor parte del trabajo pasa por la confección de los apartados de la memoria relacionados con la concepción y la arquitectura y por la realización de una versión beta de la aplicación backend.
3) **Tercera fase.** Con los servicios REST disponibles en versión beta, el foco pasa a la aplicación frontend. En esta fase se diseña y se desarrolla la aplicación cliente que consumirá los servicios y se encargará de presentar la información a los usuarios. Durante esta fase también se corregirán los errores que se pueden encontrar en la aplicación backend. Se completará la memoria con los apartados correspondientes al diseño y las funcionalidades incorporadas en la aplicación.
4) **Cuarta fase.** En esta fase se cerrará el desarrollo y la corrección de errores en ambas aplicaciones y se trabajará en el despliegue de la aplicación *PetFinder* mediante el uso de Docker. En paralelo, se desplegará la versión final de la aplicación para que funcione de forma estable. Se completarán los documentos del proyecto y se preparara todo el material para ser entregado al final de esta fase. Esta es la última fase del proyecto.

A continuación, se adjunta el diagrama de Gantt con las fases explicadas anteriormente, sus fechas y su distribución sobre el calendario del proyecto.



| Id | Nombre de tarea | Comienzo | Fin |
|---|---|---|---|
| 1 | | | |
| 2 | Primera Fase - Inicio del Proyecto | mar 15/09/20 | mié 30/09/20 |
| 3 | Segunda Fase - Aplicación Backend | jue 01/10/20 | sáb 31/10/20 |
| 4 | Tercera Fase - Aplicación Frontend | lun 02/11/20 | lun 07/12/20 |
| 5 | Cuarta Fase - Finalizacion del Proyecto | mar 08/12/20 | mié 06/01/21 |

# 11. APIs utilizadas

La aplicación, para su normal funcionamiento, consume principalmente 3 APIs diferentes de dos proveedores de servicios en la nube:

- ***Facebook***. De la gran variedad de servicios en la nube que *Facebook* propone, se ha utilizado la funcionalidad para el inicio de sesión (OAuth 2).

- ***Google:*** De la misma manera que *Facebook*, *Google* propone también una amplia gama de servicios en la nube. Sin embargo, para este proyecto se van a utilizar los servicios de autenticación OAuth (igual que hacemos con *Facebook*).

## 11.1 Servicio *Facebook*

Para poder consumir los servicios REST de *Facebook* es necesario, en primer lugar, crear una aplicación en el portal de desarrolladores de *Facebook* (https://developers.facebook.com/).

El proceso de creación es bastante sencillo y, tras rellenar campos como el nombre de la aplicación o el email de contacto, la aplicación se crea rápidamente y el siguiente paso es elegir los servicios que se quieren consumir entre todos los propuestos por *Facebook*.

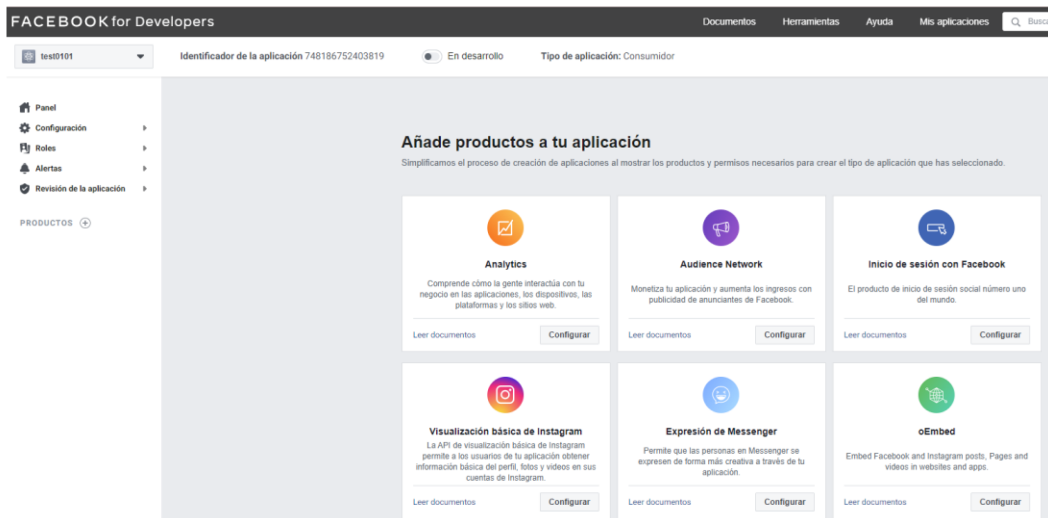



*Imagen: Servicios Facebook a disposición*

Como hemos explicado, una vez creada la aplicación, es momento de seleccionar la funcionalidad que se quiere usar. En el caso de esta aplicación, "Inicio de sesión con Facebook". Seleccionamos el tipo de aplicación "web" y continuamos.

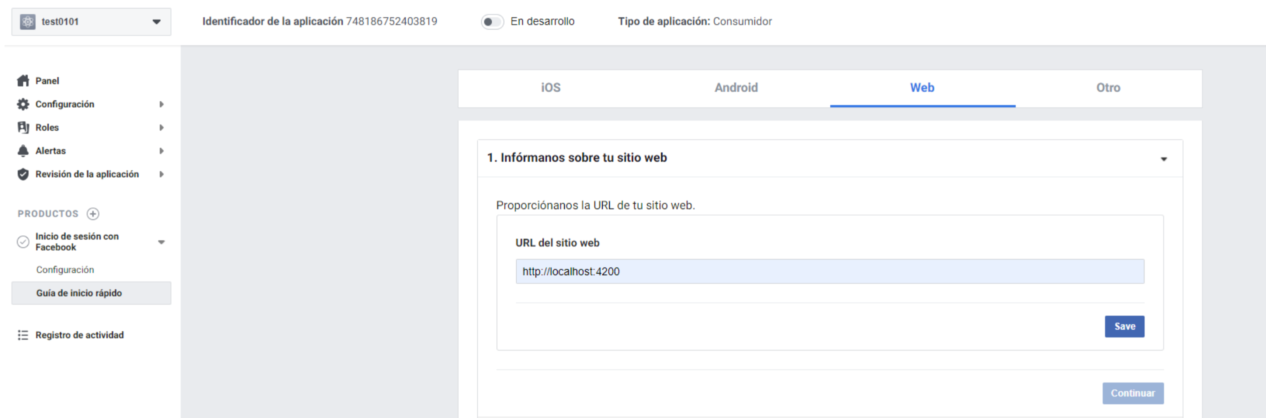



*Imagen: Redirección de la autenticación*

En este punto, solo queda indicar la URL a la que Facebook debe redirigir la petición tras la autenticación. Esta URL debe corresponder con la URL de la aplicación frontend.

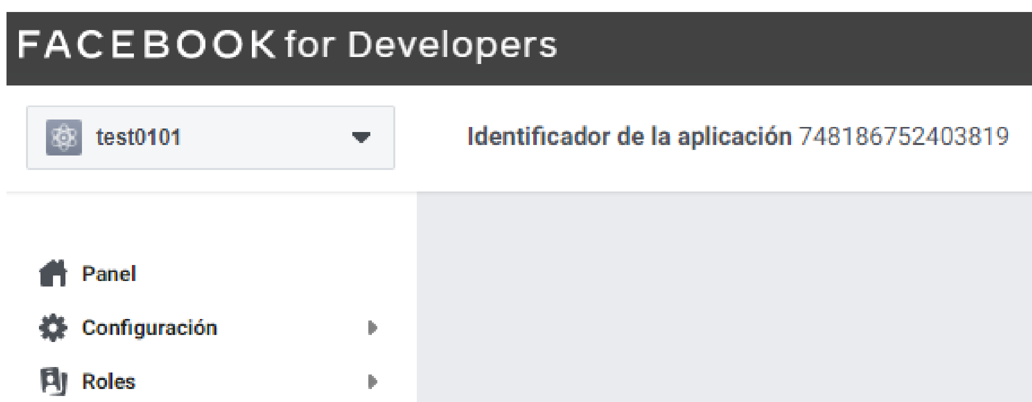



*Imagen: Identificador de la aplicación creada*

Con este identificador de aplicación ya es posible usar la API de Inicio de Sesión de Facebook.

Para usar los servicios que *Facebook* propone en la aplicación backend se puede usar la librería de *Spring Social* que implementa la integración con la API de Facebook.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-social-facebook</artifactId>
    <version>1.5.22.RELEASE</version>
</dependency>
```

*Imagen: Dependencia Maven de Spring Social Facebook*

El uso de esta librería de *Spring* permite abstraer el desarrollo de la aplicación de las especificaciones propias de la API de Facebook y permiten, mediante un objeto *FacebookTemplate* obtener la información necesaria de Facebook.

```java
public LoginResponseDto connectWithFacebook(TokenDto tokenDto) {
    UserDto userDto = this.extractFacebookUserFromToken(tokenDto);

    if (!this.userRepository.existsByEmail(userDto.getEmail())) {
        userDto.setIsSocial(Boolean.TRUE);
        userDto = this.userService.createUser(userDto);
    }

