GENERALITAT VALENCIANA
Conselleria d'Educació, Cultura i Esport

I.E.S. CONSELLERIA

Unión Europea
Fondo Social Europeo
El FSE invierte en tu futuro

Monestir de Poblet, s/n 46015 – Valencia
Cód. 46022257 Tl. 96 120 61 00
Fax 96 120 61 01

INSTITUTO DE EDUCACIÓN SECUNDARIA
I.E.S. CONSELLERIA -

FAMILIA PROFESIONAL DE INFORMATICA Y COMUNICACIONES

Creación de una web dedicada a la gestión documental

PROYECTO DE DESARROLLO DE APLICACIONES WEB
NÚMERO INF-20-05

Autor: Iván Córdoba Donet
Tutor individual: Javier García

Índice

1. Introducción

1.1. Justificación

Docuco es un gestor documental que pretende facilitar la tarea de administrar los diferentes tipos de documentos financieros de la manera más cómoda posible para el usuario. Este *software* solventa la complicada tarea de tener un control de los documentos, ya que en muchas ocasiones se pierden o no están bien organizados. Además, proporciona diferentes análisis que resultan de utilidad, siendo así una herramienta muy completa para cualquier empresa, autónomo o particular. Se trata de una herramienta intuitiva y con un diseño agradable que proporciona una buena experiencia de usuario, sin necesidad de invertir tiempo en complicados aprendizajes.

1.2. Contexto

Este *software* está relacionado con varios de los módulos impartidos en clase, puesto que al estar desarrollado desde cero se involucran diversos aspectos.

En relación al módulo de "Diseño de Interfaces Web", se crean varios *wireframes* iniciales para comprender el flujo del usuario en la web y, de esta manera, validarlo antes de empezar a desarrollar, así como también el logo y el diseño general de la web. Al generar un *front end* en Angular, aprovechando las ventajas que ofrece este *framework*, se conecta con "Desarrollo en Entorno Cliente". En cuanto a "Desarrollo en Entorno Servidor" se ha desarrollado una API* en Laravel, de forma que este *back end* trabaja con la base de datos, ofreciendo de la manera más simple los datos al *front end*. También, al crear las diferentes entidades en el esquema relacional UML* y administrar la base de datos en PostgreSQL, se relaciona con el módulo de "Gestión de Base de Datos". Por último, está vinculado con "Despliegue de Aplicaciones Web", al configurar Nginx para servir la web y al desplegar la aplicación en los servidores de AWS (Amazon Web Service).

1.3. Objetivos

- Creación de un *software* para gestionar documentos financieros
- Utilización de Angular para el *front end* de la web
- Utilización de Laravel para el *back end* de la web
- Utilización de PostgreSQL como gestor de base de datos
- Utilización de Nginx como *proxy* inverso para servir la web
- Implementación de test automáticos
- Implementación de Arquitectura hexagonal junto a DDD (Domain Driven Design) en el *back end*
- Desarrollo del proyecto contenido en Docker utilizando Docker Compose
- Despliegue de la web en la nube usando AWS

1.4. Organización

Al inicio se puede ver la planificación del proyecto, desde cómo se ha pensado el sistema de negocio para sacar rentabilidad, hasta las herramientas usadas para su desarrollo. A continuación, en el desarrollo del proyecto, se empieza explicando más a fondo algunos conceptos importantes, cuáles han sido los pasos previos a implementar las funcionalidades y cómo ha evolucionado desde la idea inicial. Seguidamente, se observa la evaluación, donde se comentan los objetivos conseguidos y posibles mejoras. En la conclusión, se hace una reflexión sobre el proyecto de forma general y, por último, en la sección de referencias se encuentran los enlaces en los cuales se basa esta documentación. El código completo del *software* se encuentra en GitHub, https://github.com/nabby27/docuco, de manera pública para poder verlo y hacer cualquier comprobación. En esta documentación no se verá casi código ya que se trata de una explicación sobre cómo se ha hecho y qué se ha usado.

2. Planificación

2.1. Idea de negocio

El plan de negocio que se pretende establecer con este *software* es mediante una suscripción mensual. Hay tres modelos de suscripción dependiendo de las funcionalidades que se necesite. El primer modelo es el **Básico**, mediante el cual únicamente se puede tener acceso a la gestión de los documentos, un único usuario y análisis gráficos genéricos sobre el balance financiero. En el segundo modelo, el **Estándar**, además de tener acceso a la gestión de los documentos se pueden gestionar diferentes usuarios. Y por último, el modelo **Avanzado**, añadiendo análisis gráficos personalizables y futuras funcionalidades. En cuanto a los precios de cada plan son: 5 €/mes el plan Básico, 10 €/mes el plan Estándar y 20 €/mes el plan Avanzado.

2.2. Aspectos técnicos

En cuanto a la forma de trabajar, en la parte del *back end* se seguirá la metodología TDD (Test Driven Development) y se utilizará DDD junto a la arquitectura hexagonal para mantener una base sólida a nivel se estructura de *software* (se explicará todo esto más adelante). En la parte del *front end,* Angular ofrece una estructura de módulos muy estable y escalable, por tanto es la que sigue la aplicación, aunque también se podrían utilizar los conceptos usados en el *back end*. El despliegue se realiza mediante los servicios de AWS. Además se siguen diferentes estrategias de test automatizados para asegurar la calidad del código.