    return this.autoLoginUser(userDto);
}

private UserDto extractFacebookUserFromToken(TokenDto tokenDto) {
    UserDto userDto;
    try {
        Facebook facebook = new FacebookTemplate(tokenDto.getToken());
        String[] userFields = {"email", "picture"};
        User user = facebook.fetchObject( objectId: "me", User.class, userFields);
        userDto = UserDto.builder()
                .firstName(user.getFirstName())
                .lastName(user.getLastName())
                .email(user.getEmail())
                .password(this.encoder.encode(this.defaultSocialPassword))
                .build();
    } catch (Exception e) {
        throw new PetFinderException("Error: Impossible to connect with Facebook", e.getLocalizedMessage(), HttpStatus.BAD_REQUEST, null);
    }
    return userDto;
}
```

*Imagen: Uso de la librería Spring Social Facebook*

Se puede ver que a partir del accessToken que se envía desde la aplicación frontend y mediante el uso de *FacebookTemplate* se recupera y se guarda en base de datos la información relativa al usuario que esta intentando identificarse en la aplicación.

En caso que el token proporcionado no sea válido, el método *fetchObject()* lanza una excepción que es procesada y se traduce en un error en la petición de login.

## 11.2 Servicio *Google*

El caso de *Google* es bastante parecido al de Facebook. Para poder consumir los servicios REST de Google se debe crear una aplicación en el portal de *Google Cloud Platform* (antiguamente conocido como la Consola de desarrolladores de *Google*) (https://console.developers.google.com/).

La creación de una aplicación en Google es, si cabe, más rápido de realizar que en *Facebook*. Basta con indicar el nombre de la aplicación y crearla.

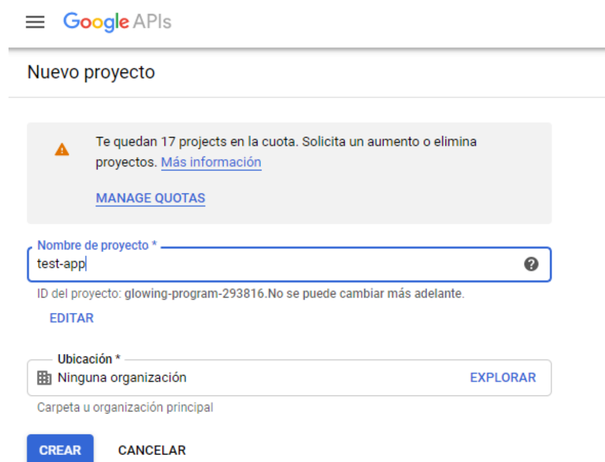



*Imagen: Creación de una aplicación Google*

Seguidamente, con la aplicación ya creada, hay que dirigirse a la opción “Credenciales” y crear una nueva credencial de tipo OAuth2. Para ello, primero, será necesario configurar la pantalla de consentimiento en que se indica la información que se va a compartir con la aplicación cuando un usuario se autentica con Google.

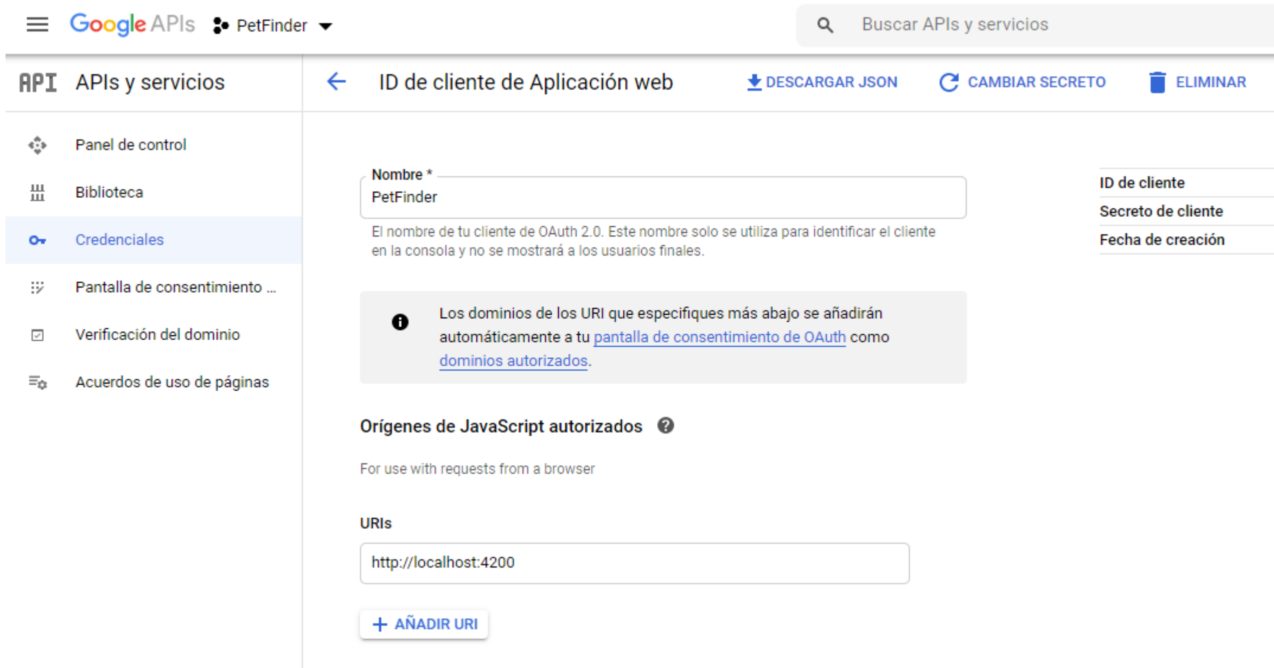



*Imagen: Resumen de creación de las credenciales*

Con el Id de cliente y el secreto de cliente, ya es posible consumir la API desde un tercero. Es importante destacar que el secreto de cliente es eso, un secreto, y en ninguna circunstancia se debe enviar al cliente o divulgar.

En este caso, se ha optado por codificar en un nivel más bajo la conexión con los servicios de Google y prescindir, de este modo, del paquete de *Spring Social Google*. De este modo, también se codifica una conexión que incluye, entre otros, un *deserializer* (Jackson) o un canal de transporte.

```
<dependency>
    <groupId>com.google.api-client</groupId>
    <artifactId>google-api-client</artifactId>
    <version>1.30.10</version>
</dependency>
```

*Imagen: Librería Maven para la conexión con Google APIs*

Como apuntaba en las líneas anteriores, se ha decidido partir de un cliente HTTP preconfigurado para la API de Google para codificar la llamada a los servicios de OAuth2 de Google.

```
public LoginResponseDto connectWithGoogle(TokenDto tokenDto) {

    UserDto userDto = this.extractGoogleUserFromToken(tokenDto);

    if (!this.userRepository.existsByEmail(userDto.getEmail())) {
        userDto.setIsSocial(Boolean.TRUE);
        userDto = this.userService.createUser(userDto);
    }

    return this.autoLoginUser(userDto);

}

private UserDto extractGoogleUserFromToken(TokenDto tokenDto) {
    UserDto userDto;
    try {
        NetHttpTransport httpTransport = new NetHttpTransport();
        JacksonFactory jacksonFactory = JacksonFactory.getDefaultInstance();
        GoogleIdTokenVerifier.Builder verifier = new GoogleIdTokenVerifier.Builder(httpTransport, jacksonFactory)
                .setAudience(Arrays.asList(this.googleId));
        GoogleIdToken idToken = GoogleIdToken.parse(jacksonFactory, tokenDto.getToken());
        GoogleIdToken.Payload payload = idToken.getPayload();
        userDto = UserDto.builder().email(payload.getEmail())
                .password(this.encoder.encode(this.defaultSocialPassword)).build();
    } catch (Exception e) {
        throw new PetFinderException("Error: Impossible to connect with Google", e.getLocalizedMessage(), HttpStatus.BAD_REQUEST, null);
    }
    return userDto;
}
```

*Imagen: Conexión con Google para validar el token*

De la misma forma que se realizó con Facebook, el proceso empieza con un idToken enviado desde el frontend (después de la redirección de la autenticación de Google). Se crea un objeto *GoogleTokenVerifier* que es el que permite verificar que el token es válido y recuperar, en el cuerpo de la respuesta de Google, la información relativa al usuario identificado.

Comparado con el proceso de validación de token y recuperación del usuario que se implementó con *Facebook*, vemos que el proceso implica la creación de más objetos y una configuración más manual de la llamada al objeto *Verifier*, mientras que, si se hubiera usado el paquete *Spring Social Google* desarrollado por la comunidad, la autenticación seria más simple de implementar.

# 12. Diagramas UML

En este apartado se van a presentar los diagramas más importantes que han permitido diseñar y desarrollar la aplicación y el modelo de datos.

## 12.1 Casos de Uso

Para entender las diferentes funcionalidades que la aplicación propone a sus usuarios y la manera como estos interactúan con el sistema, es necesario presentar el Diagrama de Casos de Uso del sistema, así como una breve presentación de los actores que toman parte y los diferentes casos de uso identificados.

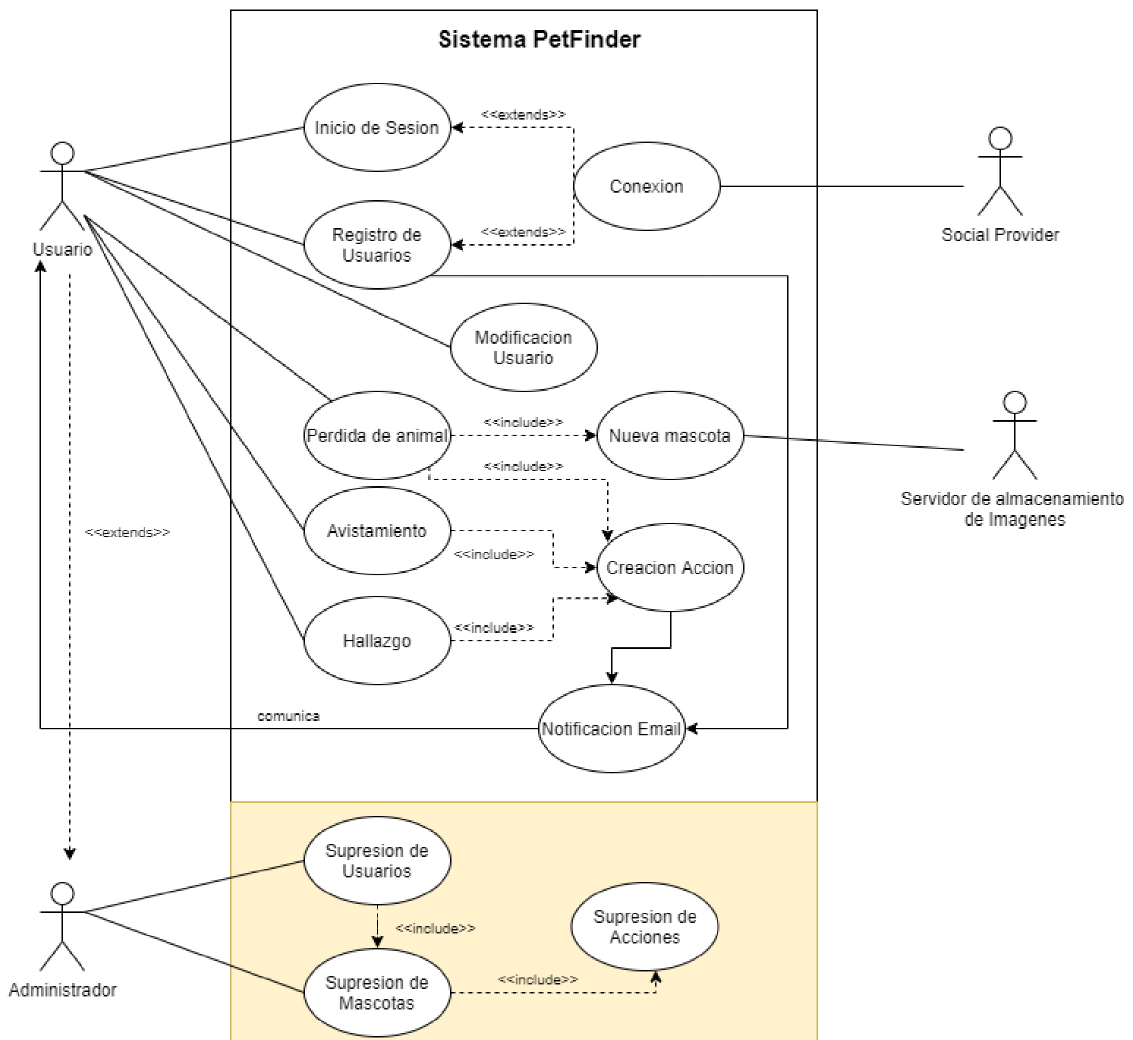



*Imagen: Diagrama de Casos de Uso del Sistema*

En el diagrama se pueden ver los casos de uso principales de la aplicación (aunque se pueden identificar otros de menor importancia, como la modificación de acciones o mascotas), sus relaciones.

Como se puede en el diagrama, se han identificado 4 actores diferentes que interactuaran con la aplicación. Estos se pueden dividir en actores primarios o secundarios:

- **Actores primarios**. Son todos aquellos actores que interactúan directamente con la aplicación con el propósito de usar alguna de sus funcionalidades (casos de uso, en lo que concierne al diagrama). En el caso de PetFinder, se identifican como actores primarios a los usuarios y a los administradores, un actor que extiende de usuario y que tiene acceso a casos de uso específicos.
- **Actores secundarios**. Son actores secundarios, todos aquellos actores que prestan soporte a la aplicación para que esta funcione correctamente y los usuarios primarios puedan explotar sus funcionalidades normalmente. En este caso, se han identificado como actores secundarios a los proveedores de identificación (Social Providers), puesto que, en determinados casos, la aplicación depende de esos servicios externos para funcionar, y el almacenamiento de imágenes en un servidor externo, puesto que no es parte integral del sistema, pero es necesario en el caso de que los usuarios primarios quieran cargar imágenes de sus mascotas.

En este punto es necesario aclarar que la base de datos, no se ha considerado un actor secundario porque se considera una parte intrínseca del sistema dado que el soporte para almacenar la información es vital para el funcionamiento de la aplicación.

Para terminar el apartado del Diagrama de Casos de Uso de la aplicación, se van a desglosar, de forma sintética, los diferentes casos de uso expuestos en el diagrama.

| Nombre | Actores | Relaciones | Descripción |
|---|---|---|---|
| **Conexión** | Social Providers | | Caso de uso genérico que evoca el reconocimiento de un usuario por parte de la aplicación. |

| | | | |
|---|---|---|---|
| **Inicio de Sesión** | Usuario | Extiende de conexión | Un usuario ya registrado en la aplicación inicia sesión, manualmente o con redes sociales. |
| **Registro de Usuarios** | Usuario | Extiende de conexión | Un usuario introduce sus datos y se inscribe para poder ser reconocido en el sistema. Manualmente o con redes sociales. |
| **Modificación de Usuario** | Usuario | | Un usuario registrado puede ver y modificar sus datos personales en el sistema |
| **Nueva mascota** | Servidor de imágenes | | Caso de uso genérico consistente en la creación de una nueva mascota en el sistema. Se crea cuando se declara una perdida. |
| **Creación de acción** | | Genera una notificación por email | Caso de uso genérico consistente en la creación de una acción de perdida, avistamiento o hallazgo |
| **Perdida de animal** | Usuario | Incluye creación de acción. Puede incluir Nueva mascota | Un usuario registrado indica a la aplicación que ha perdido su mascota. Si la mascota no existía en el sistema se crea. Una acción de perdida se crea en todo caso. |
| **Avistamiento** | Usuario | Incluye creación de acción | Un usuario registrado comunica que ha avistado una mascota perdida (existente en el sistema). Una acción de avistamiento se crea sobre esa mascota. |
| **Hallazgo** | Usuario | Incluye creación de acción | Un usuario registrado comunica que ha encontrado un animal perdido y lo retiene (sino se considera avistamiento). Una acción de hallazgo se crea sobre esa mascota |
| **Notificación Email** | Comunica a Usuario | Se activa con Registro de Usuario y con Creación de acción | El usuario recibe en su cuenta de correo, notificaciones cuando se registra y cuando se agrega alguna acción a su mascota. (Perdida, Avistamiento o Hallazgo) |
| **Supresión de usuarios** | Administrador | Incluye Supresión de Mascotas | El administrador puede suprimir usuarios considerados maliciosos, así como sus mascotas. |
| **Supresión de mascotas** | Administrador | Incluye Supresión de Acciones | El administrador puede suprimir una mascota y todas las acciones asociadas a |

| | |
|---|---|
| | esta si considera que se trata de contenido falso o malicioso. |
| **Supresión de acciones** | Caso de uso implícito en que se suprimen las acciones ligadas a una mascota cuando esta es suprimida. No se puede invocar directamente. |

Con el desglose de los casos de uso más importantes previstos en la aplicación se cierra el apartado dedicado a los Casos de Uso.

## 12.2 Clases

En toda aplicación que adquiera un cierto nivel de complejidad en cuanto a su estructura de datos, resulta útil crear un diagrama de clases para entender como están relacionados los datos que se guardan en la base de datos y que constituyen la base del funcionamiento de la aplicación.

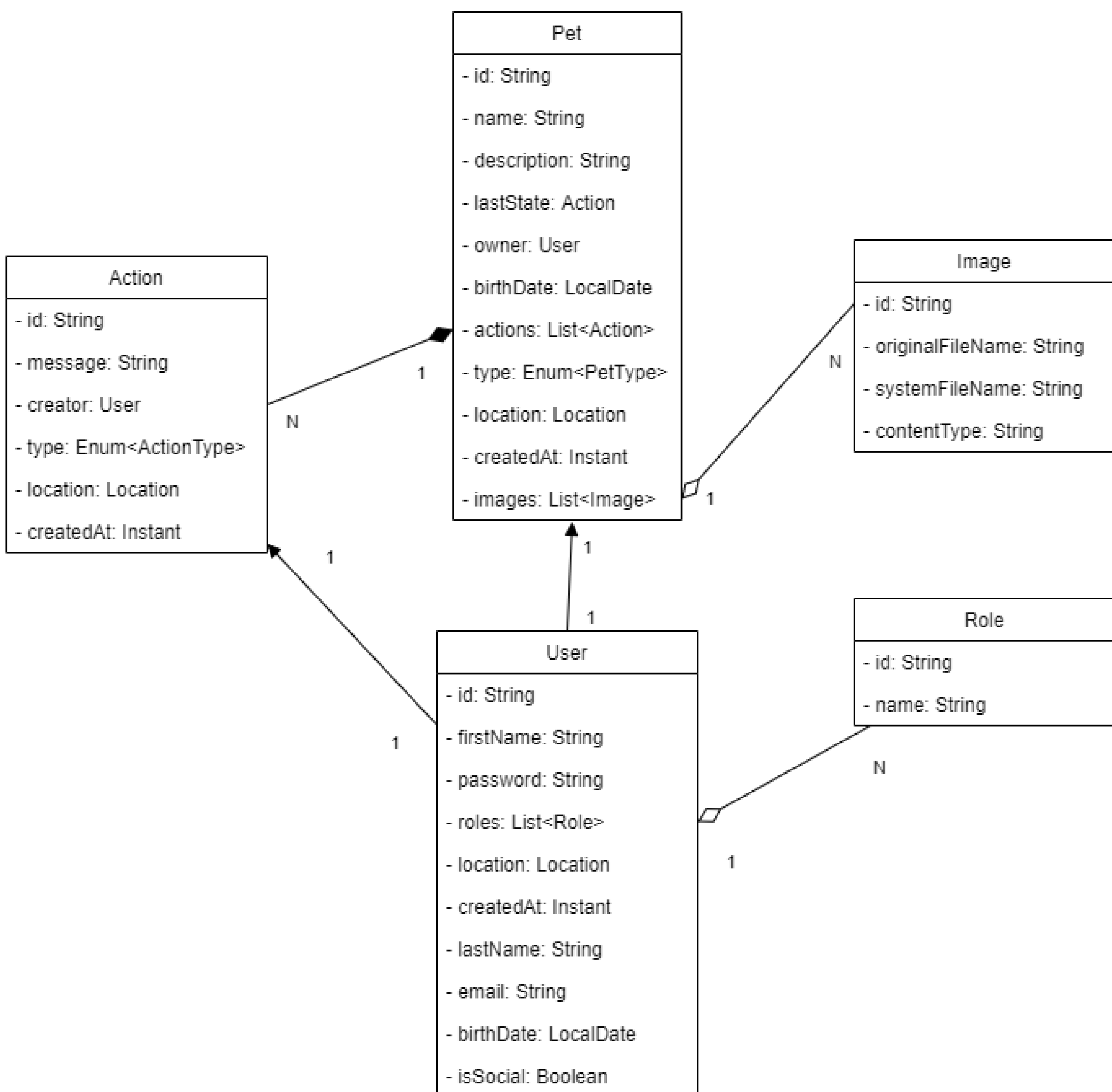



*Imagen: Diagrama de Clases/Entidades*

En el diagrama de clases expuesto se muestran las clases que mapean las colecciones existentes en la base de datos y que, a su vez, sirven de base para el desarrollo de la lógica de la aplicación.

Hay varios puntos del diagrama que me gustaría explicar. En primer lugar, el símbolo menos (-) que aparece al principio de cada atributo corresponde con el nivel de acceso a esta variable. Es una buena practica de desarrollo gestionar el acceso a los atributos de una clase u objeto a través de los métodos *getters* y *setters* en vez de hacerlo directamente.

En segundo lugar, es necesario que se tratan de modelos "anémicos", es decir, clases cuya única función es mapear la información contenida en la base datos y crear objetos en la aplicación a partir de los que poder trabajar. Es por esa razón que no se encuentran métodos en las clases expuestas. Por encima de estas clases, en la aplicación, existen los servicios y las clases de trabajo (o DTO, *Data Transfer Objects* por sus siglas en ingles) que modifican la información recuperada por los modelos y aplican la lógica aplicativa necesaria. En este sentido, aunque existen métodos *getters and setters* en las clases del diagrama, estos métodos no se reflejan por considerarse como código altamente redundante y para no sobre cargar el diagrama.

Para terminar este apartado, creo necesario explicar las relaciones más importantes en la aplicación. Como se puede ver en el diagrama, existe una relación de composición entre Action y Pet, lo cual parece lógico puesto que las acciones no tienen sentido si no se asocian a una mascota sobre la que aportan información. En cambio, la relación entre Pet y Image se considera como una agregación. La razón es más técnica que funcional, puesto que, aunque una imagen sin mascota asociada jamás será utilizada, la imagen continuará existiendo en el servidor de almacenamiento de imágenes hasta que sea suprimida y, por esa razón, la imagen debe preservarse en el sistema, aunque la mascota sea suprimida. Por último, las relaciones que parten de usuario son se tratan de asociaciones direccionales simples, dado que User es una entidad con un ciclo de vida propio e independiente del ciclo de vida de Pet o Action.