A continuación se agrupan las herramientas usadas para este proyecto explicando brevemente cual es su función.

Front end		
Herramienta	**Descripción**	**Link**
Node.js	Entorno para ejecutar JavaScript, necesario para angular	https://nodejs.org/es/
Angular	Es un *framework* de JavaScript que permite desarrollar SPA (Single Page Application) de forma rápida	https://angular.io/
TypeScript	Lenguaje de programación con un superconjunto de funcionalidades para JavaScript	https://www.typescriptlang.org/
Angular Material	Librería de diseño para Angular con estilo material	https://material.angular.io/
Cypress	Herramienta para hacer test automáticos e2e desde el *front end*	https://www.cypress.io/
Chart.js	Librería de JavaScript para implementar gráficas en la web	https://www.chartjs.org/
Moment.js	Librería de JavaScript para el uso y manipulación de fechas	https://momentjs.com/
ng2-pdf-viewer	Librería de JavaScript para visualizar PDFs en la web	https://github.com/VadimDez/ng2-pdf-viewer

Back end / API		
Herramienta	**Descripción**	**Link**
Laravel	*Framework* de PHP para desarrollar aplicaciones web (en este proyecto se usa a modo de API)	https://laravel.com/
passport	Librería de PHP para la gestión de *tokens* de usuario	https://github.com/laravel/passport
php-unit	Librería de PHP para hacer test automáticos e2e y unitarios de la API	https://phpunit.de/
phpcpd	Librería de PHP para detectar código duplicado	https://github.com/sebastianbergmann/phpcpd
phpcs	Librería de PHP para indicar fallos de código estandarizado	https://github.com/squizlabs/PHP_CodeSniffer

phpcbf	Librería de PHP para solucionar problemas de código estandarizado	https://github.com/squizlabs/PHP_CodeSniffer

Base de datos		
Herramienta	**Descripción**	**Link**
PostgreSQL	Motor de base de datos	https://www.postgresql.org/
DBeaver	Editor SQL multiplataforma que permite trabajar con diferentes motores de base de datos	https://dbeaver.io/
RDS	Servicio de AWS que nos ofrece diferentes bases de datos para alojar en la nube	https://aws.amazon.com/es/rds/

Servidores y Hosting		
Herramienta	**Descripción**	**Link**
Docker	Permite contener el proyecto sin necesidad de instalar en local ningún programa necesario. Facilita la creación de servidor y entorno en local a la hora de desarrollar	https://www.docker.com/
Docker Compose	Orquestra diferentes contenedores *docker* como servicios	https://docs.docker.com/compose/
Nginx	Proxy inverso/servidor web para servir páginas web	https://www.nginx.com/
EC2	Servicio de AWS que nos ofrece servidores privados	https://aws.amazon.com/es/ec2/
Route53	Servicio de AWS para gestionar rutas del dominio hacia los servidores	https://aws.amazon.com/es/route53/

Otros		
Herramienta	**Descripción**	**Link**
Visual Studio Code	Editor de código simple, sencillo e intuitivo pero con mucha potencia	https://code.visualstudio.com/

Inkscape	Editor de gráficos vectoriales	https://inkscape.org/es/
Git	Sistema para el control de versiones del código	https://git-scm.com/
GitHub	Plataforma para alojar repositorios de proyectos utilizando Git	https://github.com/
Insomnia	Aplicación que realiza peticiones HTTP	https://insomnia.rest/
Trello	Tablero canvas para la gestión de las tareas a realizar	https://trello.com/es
Arquitectura hexagonal	Estructura organizativa del código	
DDD	Estructura organizativa del código	

3. Desarrollo del proyecto

3.1. Conceptos previos

Antes de detallar el proceso de desarrollo del *software* se explicarán algunos conceptos detalladamente para poder entender mejor cada paso y el por qué se hace de esa manera.

3.1.1. SOLID

Robert C. Martin recogió cinco directrices y creó el acrónimo mnemotécnico de estos principios, llamado principios SOLID. Estas directrices o principios facilitan a los desarrolladores la labor de crear programas legibles y sobretodo mantenibles.

En este proyecto se utilizan todo lo posible estos principios pero no se centra en ellos, ya que, como todo, los casos que se explican en estos los principios son genéricos y depende del contexto, la situación, los objetivos que se quieran, etc. No hay que seguirlos siempre a raja tabla. A continuación se explican por encima cada uno de estos principios.

La **S**, single responsability principle (principio de responsabilidad única), indica que cada unidad del proyecto (clase, función, componente, servicio….) debe ser responsable de una única cosa. Si esto no se cumple, al modificar una parte del código que tiene dos responsabilidades, se podría cambiar el comportamiento de una de ellas sin querer.

La **O**, open close principle (principio de abierto cerrado), establece que las entidades deben estar abiertas a su extensión pero cerradas a su modificación, lo que significa que ante peticiones de cambio en nuestro programa, hay que ser capaces de añadir funcionalidad sin modificar la existente .

La **L**, Liskov substitution principle (principio de substitución de Liskov), declara que una subclase debe ser sustituible por su superclase y el programa no debe fallar.