## 12.3 Secuencia del Social Login

Antes de terminar este apartado de diagramas, creo necesario dedicar un apartado a explicar el proceso del *Social Login,* el mecanismo que permite a los usuarios de PetFinder registrarse y conectarse mediante un usuario creado con información de un proveedor de autenticación tercero (*Google y Facebook* en el caso de la aplicación).

Una buena forma de ilustrar el funcionamiento de esta funcionalidad es mediante el uso de un diagrama de secuencia.

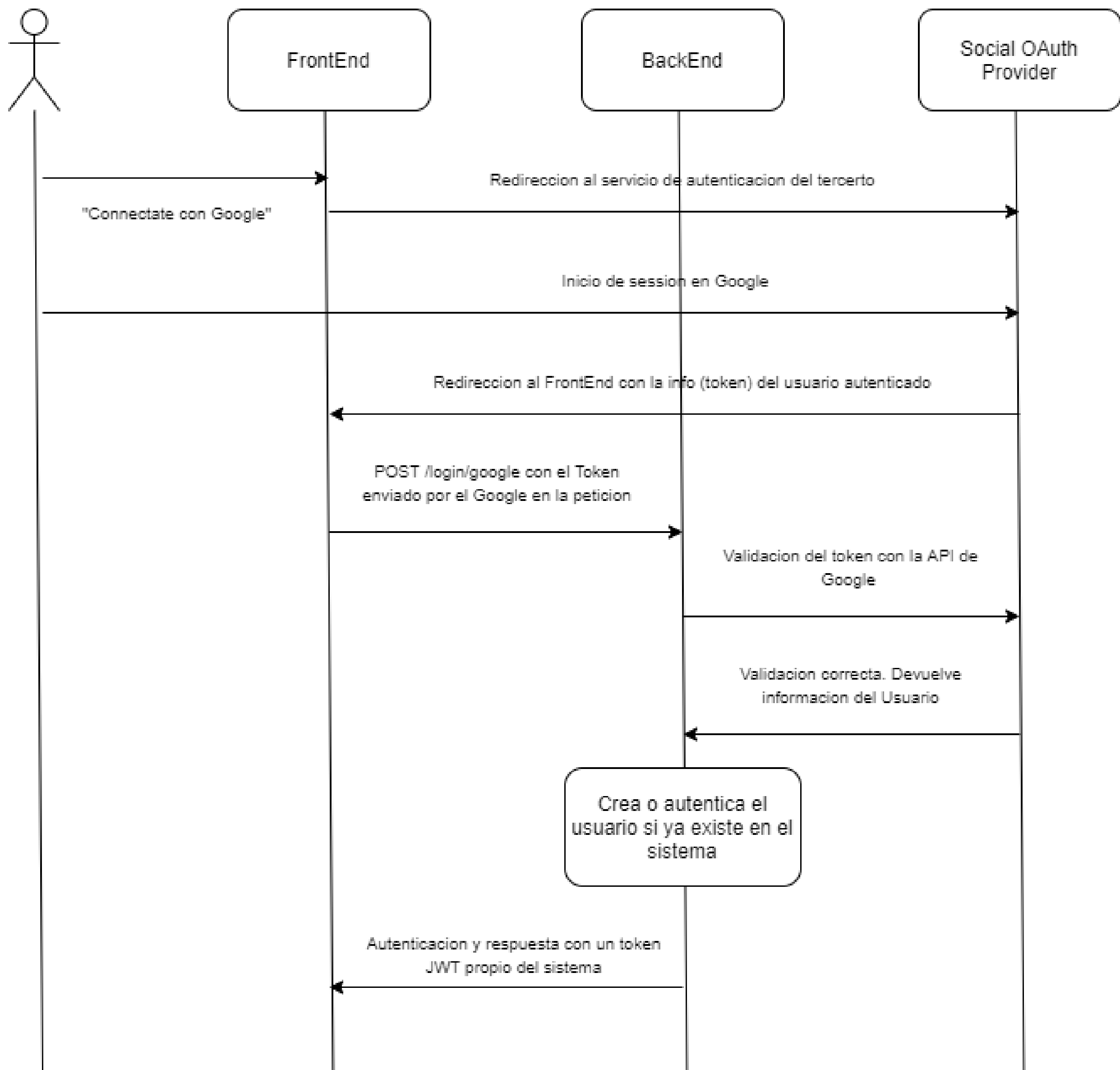



*Imagen: Diagrama de Secuencia para el Social Login*

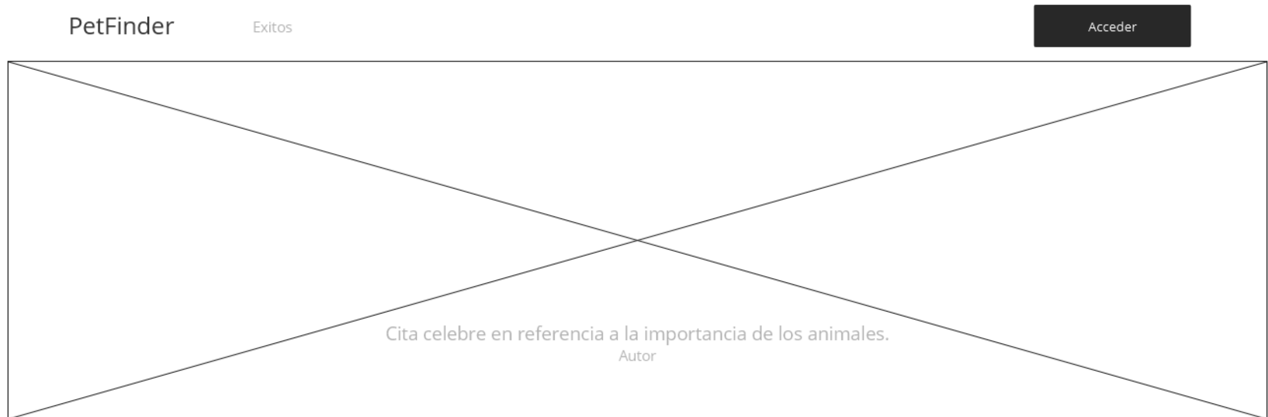
En el diagrama se puede ver la secuencia que acciones que se realizan desde los diferentes actores del proceso para que el usuario se autentique en el sistema cuando usa un proveedor de OAuth externo.

Creo importante destacar que el proveedor de autenticación es utilizado dos veces durante el proceso de autenticación: la primera vez desde la aplicación frontend para que el usuario se

redirigido a la página de login del proveedor y la segunda vez, desde la aplicación backend, para validar que la autenticación desde el frontend es correcta (mediante un token) y, si es así, recuperar la información del usuario, crear un usuario si no existe en la aplicación y autenticarlo para devolverle un token JWT de la aplicación y que pueda conectarse normalmente.

# 13. Prototipos

El prototipado inicial de la aplicación frontend ha dado como resultado un diseño basado en múltiples ventanas modales y dos páginas principales –las más importantes y visitadas– de la aplicación: la página de inicio y la página de detalle.



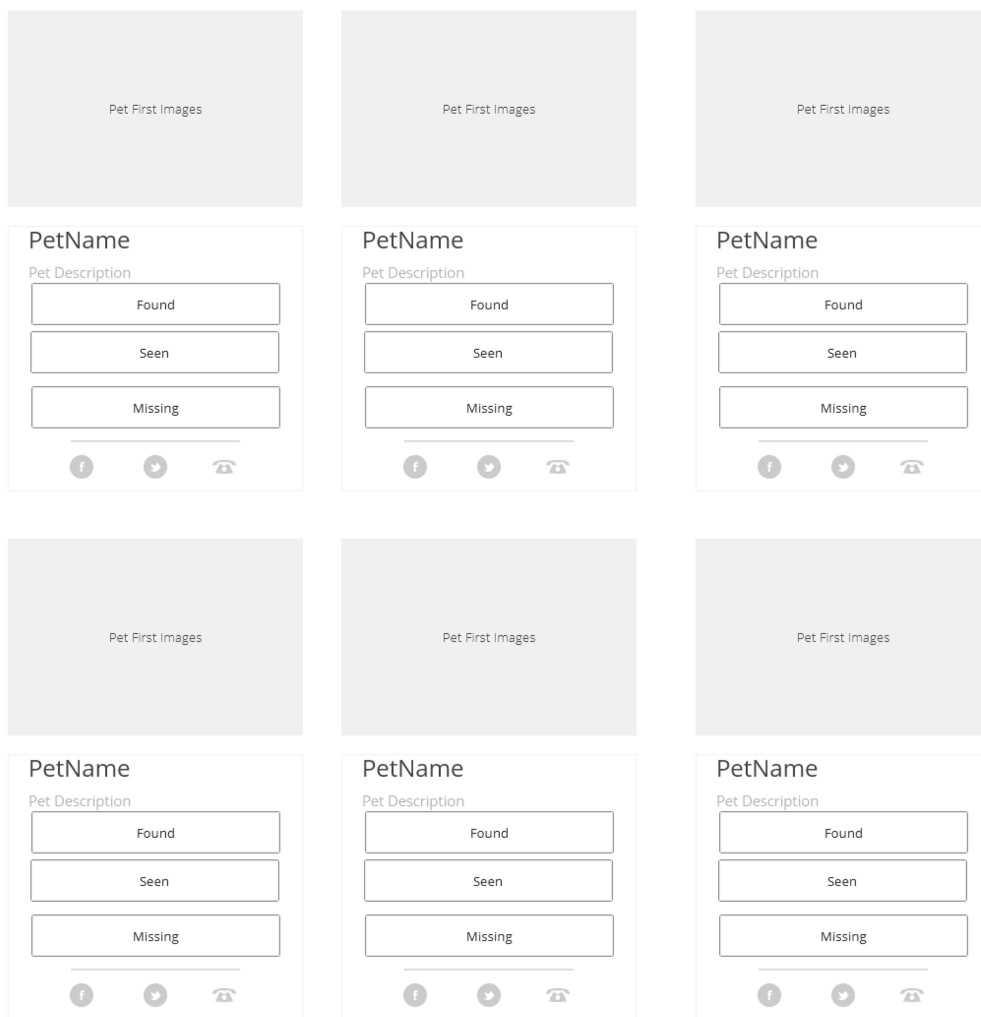



*Imagen: Prototipo de la página de inicio*

Como se ha especificado en el apartado de Usabilidad, se ve como se implementa un sistema de *Grid* con *Cards* para agrupar la información de cada mascota. En esta tarjeta se destacan la primera imagen de cada mascota y su nombro, así como la distancia entre la última acción de la mascota y la geo posición del usuario.

En relación con la navegación, se puede apreciar que, en dispositivos grandes, la navegación se realiza mediante un menú horizontal superior y un botón desplegable con las opciones de usuarios. Sin embargo, en dispositivos móviles, la navegación se realiza mediante un menú lateral deslizante que aparece gracias a un botón “Burger” situado al lado del título de la aplicación “PetFinder”.

En el caso de dispositivos pequeños o de smartphones, las tarjetas de cada mascota se muestran apiladas una debajo de la otra, con lo que se adapta sin problemas a pantallas de ancho pequeño.

La navegación hacia el detalle de cada mascota se realiza clicando en cualquier parte de las tarjetas de mascotas.

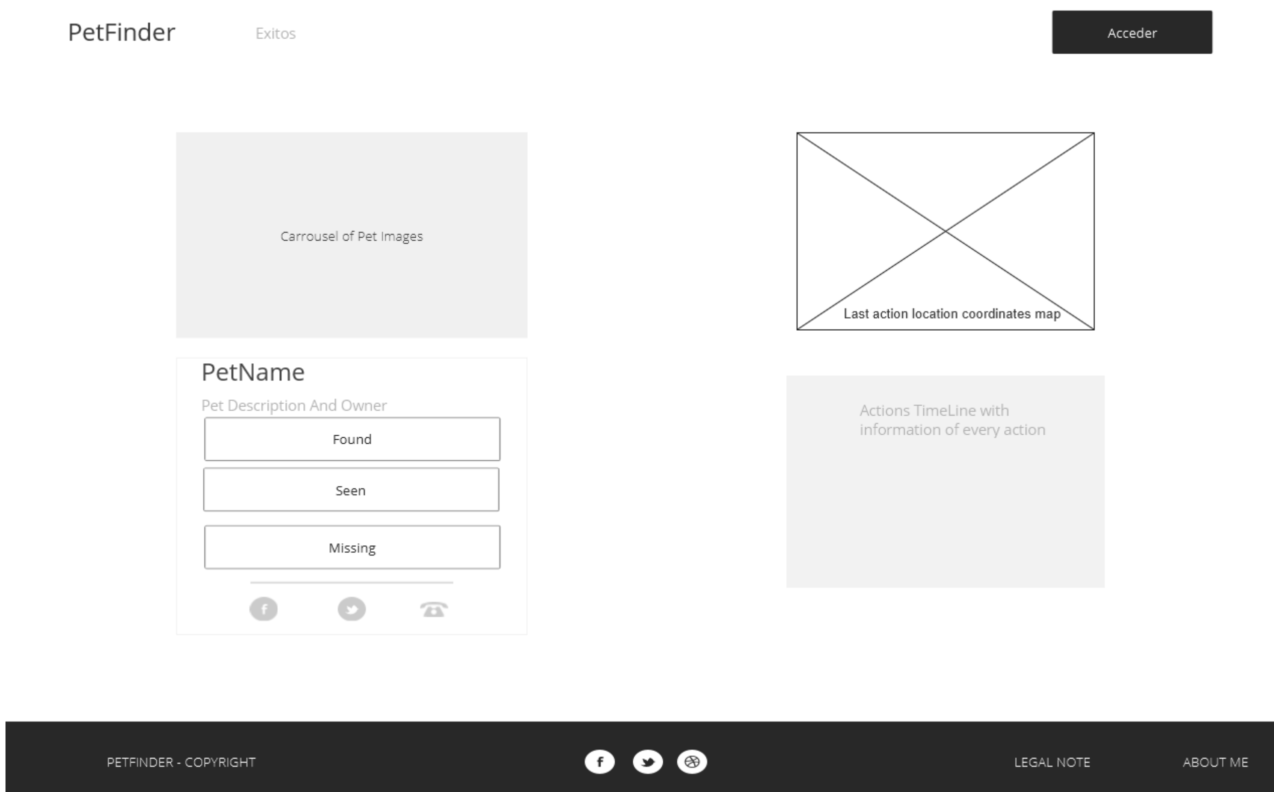



*Imagen: Prototipo de la página de detalle*

En la imagen anterior se aprecia el diseño de la página de detalle de mascota. De forma parecida al diseño de la página inicial, la página de detalle se basa en el uso de un *grid* dividido en 2 columnas (que pasan a una sola columna en el caso de dispositivos pequeños).

En la primera columna se muestra contenido muy parecido a la tarjeta de mascota que se utilizan en la pantalla de inicio, con la excepción notable de que, en lugar de una imagen, se incluye un carrusel que muestra todas las imágenes de la mascota, puesto que se pueden agregar hasta 5 imágenes. También se muestra un icono, en caso de ser el propietario de la mascota, que permite editar y modificar la información y las imágenes de la mascota.

# 14. Perfiles de usuario

Para identificar los diferentes usuarios que se encuentran en la aplicación se pueden clasificar en función de dos criterios: roles y procedencia del usuario en la aplicación.

En función de los roles, se diferencian dos usuarios diferentes:

- **Usuarios Administradores.** Se trata de un tipo de usuario con poder para suprimir otros usuarios, mascotas o acciones sobre mascotas además de realizar todas las acciones habituales. Solo se genera un usuario administrador al arrancar la aplicación.
- **Usuarios sin privilegios.** Todos los usuarios que se registran en la aplicación y que pueden realizar las acciones habituales previstas en la aplicación.

En función del origen de la información utilizada para crear el usuario se diferencian dos tipos de usuarios:

- **Usuarios propios.** Se considera un usuario propio a aquel que se ha registrado mediante el formulario de registro de usuarios en la aplicación. Los usuarios de este tipo tienen mayor flexibilidad para modificar sus datos personales en la aplicación.
- **Usuarios sociales.** Se considera un usuario social a aquel que se ha registrado mediante el uso de *Google* o *Facebook*. Los usuarios de este tipo son generados con menos datos que los usuarios propios y tienen ciertas restricciones a la hora de modificar su perfil.

# 15. Usabilidad/UX

La realización de la aplicación frontend de PetFinder se ha inspirado de los principios del *Mobile First*. El diseño y la concepción de los elementos que dan forma a la aplicación PetFinder se han realizado priorizando los dispositivos móviles. Eso se traduce en el uso de patrones de diseño de interfaces (como el uso de cards o modales) que se muestre sin problemas y sin que se tengan que realizar grandes cambios en dispositivos.

En términos prácticos, la aplicación de este principio significa, por ejemplo, que cuando se codifica el comportamiento de un componente para diferentes medidas de pantalla, primero se codifica el comportamiento en pantallas pequeñas (por defecto) y luego se introducen las modificaciones necesarias a medida que el tamaño aumenta. No al revés.

Este principio de diseño supone una evolución del conocido como Responsive Design y, por ello mantiene sus premisas de adaptar el contenido a los diferentes anchos de pantalla de los diferentes dispositivos, desde pantallas anchas hasta las pantallas de los smartphones.

En relación con la información mostrada en la página principal de la aplicación, se ha procurada aportar al usuario una vista sintética, con la información justa de las mascotas y con el uso de colores e iconografía especial para identificar el estado o el tipo de cada animal perdido. De este modo, se intenta mantener la interfaz simple y entendible para el usuario.

La información detallada –tanto de las mascotas como de los usuarios o de las acciones– se encuentra disponible en páginas de la aplicación separadas y requiere que el usuario acceda a ellas, para visualizar todo el contenido. Con ello, se consigue mantener la aplicación simple y clara para el usuario.

En relación con los principios de diseño de interfaces aplicados en el diseño de la aplicación, es preciso destacar los siguientes:

- **Cards.** En la página principal de la aplicación, la información de las diferentes mascotas se agrupa de forma lógica en tarjetas o Cards que se muestran agrupadas en una fila (el número de tarjetas por fila varía en función del ancho de la pantalla) y que permiten acceder al detalle de la mascota y declarar una acción sobre esa mascota.
- **Continuos Scroll**. El *Continuos Scroll* es una técnica que permite cargar contenido en una página dinámicamente a medida que el usuario se desplaza haciendo *scroll* en la página.

De este modo el usuario percibe que el contenido de la página no termina hasta que se acaba la información, por bien que cada vez que se realiza una nueva carga de contenido, una petición es enviada al servidor. Esta técnica se aplica en la página principal de la aplicación.

- **Modals** El uso de modales permite atraer toda la atención del usuario hacia un contenido determinado que se muestra superpuesto al contenido previo. El contenido de las modales debe, además, reducido para que se muestre correctamente en los smartphones. En la aplicación, se usa esta técnica para recoger información a la hora introducir una acción sobre una mascota.
- **Carrousel.** El uso de *carrouseles* para mostrar imágenes es una técnica ampliamente utilizada que permite, entre otras cosas, temporizar la carga de imágenes a medida que el usuario las visualiza para, de ese modo, reducir la cantidad de imágenes descargadas cuando, por ejemplo, el usuario no visualiza más que la primera imagen.

En relación con el diseño de la navegación en la aplicación se ha pensado en dos formas distintas de navegación en función del ancho de la pantalla para maximizar la usabilidad de la aplicación tanto en grandes dispositivos como en *smartphones*.

- **Horizontal Menu Dropdown.** En el caso de las grandes pantallas, la aplicación se muestra con un menú horizontal de navegación que se encuentra en la parte superior de la aplicación. Se trata de un menú simple que ocupa todo el ancho de la pantalla y que contiene pocos enlaces directos un botón desplegable para que el usuario pueda modificar sus datos o desconectarse de la aplicación.
- **Lateral Sidebar Menu.** En el caso de los pequeños dispositivos, se ha optado por introducir un sidebar escondido que se abre con un *Burger button* y que permite ganar más espacio para mostrar las mismas opciones de menú que para los grandes dispositivos en un formato más claro y espaciado.