La **I**, interface segregation principle (principio de segregación de interfaz), indica que las clases no deberían verse forzadas a depender de interfaces si no usan todos los métodos de esta.

La **D**, dependency inversion principle (principio de inversión de la dependencia), estable que las dependencias deben estar en las abstracciones (interfaces), no en las concreciones (clases).

3.1.2. Arquitectura hexagonal

La aplicación web utiliza esta arquitectura de *software* en la parte *back end* con Laravel. La arquitectura hexagonal, también conocida como arquitectura de puertos y adaptadores, es un conjunto de reglas durante el desarrollo, las cuales proporcionan un código desacoplado de la infraestructura. Como cualquier arquitectura de *software* se utilizan diferentes patrones que guían a la construcción del *software*, permitiendo que en un equipo de desarrollo se sigan la mismas líneas de trabajo.

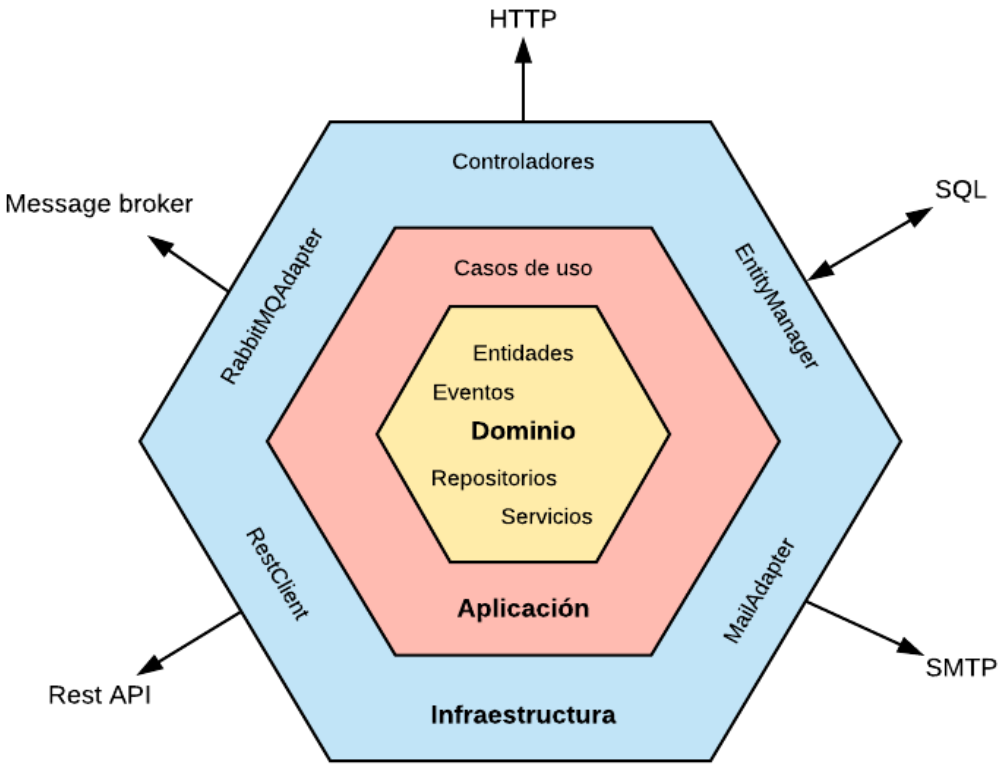

Figura 1: División de las capas de la arquitectura hexagonal

Como se puede observar en la Figura 1, el color azul determina la capa de **Infraestructura** de la que se hablaba anteriormente. Se llama infraestructura a toda la capa externa a la aplicación, en este caso el *back end*, como puede ser la base de datos, la capa de aplicación en el modelo OSI (HTTP, HTTPS, FTP…), conexión con APIs externas, etc. La siguiente capa hacia adentro es la de **Aplicación** (no confundir con la capa de Aplicación del modelo OSI, en arquitectura hexagonal representa la aplicación en si), es decir, los casos de uso, también llamados servicios de aplicación en DDD o Acciones en IDD, son las diferentes cosas que puede hacer la aplicación, de ahí su nombre. Y por último encontramos la capa de **Dominio**, que es el núcleo del *software*. En esta capa influye sobretodo el departamento de negocio, ya que es quien guía las funcionalidades del *software* y cómo serán estas. Entre otras cosas en esta capa se encuentran las entidades, los *value objects* y los repositorios (no las implementaciones, ya que si fuera así el dominio conocería la capa de infraestructura y eso no puede ocurrir, más adelante se explica). Por último, comentar que lo más importante de esta arquitectura no es solo la diferenciación de las capas sino el flujo de estas y cómo interactúan entre si. De esta forma, si la aplicación no conoce nada de la capa externa podemos intercambiar esa capa sin que afecte a nuestro dominio. En el siguiente diagrama se observa mejor.

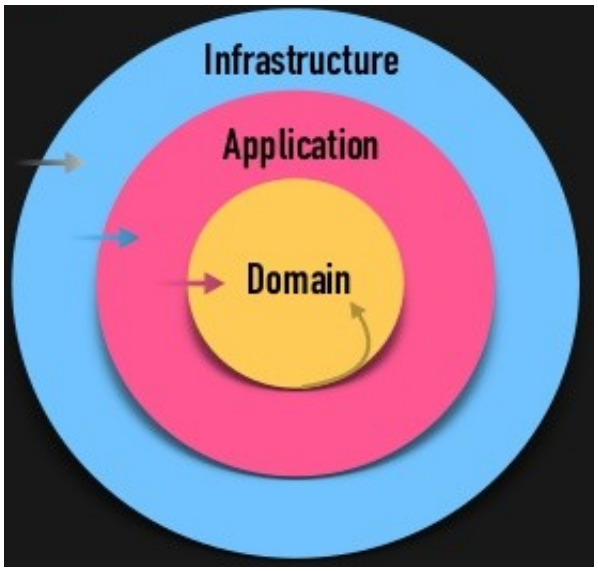

Figura 2: Flujo de la arquitectura hexagonal

Como indican las flechas, la capa externa conoce a las siguiente hacia dentro pero la interna no conocen a las capas de fuera. De este modo, la infraestructura solo sabe los casos de uso de la aplicación y el dominio; los casos de uso no conocen nada de la infraestructura pero sí conocen nuestro dominio; y por último, el dominio solo se conoce a sí mismo. Esto proporciona la ventaja de poder desacoplar lo que es nuestra aplicación (dominio) de las herramientas que se usan para llevarla a cabo. Para conseguir este flujo se hace uso de la inversión de dependencias de SOLID, que se verá cómo se aplica más adelante.

Otra ventaja de usar esta estructura es que facilita mucho a la hora de hacer test, sobretodo los unitarios, ya que permite *mockear* (falsear) la capa de infraestructura de una forma sencilla. Gracias a esta estructura también se hace mucho mas cómodo implementar los principios SOLID ya que guía a realizar estas buenas prácticas.

Para poder realizar esta arquitectura se ha explicado que se consigue gracias a la inversión de dependencias, pero, ¿qué es esto? La inversión de dependencias se basa en inyectar

mediante el constructor una instancia de clase que implemente la interfaz puesta como contrato. Se utilizan las interfaces para separar barreras con la infraestructura, se ve mejor con un ejemplo: Se tiene el caso de uso "GetOneDocumentAction", es una clase en cuyo constructor colabora una interfaz llamada "DocumentsRepository"; por otro lado existe una clase "DocumentsRepositoryPostgreSQL" que implementa la interfaz "DocumentsRepository". Al instanciar la clase "GetOneDocumentAction" se le pasa una instancia de "DocumentsRepositoryPostgreSQL". La clase "GetOneDocumentAction" no sabe qué implementación utiliza exactamente, solo conoce los métodos de la interfaz "DocumentsRepository". Con esto, si en un futuro se quiere cambiar de base de datos, se crearía una clase "DocumentsRepositoryPostgreSQL" que implemente la interfaz "DocumentsRepository" y se inyectaría al caso de uso "GetOneDocumentAction", que no se vería afectado por el cambio. También ayuda a la hora de hacer test unitarios, ya que se cambia esa implementación por una instancia de "DocumentsRepositoryInMemory" y así se puede falsear esa capa. De esta manera se consigue que el caso de uso de la aplicación ("GetOneDocumentAction") no conozca cómo se implementa la infraestructura ("DocumentsRepositoryPostgreSQL"), solo conoce el contrato en común que se tiene gracias a la interfaz ("DocumentsRepository").

```php
class GetOneDocumentAction
{
    private $repository;

    public function __construct(DocumentsRepository $repository)
    {
        $this->repository = $repository;
    }

    public function execute(int $user_group_id, int $document_id): ?Document
    {
        return $this->repository->get_one_document_by_user_group_id($user_group_id, $document_id);
    }
}
```

Figura 3: Clase que inyecta un colaborador

```php
class DocumentController extends Controller
{
    private $document_repository;

    public function __construct()
    {
        $this->document_repository = new DocumentsRepositoryPostgreSQL();
        $this->get_user_group_id_from_request_service = new GetUserGroupIdFromRequestService();
    }

    public function get_one_document(Request $request, int $document_id)
    {
        $user_group_id = $this->get_user_group_id_from_request_service->execute($request);

        $get_one_document_action = new GetOneDocumentAction($this->document_repository);
        $document = $get_one_document_action->execute($user_group_id, $document_id);

        if (isset($document)) {
            return response()->json(['document' => $document], 200);
        }
        return response()->json(['message' => 'Document not exist.'], 404);
    }
```

Figura 4: Inyección de dependencia mediante el constructor

Después de ver algunas de sus ventajas y características también hay que destacar que tiene una desventaja, y es el nivel de complejidad, ya que se incrementa al inicio del *software* respecto otras arquitecturas como pueden ser MVC (Model View Controller), MVVM (Model View View Model) o MVP (Model View Presenter), pero si se pretende escalar y añadir funcionalidades de forma iterativa vale la pena la curva de aprendizaje, puesto que luego es más sencillo seguir la misma guía inicial. Así que... ¿Cuándo no usar esta arquitectura? Esta arquitectura no es recomendable usarla para aplicaciones pequeñas, para aplicaciones que necesitan estar en un corto periodo de tiempo o en aplicaciones que no van a escalar.