Por último, en lo concerniente a los formularios, también se han aplicado varios principios y patrones de diseño de interfaces.

- **Password Strength Meter.** En la página de registro se ha introducido un medidor de seguridad de la contraseña introducido por el usuario. Este principio tiene por objetivo que el usuario introduzca contraseñas más robustas y seguras frente a un ataque.
- **Calendar Picker**. En la pagina de registro y en el proceso de creación de una nueva mascota (como perdida) se ha incluido el uso de *DatePickers* en los inputs de tipo fecha con el objetivo

de simplificar la introducción de fechas y evitar problemas relacionados con el formato de estas.

- **Fill in the Blanks.** Este principio hace referencia a la gestión de los inputs cuando estos se muestran vacíos. En el diseño de la aplicación se ha previsto el uso de valores existentes para alimentar los campos de un formulario cuando el usuario este actualizando o modificando la información, así como el uso de *placeholders* en aquellos casos en que se trata de añadir nueva información al sistema.
- **Good Defaults.** Este principio de diseño esta bastante ligado al anterior. En este caso, al insertas una nueva mascota, por ejemplo, la opción de tipo de mascota preseleccionada es la del perro, puesto que, en España, el perro es la mascota más común. Este es un ejemplo de la importancia de preseleccionar y usar valores por defecto correctos para ahorrar tiempo al usuario y mejorar su experiencia de usuario.

# 16. Seguridad

El esquema de seguridad aplicado en la aplicación *PetFinder* tiene varias capas que se superponen y que persiguen dos objetivos; por un lado, aislar la aplicación y los sistemas con los que interactúa para reducir al máximo el riesgo de intrusión (por ejemplo, el riesgo de intrusión directa en el servidor de base de datos) y, por otro lado, proteger el flujo de datos servidos por la API para prevenir en este punto el robo de información de clientes.

- **Aislamiento del sistema.** Para garantizar el funcionamiento seguro de los componentes de la aplicación, es necesario que estos trabajen de forma aislada y que, desde el exterior, no se pueda acceder directamente a ellos. Esto se consigue mediante el uso de *docker networks*. Utilizando la tecnología de *docker-compose*, los contenedores que contienen los diferentes componentes de la aplicación se crean, por defecto, dentro de una red virtual creada por Docker. Los contenedores que se encuentran en ella están aislados de otros contenedores y de la red del servidor y la forma de acceder a estos contenedores desde el exterior es mediante el mapeo de puertos entre los contenedores y el host.

  De este modo, al crear la estructura de contenedores con el fichero *docker-compose.yml* se puede decidir que puertos de cada contenedor se van a abrir al exterior (mediante el uso de la instrucción *ports*) y que puertos van a quedar expuestos dentro, únicamente, de la red creada por Docker (esto mediante la expresión *expose*, si bien no es necesario indicarlos).

  Esta encapsulación en forma de red permite aislar de forma efectiva los componentes de la aplicación de otros contenedores y de conexiones entrantes del exterior.

- ***Transport Layer Security (TLS).*** Los datos que la API expone son intercambiados con los clientes (entre ellos, la aplicación cliente desarrollada en Angular) por medio de HTTP. HTTP es un protocolo de comunicación cliente-servidor que sirve de base para el desarrollo moderno de aplicaciones web y APIs pero que presenta un problema elemental, y es que los datos que se envían por medio de HTTP no son encriptados ni cifrados y se envían en plano: tal como son introducidos por el usuario. No es necesario indicar lo fácil que seria "esnifar" el trafico de una red y robar información sensible.

  Para evitar este problema potencial de seguridad, se aplica TLS, un protocolo de cifrado de información basado en el uso de certificados que permite cifrar y descifrar la información enviada con un par de claves asimetrías (una clave publica que utiliza el cliente para cifrar y

otra clave privada que el servidor aplica para descifrar los mensajes entrantes). La combinación de estos dos protocolos ha llevado, con el tiempo, a la creación del protocolo HTTPS (el candado que aparece en la barra de navegación de los navegadores) y que permite verificar la identidad de un sitio web y proteger la información que se envía a través de internet.

- ***Json Web Token (JWT).*** Tras haber analizado el aislamiento del sistema y el uso de HTTPS como tecnología de cifrado de la información que transita la red entre el servidor y sus clientes es momento de ver como se protege la sesión y la identidad del usuario en la aplicación.

  Actualmente existen varias formas para gestionar la autenticación y guardar la información del usuario y la sesión en una aplicación, aunque pueden agruparse en los métodos en que el servidor conserva información de la sesión entre varias peticiones (*statefull*, en inglés) y aquellos en que el servidor no conserva información del usuario entre peticiones (*stateless*).

  Para el desarrollo de la aplicación se ha optado por un modelo *stateless* usando el estándar JWT para identificar las peticiones. Este estándar se basa en el uso de token firmados en el servidor con una fecha de expiración para identificar al usuario. El token se genera en el servidor y se envía al cliente cuando este se autentica en la aplicación. Cuando el cliente debe realizar una petición con autenticación, debe enviar el token en la petición dentro del encabezado *Authorization* y precedido de la palabra clave *Bearer.*

  El contenido del token se firma mediante una clave o un certificado, por lo que, si el cliente intentara modificarlo, la firma no coincidiría y el servidor rechazaría la petición. Este mecanismo permite que el servidor solo deba hacer una verificación del token en lugar de guardar en memoria la información de la sesión a la vez que traspasa una parte importante de la gestión de la sesión y la identificación del usuario al cliente.

# 17. Tests

Como se ha indicado en apartado precedentes, PetFinder esta formada por dos aplicaciones: un backend que sirve de API y un frontend que, como una aplicación independiente consume y utiliza los datos de esta API. Esto hace que se deban realizar pruebas a dos niveles: la API y el cliente.

Para testear la API se han realizado tests funcionales mediante el uso de Postman, un cliente HTTP ampliamente utilizado, con el objetivo de testear funcionalidades de la API, como, por ejemplo, la autenticación o la gestión de errores.

En paralelo, en la aplicación del backend se han creado tests unitarios para comprobar la capa de controladores de la API, importante de cara a cambios futuros en la estructura de los servicios de la API.

En lo que concierne al frontend de PetFinder, se han realizado tests funcionales y de usabilidad en los tres navegadores con mayor cuota de uso: *Google Chrome*, *Mozilla Firefox Developer Edition* y *Microsoft Edge*.

# 18. Versiones de la aplicación/servicio

La realización de este proyecto termina con la entrega de dos aplicaciones independientes y separadas con versiones finales distintas.

- **Aplicación PetFinder Backend.** La versión en producción al momento de la entrega del proyecto es la versión 1.0.0-FINAL.
- **Aplicación PetFinder Front.** La versión en producción el momento de entregar el proyecto es la versión 1.0.0.

# 19. Requisitos de instalación/implantación/uso

Para poder instalar la aplicación PetFinder en un servidor, se deben cumplir 2 requisitos.

1. **Tener Docker instalado.** El servidor en que se pretenda instalar la aplicación debe tener instalado Docker en su versión 1.13 o superiores, para que sea compatible con la versión del fichero *docker-compose.yml* utilizada.
2. **Tener conexión a internet.** Es necesario que el servidor se pueda conectar a internet, puesto que, para instalar la aplicación, Docker debe descargar las imágenes del repositorio remoto DockerHub.

# 20. Instrucciones de implantación

La instalación de la aplicación en un servidor que cumpla con los requisitos es rápida y sencilla.

1. Crear un repertorio con el nombre *petfinder* y copiar dentro el fichero *docker-compose.yml* de la aplicación.
2. Ejecutar *docker-compose pull* para recuperar las imágenes del repositorio remoto: si las imágenes no se encuentran ya en memoria se descargarán 4 imágenes.
3. Para lanzar el *stack* de contenedores que conforman PetFinder, hay que navegar dentro del repertorio creado anteriormente y ejecutar el comando *docker-compose up -d.*
4. En caso de querer detener todos los contenedores de la aplicación, deben ejecutar el comando siguiente: *docker-compose down*.

# 21. Instrucciones de desarrollo

En este apartado se van a explicar las instrucciones para lanzar la aplicación de backend y la aplicación de frontend de PetFinder en modo de desarrollo en nuestra maquina local.

## 21.1 Aplicación Backend

Para lanzar la aplicación backend es necesario tener instalado en el ordenador el JDK 1.8 y Docker. A continuación, adjunto la línea de Chocolatey (un gestor de paquetes desde línea de comandos para Windows) para instalar estos componentes en Windows.