3.1.3. DDD

Ya que para la API se usa una arquitectura que desacopla el dominio de todo lo externo y es el núcleo de nuestra aplicación web, podemos encajar perfectamente la filosofía de DDD, Domain Driven Design (diseño guiado por el dominio). DDD es un enfoque para el desarrollo que se centra en un lenguaje ubicuo con negocio/cliente y centrar la lógica en el dominio, y puesto que no define implementaciones, sino conceptos, es independiente a cualquier *framework* o lenguaje de programación. Esto resuelve el problema que ocurre normalmente, y es que, el cliente/negocio sabe mucho sobre los diferentes casos y el funcionamiento de estos (el dominio) pero no saben nada de la parte técnica del desarrollo. En cambio, muchas veces los desarrolladores saben cómo realizar las cosas pero no el funcionamiento que debería tener. Con un lenguaje común para ambas partes, incluso dentro del código, se soluciona esa comunicación e incluso facilita el mantenimiento y la escalabilidad del *software,* ya que el código será entendible. Para eso, un punto clave es que tanto los desarrolladores como los clientes/negocio tengan una relación estrecha y se entiendan bien

todos los conceptos para que no haya malentendidos. DDD tiene muchos conceptos para su buen funcionamiento como son las entidades (que no pueden ser anémicas), *value objects*, repositorios, agregados…

3.1.4. Test automáticos

Los test automáticos sirven para validar que una funcionalidad hace lo que corresponde y ayuda a la refactorización[1]. Si el comportamiento cambiase a la hora de refactorizar, los test fallarían. Hay diferentes tipos de test: unitarios, de integración y de funcionalidad (también llamados e2e, *end to end*).

Los **test unitarios** son los más rápidos ya que no tocan nada de entrada/salida, como pueden ser base de datos, protocolo HTTP, FTP, etc. Prueban la funcionalidad de forma unitaria pero completa, son el tipo de test que más tienen que haber según la pirámide de test ya que, aunque no nos proporcionen una gran cobertura (como sí hacen los test de funcionalidad), se ejecutan mucho mas rápido y eso da un gran valor para poder lanzar los test mientras se desarrolla. Los **test de integración** se basan sobretodo en comprobar que la comunicación entre los diferentes componentes sea la correcta. De estos tienen que haber menos que test unitarios pero más que los e2e. Por último, los **test de funcionalidad o e2e** son los que realizan más cobertura del código, pero son más lentos puesto que prueban desde el principio hasta el final del flujo del código, incluyendo la entrada/salida, como puede ser una interfaz grafica, una petición HTTP o una conexión a base de datos. En el siguiente diagrama se puede ver la pirámide de test de Cohn:

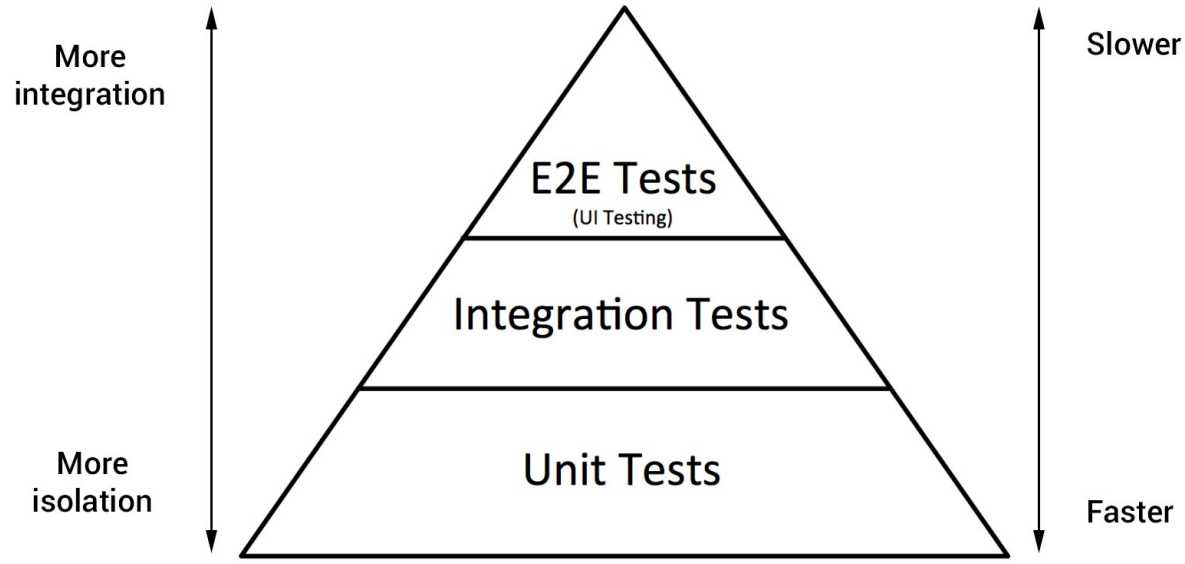

Figura 5: Pirámide de test de Cohn

1 La refactorización es una técnica para cambiar el código, mejorandolo, sin que cambie el comportamiento

Los test automáticos en este proyecto están planteados de varias maneras. Por un lado se encuentran los test de la API en los que están los test e2e y unitarios, ambos escritos con la librería php-unit. Por otro lado, se encuentran en el *front end* los test e2e de toda la aplicación, escritos con cypress, de esta manera comprobamos que todo el flujo del *software* funciona como debería, desde el *front end* hasta la API. Lo bueno de los test automáticos es que si existe un fallo es tan fácil como crear un test que simule ese fallo y así solucionarlo para que no ocurra más, porque si ocurriese, los test fallarían y avisarían que esa funcionalidad falla. Por último, aclarar que el tema de la pirámide de test y la cantidad de test de cada tipo es algo orientativo y que depende del dominio y las necesidades de cada proyecto. Lo importante es que es mejor tener algún test antes que no tener. Para desarrollar los test se utiliza el patrón AAA, Arrange Act Assert (preparación actuación afirmación). Esto sugiere hacer la estructura de cada test de forma que la preparación del test (el Arrange) está al principio, a continuación la acción (el Act) que se requiere para el test y por último lo que se espera de esa acción (el Assert), todo en ese orden.

Aquí se ve un ejemplo de test en el cual se verifica que la acción de "comprobar si el usuario tiene permisos de edición" funciona.

```php
public function test_return_true_if_user_can_edit()
{
    [$user, $user_group, $role] = $this->create_edit_user();
    $check_user_can_edit_action = new CheckUserCanEditAction();

    $response = $check_user_can_edit_action->execute($role);

    $this->assertEquals(true, $response);
}
```

Figura 6: Ejemplo de test

3.1.5. TDD

Este *software* intenta implementar TDD, Test Driven Development (desarrollo guiado por test). Es una metodología de desarrollo en la que se siguen unas reglas para implementar cada funcionalidad. Para ello se sigue un ciclo que se repite constantemente. La primera acción a realizar antes de nada es escribir el test, es el caso que queremos que haga la aplicación. De normal se escriben primero los test negativos (cuando un dato es *null*, es vació, fuera de rango, probar los limites entre números…) y poco a poco ir creando los demás test. Una vez creado el test, al ejecutar la suite de test dará error o fallo, se diferencia de que el **error** es porque hay un error en compilación por así decirlo (por ejemplo : no está creada la clase o el método correspondiente); en cambio, el **fallo** es que el código está correcto pero la funcionalidad no hace lo que se espera (por ejemplo: el método debería devolver un 3 y en realidad devuelve un 2). En resumen, se tendrá el test en rojo, es entonces cuando se hará

la funcionalidad mínima para que pase ese test, de tal forma que se implemente de la manera más rápida (aunque no sea la más eficiente). Hecho esto, al pasar el test, se encontrará en verde y se podrán realizar los *refactors necesarios*, tantos como se quiera, ya que el test nos asegura que el comportamiento no cambia. Terminada ya la optimización del código para ese test, se crea otro test para otra funcionalidad y se siguen los mimos pasos anteriormente citados.

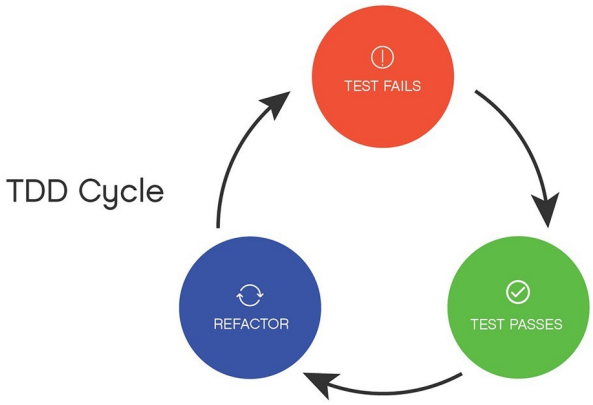

Figura 7: Ciclo de TDD

Las ventajas que nos ofrece realizar esto es que se sabe que todas las funcionalidades de la aplicación están testeadas y funcionan como es debido para todos los casos, esto da un gran valor añadido a la aplicación. Por otro lado, la desventaja de realizar TDD es el coste de tiempo que se invierte en crear el *software,* ya que hay que programar no solo el *software* en sí sino también los test, además de crear toda la estructura pertinente para poder ejecutarlos.

3.1.6. Docker y Docker Compose

Docker proporciona un entorno de desarrollo almacenado en contenedores sin necesidad de ninguna aplicación o dependencia. Por ejemplo si se requiere un entorno LAMP (Linux, Apache, MySQL y PHP) se puede tener un contenedor con estos programas y ejecutarlo en el ordenador local de desarrollo sin necesidad de instalar ninguno de estos programas. Esto facilita el desarrollo del *software* ya que agiliza la puesta en marcha. Si un nuevo integrante tuviera que desarrollar en el *software* solo tendría que instalar Docker y levantar el proyecto, sin preocuparse de qué programas tiene que instalar o cómo configurarlos.
En muchas ocasiones, a la hora de utilizar Docker para desarrollar, se necesita más de un contenedor o servicio para englobar la aplicación y simular lo más parecido posible el entorno de producción, donde puede haber un servidor para el *front end*, otro para el *back end* y otro para la base de datos, incluso podrían haber más, como por ejemplo, para un sistema de colas o un servidor de correo. Para esto existen los **orquestadores**, uno de ellos, y el que se usa en el proyecto, es Docker Compose, cuya función es administrar los diferentes servicios o contenedores de Docker que tenemos en la aplicación. En el caso de este

proyecto, existe un servicio para el *front end*, otro para Nginx, otro para la base de datos y otro para el PHP con el *back end*. De esta manera se pueden levantar todos los servicios a la vez de una forma rápida para desarrollar. Docker se compara mucho con máquinas virtuales, puesto que las dos cosas ofrecen un entorno para desarrollar. No obstante, Docker tiene varias ventajas respecto a estas. En primer lugar hay que entender cómo funcionan las máquinas virtual y Docker. Una máquina virtual tiene que instalar el sistema operativo, los drivers para la compatibilidad del sistema operativo con el hardware, hypervisor, librerias, etc. Sin embargo, Docker no instala la capa del sistema operativo ni el hypervisor, por lo que hace que sea mucho más ligero y rápido, aquí se ve un diagrama que lo ilustra muy bien.

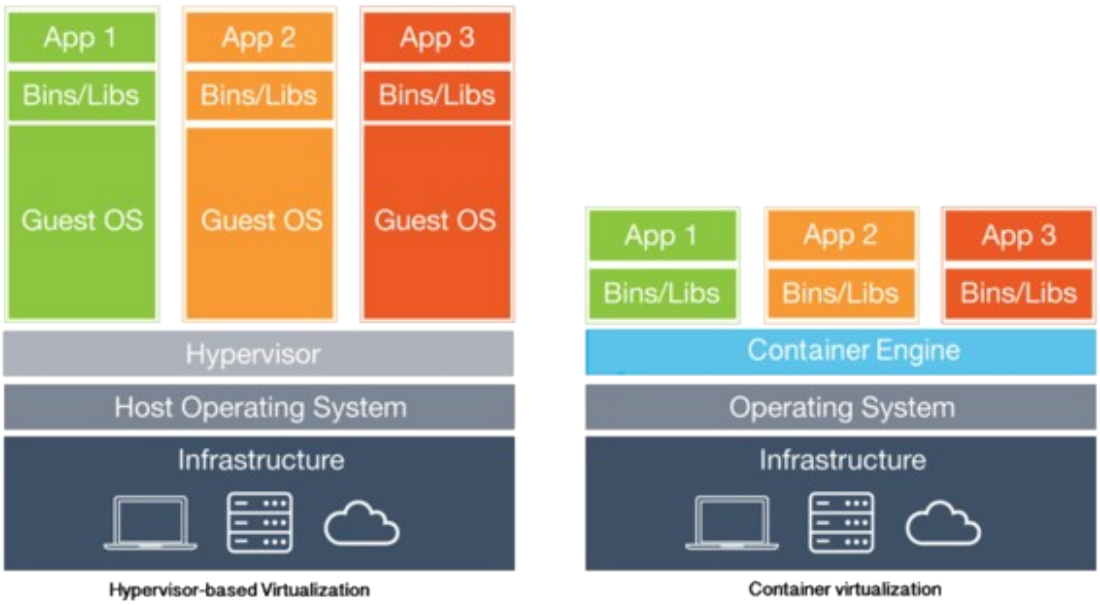

Figura 8: Máquina Virtual vs Docker

3.2. Desarollo del *front end*

3.2.1. Angular

Angular es un *framework* de JavaScript que utiliza TypeScript como lenguaje para tener más funcionalidades, entre ellas el tipado estático. Con Angular se realizan SPA, es decir, Single Page Application, que quiere decir que son aplicaciones de una sola página. Esto es así gracias a que realiza cargas de módulos, los cuales poseen componentes, pero solo existe una html que renderiza. Angular carga un módulo principal, el encargado de levantar toda la aplicación. Dentro de los módulos de Angular se pueden declarar componentes que son los encargados de renderizar el html y la lógica de este. Los módulos, además, tienen otra característica, y es que se pueden cargar de forma dinámica con carga perezosa, que sirve para que solo cargue los módulos que el usuarios utiliza. ¿Para qué cargar un módulo donde el usuario nunca entrará? Eso solo ralentizaría la carga inicial de la web. Por otro lado en

Angular están los servicios, que tienen básicamente dos funciones: la comunicación con la API mediante peticiones HTTP/HTTPS y el mantenimiento de los datos en memoria, para poder pasar datos entre componentes de forma sencilla y de este modo mantener una especie de caché en memoria (los datos se pierden una vez se recarga la página, ya que los servicios no tienen un ciclo de vida). La última cosa importante que explicar de Angular antes de ver la estructura del proyecto es el ciclo de vida que tienen los componentes. Angular tiene un potente ciclo de vida en los componentes que hace que se puedan realizar distintas acciones dependiendo de en qué momento queramos que se ejecuten.