```
choco install jdk8

choco install docker-desktop
```

A continuación, es necesario lanzar dos contenedores de Docker, se trata de los contenedores de MongoDB y de MinIO, las dos dependencias externes de la aplicación. La primera es una base de datos NoSQL y la segunda herramienta es un servidor de almacenaje de ficheros (parecido a Amazon S3).

```
docker run -p 9000:9000 -d \
  -e "MINIO_ACCESS_KEY=AKIAIOSFODNN7EXAMPLE" \
  -e "MINIO_SECRET_KEY=wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY" \
  minio/minio server /data
```

```
docker run -d -p 27017:27017 --name mongodb mongo:4.0.4
```

Es importante no modificar los puertos ni las variables de entorno que se pasan a MinIO puesto que estas se encuentran también en la configuración de la aplicación y cualquier modificación debería también realizarse sobre la aplicación Spring Boot.

Una vez lanzados los contenedores, hay que ejecutar la aplicación Spring Boot.

```
./mvnw clean spring-boot:run
```

Cuando la aplicación se inicie correctamente, debería ser posible cargar la documentación Swagger de la API en la siguiente dirección:

*http://localhost:8080/api/swagger-ui.html#/*

## 21.2 Aplicación Frontend

Para arrancar la aplicación Angular que se ha desarrollado es necesario tener instalado Node, Angular y Typescript.

```
choco install nodejs
npm install -g @angular/cli
npm install -g typescript
```

Una vez instalados estos paquetes, se deben descargar las dependencias de la aplicación Angular que se detallan en el fichero package.json. Para ello, en la raíz del proyecto ejecutaremos la siguiente instrucción.

```
npm install
```

Una vez la instalación de dependencias haya terminado, ya podremos compilar y ejecutar la aplicación Angular mediante el comando siguiente.

```
npm run start
```

Este comando debe abrir una ventana en el navegador por defecto del sistema con la página http://localhost:4200 cargada con la aplicación.

# 22. Bugs

- Error al cargar los assets de Leaflet en la aplicación frontend compilada. (**corregido 14-Dic**)
- Error al generar el enlace asociado al botón de redirección que se envía en los emails (**corregido 9-Dic**)
- Error al iniciar la aplicación cuando el idioma del navegador no se incluye en los lenguajes para los que PetFinder tiene traducción. (**corregido 26-Dic**)

# 23. Proyección a futuro

La aplicación PetFinder puede ser ampliada con varias funcionalidades o variaciones de esta que le permitan realizar o extender sus objetivos de forma más eficaz. Estas son las mejoras que podrían hacer evolucionar la aplicación:

- **Adopción de mascotas**. Aunque la aplicación inicial solo prevé los estados de perdido, avistado y encontrado, sería una buena evolución añadir dos estados más: "para adopción" y "adoptado" para gestionar la adopción de mascotas y, de este modo, ser una opción viable para que las protectoras puedan encontrar personas responsables dispuestas a adoptar.

- **Reconocimiento visual del animal.** Una posible evolución para PetFinder seria agregar algoritmos de reconocimiento facial de perros y gatos que permita a los usuarios buscar con una imagen y saber si el animal ya se ha declarado como perdido en la aplicación.

- **Aplicación móvil**. Aunque el diseño de la aplicación frontend se ha realizado siguiendo el principio de *Mobile First Design*, para poder explotar todas las funcionalidades de los smartphones y para captar más usuarios, la realización de una aplicación móvil (usando, por ejemplo, Ionic 6) parece una buena idea.

- **Internacionalización (i18n)**. La aplicación se ha concebido con traducciones para el español y el inglés. Anadir nuevas traducciones siempre es una forma relativamente sencilla de abrir la aplicación a nuevos mercados y llegar a más personas.

- **Despliegue en la nube y escalado de la aplicación.** Este proyecto ha desarrollado las dos aplicaciones que conforman PetFinder y las ha desplegado mediante la containerizacion y el uso de *Docker Compose*. Sin embargo, para lograr resultados profesionales en aplicaciones sometidas a un uso exigente, es una buena idea desplegar las aplicaciones en proveedores de servicios en la Nube (como *Amazon Web Services*, *Microsoft* o también *Google Cloud Platform*) y añadir mecanismos de escalado horizontal, de modo que en caso de que haya un pico de uso de la aplicación, nuevas instancias de esta se creen de forma automática para responder a la demanda.

# 24. Presupuesto

A continuación, se expone el presupuesto desglosado del proyecto.

| **Presupuesto - PetFinder** | | | |
|---|---|---|---|
| Concepto | Coste (€) | Unidades | Total (€) |
| Electricidad Ordenador (0,1 kWh) | 0,012 | 130 | 1,56 |
| Tarifa Horaria de Desarrollador (50 € / h) | 60 | 130 | 7800 |
| iluminación (0,001 kWh) | 0,0012 | 130 | 0,156 |
| conexión Internet | AMORTIZADA | | |
| Uso Ordenador | AMORTIZADA | | |
| Uso Servidor de Despliegue | AMORTIZADA | | |
| Dominio web | 12 | 1 | 12 |
| SUBTOTAL | | | 7813,716 |
| IVA | | 21 | 1640,88 |
| TOTAL | | | 9454,60 |

En primer lugar, es necesario indicar que la tarifa de desarrollo se ha fijado en un precio de 60 euros por hora. Aunque puede parecer mucho, se debe tener en cuenta que se trata de un precio que engloba aspectos como las vacaciones anuales y las cotizaciones a la Seguridad Social. Además, se trata de un precio de mercado real y no muy alejado del precio que fijan las consultoras tecnológicas por sus desarrolladores.

En el presupuesto se puede ver que hay varios conceptos que no tienen un desglose monetario y que aparecen como “Amortizado”. Eso significa que este concepto no conlleva un coste adicional destacable o que, como es el caso de la conexión a internet, el coste no del concepto no depende ni varía en función del proyecto.

Por último, asumiendo que este proyecto quisiera venderse a un tercero, debería añadirse el Impuesto sobre el Valor Añadido (IVA) en su tipo impositivo habitual del 21%. El IVA es un impuesto de tipo indirecto que graba el consumo o prestaciones de bienes o servicios.

Tras aplicar todos los conceptos, el precio final de la realización del proyecto, asumiendo una carga aproximada de 130 horas de trabajo, es de 9.545 euros.

# 25. Análisis de mercado

La audiencia potencial a la que la aplicación PetFinder se dirige son todos aquellos hogares que tienen una o más mascotas. Se calcula que a nivel europeo más de 85 millones de hogares tienen una o más mascotas a finales de 2019. En España, el 40% de hogares tienen mascota: este es el público o audiencia potencial de la aplicación.

En cuanto a la segmentación, es difícil establecer un criterio para segmentar el público más allá de si efectivamente poseen o conviven con una mascota. Como opción, se podría segmentar por franjas de edad excluyendo del público de la aplicación, los más jóvenes (puesto que no gestionan el hogar) y los mayores de 65 años, dado que en esa franja de edad el uso de internet i las nuevas tecnologías son todavía bajos.

En cuanto a la competencia, existe una aplicación llamada WizaPet que ya realiza la mayor parte de las funciones que se proponen en PetFinder: implementa el uso de proveedores para la autenticación, gestiona perdidas de mascotas e incluso, y lo que parece más interesante, implementa un sistema de recompensas mediante *Stripe* para incentivar el hallazgo de mascotas perdidas. Entre otras prestaciones.

Esta aplicación se encuentra en su versión 3.2.60 y está disponible como aplicación web y, también, como aplicaciones nativas para Android e iOS y, en Google Play, está calificada con una puntuación de 4.1 sobre 5 con más de 780 opiniones.

En relación con los márgenes de precio, dado que la aplicación no comercializa ningún servicio propio, no tiene sentido abordar este tema.

# 26. Marketing y Ventas

Este proyecto, como se puede constatar en la sección de objetivos, no pretende convertirse ni generar negocio con su uso, por lo que no existe ninguna política ni estrategia de ventas. Las funcionalidades ofrecidas por la aplicación PetFinder no están sujetas a ninguna contraprestación económica.

No obstante, es necesario señalar que se debe encontrar una fuente de ingresos que permita suplir los siguientes gastos:

- **Presupuesto**: el presupuesto refleja una serie de gastos que, en la medida de lo posible, deben ser cubiertos.
- **Puesta en producción y mantenimiento en la nube**. En caso de que la aplicación sea puesta en producción de forma profesional (este proyecto ha desplegado la aplicación en un servidor personal), se deberán cubrir los gastos derivados de estas operaciones: alquiler de máquinas virtuales, espacio de almacenamiento, compra de dominios, etc.

Una mención aparte merece la publicidad necesaria para que, una vez la aplicación alcance un cierto grado de madurez y fiabilidad, la aplicación empiece a ser utilizada por los usuarios. Aunque una parte importante de esta publicidad pasa por presentar la aplicación a Ayuntamientos y Protectoras y otros establecimientos que trabajen con mascotas, es innegable que una parte de la publicidad debe ser hecha mediante campañas de anuncios en las redes

A pesar de que se aleja del ámbito de este proyecto, se podría empezar utilizando la plataforma *Google Ads*, un completa herramienta gestionada por *Google* y que permite generar campañas y anuncios personalizados y mostrarlos en aquellas paginas o a aquellas personas que se consideran un público probable de la aplicación. Estas campañas tienen un coste que se debería ser cubierto.

Para cubrir estos costes de funcionamiento de la aplicación, se ha pensado en la introducción de publicidad en la página principal de la aplicación, de forma que no sea muy intrusiva para el usuario. Para llevar a cabo esta tarea, se puede recurrir a *Google AdSense*, el producto complementario de *Google Ads* que permite insertar anuncios en las páginas web para monetizar su uso.

Este tipo de monetización es el más utilizado y puede ser aplicado fácilmente en la aplicación puesto que es un tercero el que gestiona la inserción de los anuncios.

## 26.1. Métricas de la aplicación

Adicionalmente, y con el objetivo de aumentar la trazabilidad de la interacción y los eventos generados por los usuarios en la aplicación web, se ha realizado una implementación de la herramienta *Google Analytics* en la aplicación.

*Google Analytics* es una plataforma creada por Google en el año 2005 cuyo objetivo principal es realizar un seguimiento de la actividad de los sitios web y de sus usuarios y proporcionar y proporcionar a los administradores o responsables de los sitios estadísticas e información detallada de la actividad.

Esta plataforma, eminentemente orientada al *ecommerce*, se basa en el envió de eventos asíncronos des de los sitios web a partir de la interacción de los usuarios. Existen dos categorías diferenciadas que aportan información diferente sobre la actividad del usuario en el sitio.

- ***PageView.*** El *PageView* es un mensaje que se envía a Analytics cada vez que el usuario carga una nueva página de la aplicación. Esto permite recopilar información sobre la navegación de los usuarios y las páginas más visitadas de una aplicación. Aunque, al tratarse de una *Single Page Application*, el cambio de página se limita un cambio en la aplicación cliente (no hay petición al servidor de una nueva página), Angular permite suscribirse a los eventos de navegación y enviar esta información a Angular de forma precisa cuando se produce un cambio de página.
- **Eventos**. Además de los pageView, *Google Analytics* trabaja también con eventos. Los eventos son mensajes que se envían desde la aplicación a la plataforma cuando se produce una interacción del usuario con el sitio. Ejemplos clásicos de eventos seguidos son las compras o la acción de añadir al carrito (*purchase* y *add_to_cart*). Los eventos, sin embargo, son personalizados en función de cada aplicación.

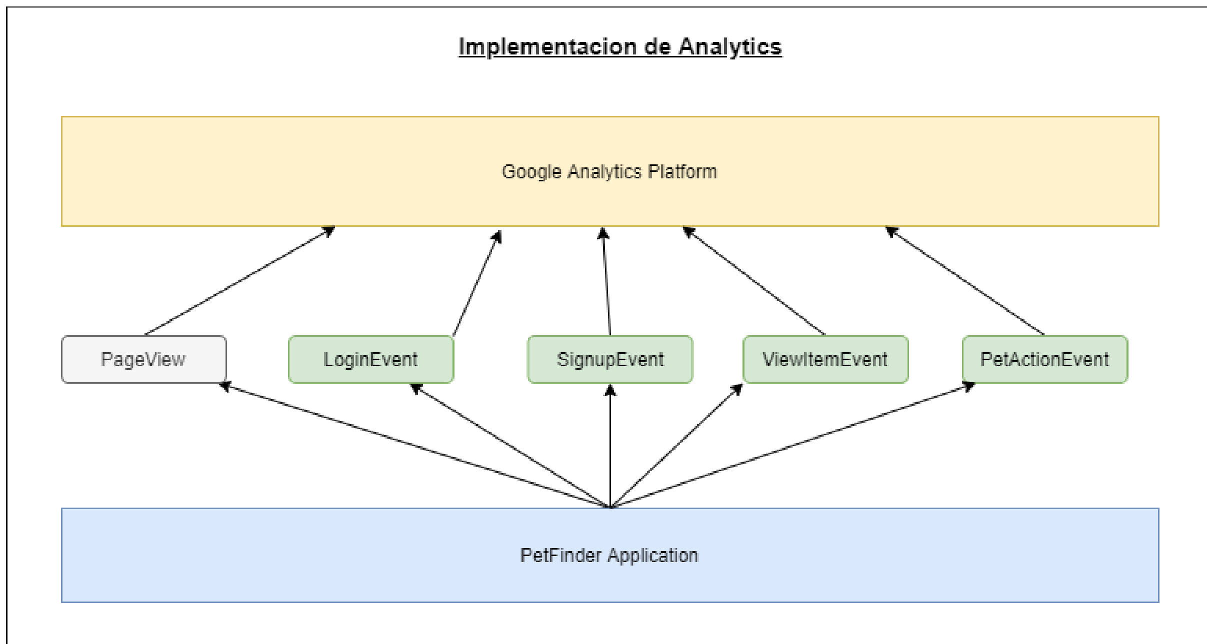



*Imagen: Implementación de Google Analytics en PetFinder*

Como se puede ver en la imagen, para la implementación inicial del seguimiento con Analytics se han escogido 4 eventos para monitorear la interacción general de los usuarios con la aplicación: *LoginEvent* y *SignupEvent* para medir el grado de captación de usuarios y de retorno de usuarios ya registrados en la aplicación y los otros dos eventos, *ViewItemEvent* y *PetActionEvent* para medir la interacción de los usuarios al mirar el detalle de las mascotas y al introducir nuevas acciones, como un avistamiento.

# 27. Conclusiones

Para elaborar las conclusiones de este proyecto se han tomado como referencia los objetivos fijados al principio del proyecto para ver en qué grado estos se han alcanzado.

En relación con los objetivos principales que perseguía este proyecto, considero que se han alcanzado los objetivos establecidos. El estudio de los casos de uso principales, así como el diseño de la aplicación han permitido dar una respuesta a los usuarios de PetFinder: aquellos que han perdido su mascota y aquellos que han visto un animal de compañía perdido en la calle.

Tras haber realizado y desplegado la aplicación backend de PetFinder y el cliente angular que sirve de frontend, puedo afirmar que este proyecto me ha permitido aprender y profundizar en los conceptos claves en la realización de aplicaciones, desde la arquitectura de software hasta el uso de contenedores para orquestar los despliegues.

Enlazando con los objetivos secundarios que se habían fijado, el trabajo de estudio y realización del despliegue de la aplicación –mediante el uso de contenedores Docker y Docker Compose– me han permitido aprender y mejorar los conocimientos que ya tenía sobre esta tecnología de containerizacion y su potencial para la gestión de aplicaciones en producción y en la nube, puesto que, el eslabón siguiente seria el uso de orquestadores como *Docker Swarm* o el conocido *Kubernetes* para la creación de clústeres de servidores sobre los que desplegar. Se trata de una tecnología muy utilizada a nivel profesional y que, con este proyecto, he podido utilizar en una situación real.

El uso de la información georreferenciada –coordenadas– planteaba un reto al principio del proyecto porque esta información debe ser almacenada e indexada de una forma especial para que la base de datos pueda usarla en peticiones espaciales. Al principio hubo ciertos problemas con el formato de los datos almacenados y con la creación de los índices geoespaciales en MongoDB, pero gracias a la documentación oficial de MongoDB y a la ayuda de la comunidad, estos problemas fueron resueltos y se pudo empezar a explotar esos datos. Se trata sin duda de un punto importante de la aplicación, que le da cierta complejidad, pero que me ha permitido aprender a trabajar con coordenadas (desde el frontend, hasta la base de datos).

El uso de sistemas de notificaciones y de APIs de terceros (Google y Facebook) han sido una oportunidad de interconectar la aplicación con otros servicios y de aprender, por ejemplo, como explotar la API de Facebook o de Google para solicitar información de la persona que utiliza la

aplicación. También me ha permitido aprender cómo obtener claves API para poder utilizar esos productos.

Por último, la calidad es el último de los objetivos secundarios y, a la vez, es el punto del que estoy menos satisfecho. Aunque se puede afirmar que la aplicación funciona y no presenta problemas destacables (al menos hasta la fecha), el hecho de que no se hayan implementado tests para la mayor parte del código y las funcionalidades que la aplicación ofrece, hace que la calidad de la aplicación, sobre todo de cara futuras evoluciones, no sea muy buena.

Me hubiera gustado concebir una aplicación con las pruebas implementadas mediante el uso de metodologías como *TDD (Test Driven Development)* en que el proceso de desarrollo es “pilotado” desde la clase de prueba, o *DDD (Domain Driven Development)* en que se sitúa a los dominios como el centro integral de la lógica de la aplicación y se generan tests para los casos de uso.

# Anexo 1. Entregables del proyecto

A continuación, se indican los entregables que se presentan junto con esta memoria y que contienen el código de las aplicaciones, así como el mecanismo de despliegue de PetFinder mediante Docker Compose.

- **petfinder-back.** Es una carpeta que contiene el repositorio GIT con el código de la aplicación backend de PetFinder. Contiene la lógica de la aplicación, la capa de acceso a base de datos y los servicios expuestos en la API.
- **petfinder-front.** Una carpeta que contiene el repositorio GIT de la aplicación frontend de PetFinder creada con Angular.
- **petfinder-deploy.** Se trata de una carpeta con el script de despliegue de la aplicación (incluidas sus dependencias) mediante el uso de Docker Compose.

La aplicación producto de este proyecto se encuentra desplegada en la dirección https://petfinder.ricardmolinaferret.com/

# Anexo 2. Libro de estilo

Como se ha explicado en apartados anteriores, el diseño del frontend de PetFinder se apoya fuertemente en el uso y la personalización del *framework Bootstrap 4*. A continuación, se incluye la hoja de estilos con las personalizaciones de Bootstrap para PetFinder. (fichero_*petfinder-theme.scss*)

```
1  $font-family-base:Montserrat;
2
3  $enable-grid-classes:true;
4
5  //Bootstrap Colors
6  $primary: #27a2fc;
7  $secondary: #a0b7bc;
8  $success: #37bd74;
9  $danger: #eb6259;
10 $info: #7ebcfa;
11 $warning: #ff9b37;
12 $light: #eef0f2;
13 $dark: #3c4055;
14
15 //PetFinder Colors
16 $light-shades: #EEECE0;
17 $light-accent: #848B94;
18 $brand: #C17C4F;
19 $brand-danger: #a83640;
20 $dark-accent: #8A5E3B;
21 $dark-shades: #262222;
22
23 $google: #dd4b39;
24 $facebook: #4267B2;
25
26
27 $theme-colors: (
28   "primary":$primary,
29   "secondary":$secondary,
30   "success":$success,
31   "danger":$danger,
32   "info":$info,
33   "warning":$warning,
34   "light":$light,
35   "dark":$dark,
36   "light-shades": $light-shades,
37   "light-accent": $light-accent,
38   "brand": $brand,
39   "dark-accent": $dark-accent,
40   "dark-shades": $dark-shades,
41   "google": $google,
42   "facebook": $facebook,
43   "brand-danger": $brand-danger,
44 );
45
46
47 $body-bg: $light-shades;
48 $spacer:1.3rem;
49 $border-width:1px;
50 $btn-border-radius:.25rem;
51 $enable-shadows:true;
52 $enable-gradients:false;
53 $border-radius-lg:.4rem;
54 $border-radius-sm:.3rem;
55 $link-hover-decoration:none;
56 $input-btn-focus-box-shadow: none;
57 $input-btn-focus-width: 0;
58
59 //Paginator
60 $pagination-active-color: white;
61 $pagination-active-bg: $brand;
62 $pagination-color: $dark;
63 $pagination-hover-color: $brand;
```

# Anexo 3. Glosario

- **API.** *Application Programming Interface.* Se trata de funciones de código diseñadas para exponer o recibir cierta información de otros sistemas o aplicaciones ofreciendo para ello, una capa de abstracción.
- **Base de datos.** Una base de datos es un conjunto de datos almacenados dentro de un mismo contexto de forma sistemática con la intención de ser explotados posteriormente.
- **Docker**. Se trata de un sistema que permite crear y gestionar contenedores para virtualizar aplicaciones. Los contenedores son, por definición, más ligeros y eficaces y concebidos específicamente para las aplicaciones en comparación con las máquinas virtuales.
- **HTML.** *Hypermedia Text Markup Language.* Se trata del lenguaje de codificación estándar para la construcción de contenido web.
- **JWT.** *Json Web Token.* Se trata de un estándar de creación de tokens de acceso basado en el uso de JSON.
- **REST.** *Representational State Transfer.* Es un tipo de arquitectura de *software* basado en el envió de contenidos mediante el uso de lenguajes y protocolos hipermedia. Hoy en día constituye un estándar en la concepción de sistemas de intercambio de información.
- **Servidor.** Un servidor es una aplicación cuya función es exponer cierta información a uno o más clientes. Se habla también de servidor para mencionar ordenadores cuya función principal es alojar y exponer contenido a clientes a través de las redes.
- **SGBD.** Sistema gestor de base de datos. Son programas que permiten la gestión y explotación de una base de datos. Hay muchos ejemplos como *MySQL, PostgreeSQL* o la que se ha usado en este proyecto *MongoDB.*