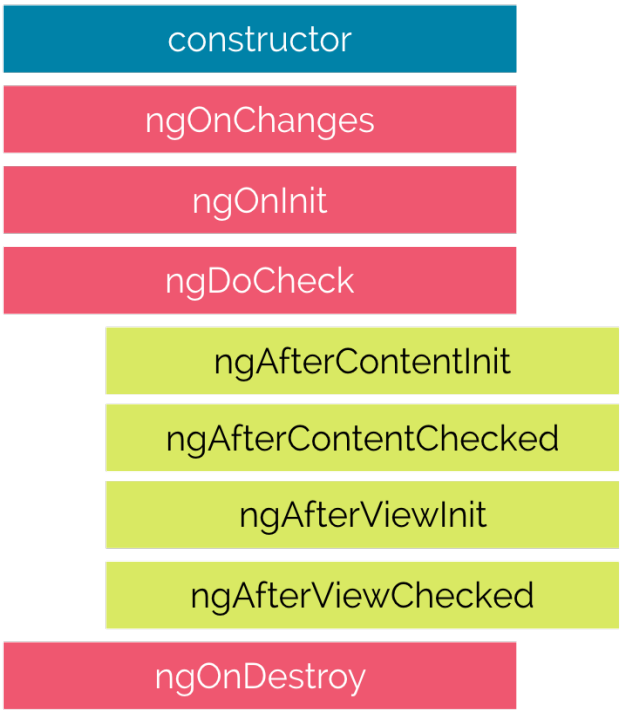

Figura 9: Ciclo de vida de Angular

Primeramente se ejecuta el constructor del componente, seguidamente el *on change* y a continuación en *on init*. En la primera carga del componente se ejecuta después el *do check* que comprobará cambios en la vista y después de esto se ejecutan en bucle los siguientes mientras haya un cambio: *after content init*, *after content checked*, *after view init*, *after view checked*, en este orden. Al ejecutarse el *do check,* si hay cambios en la vista o en el controlador, se ejecutará también el *on change*. Si por el contrario, no hay cambios no se ejecuta ninguno de los ciclos de vida. Por último, cuando el componente deja de existir se ejecuta el *on destroy*.

Una vez explicado cómo funciona Angular por encima hay que comentar el *scaffolding* de la aplicación Angular, es decir, la estructura de carpetas, y el por qué se ha realizado así.

Por un lado, existe la carpeta “cypress” donde se encuentran los test e2e de la aplicación, y a la misma altura esta “src”, donde se encuentra todo lo necesario para el funcionamiento de la aplicación. Dentro de esta carpeta está “app” donde se encuentra el código. Dentro de “app” se puede ver la carpeta “pages”, donde están los módulos de las diferentes páginas (home, añadir documento, login…). A la misma altura está “layout”, en la que se encuentran las diferentes distribuciones de las páginas (header, footer, sidenav). En la carpeta “services” están los diferentes servicios de la aplicación, tanto los que realizan llamadas a la API como los que mantienen datos en memoria a modo de caché.

3.2.2. Diseño

Para el diseño de la web se utiliza Angular Material. Es un *framework* CSS como lo son por ejemplo Bootstrap, Materialize, Foundation, Pure, Bulma o Tailwind. A diferencia de estos, Angular Material está desarrollado por el mismo equipo que Angular (Google), y por tanto está completamente integrado con Angular y se hace muy sencillo su uso. Además, ofrece componentes ya diseñados con estilo "material". Este estilo está caracterizado por un diseño, con colores planos, que da sensación de minimalismo y elegancia. A parte de utilizar este *framework* de diseño hay modificaciones utilizando Sass.

Sass es un lenguaje de hojas de estilo que al compilar se traducen a CSS. Este lenguaje ayuda al crear variables, condiciones o mixins, cosa que le da mucha potencia al diseño web. Sass tienes dos extensiones, la primera ".sass" es la más antigua y tiene detalles de azúcar sintáctico como que no usa punto y coma ni llaves, y la otra extensión es ".scss", más moderna cuya sintaxis es igual que en CSS, con punto y coma y llaves, lo que permite que las hojas de estilo hechas con CSS sean compatibles con esta extensión. El diseño es *responsive* para que se pueda acceder tanto desde ordenador como en tablet o móvil, pero por el tipo de aplicación web que es, tiene más sentido usarla desde una pantalla de ordenador, puesto que se hace más cómoda la administración de documentos. Aun así, se diseña siguiendo la filosofía de ***mobile first***. Esta se basa en realizar primero el diseño para tamaños pequeños, luego se continua pensando en el tamaño en tablet y por último en tamaño de una pantalla de ordenador. Se puede hacer de varias formas pero se elige la de ir sobrescribiendo solo las clases CSS que se necesiten modificar, por tanto los estilos para tablet y ordenador ocupan mucho menos, ya que la mayoría se encuentra en la parte móvil. Para conseguir esto se realizan ***media queries*** que afecten solo a partir de un tamaño, es decir, la *media query* de móvil no se elimina una vez pasa a tablet ni la de tablet una vez pasa a ordenador. Se realiza así porque la gran mayoría de clases son exactamente iguales en un tamaño que en otro y por eso solo se modifica lo poco que tiene que cambiar.

El tamaño de móvil es cuando la pantalla tiene menos de 600 píxeles, no se pone *media query* porque van a ser las clases base que se usan en todos los tamaños; la tablet es a partir de 600 píxeles hacia adelante (no hay limite superior para poder aprovechar las clases sobrescritas en el tamaño de ordenador); y por último la pantalla de ordenador es a partir de 991 píxeles.

En cuanto a la metodología a la hora de escribir el CSS se usa **BEM**, Block Element Modifier (Bloque Elemento Modificador). BEM facilita el mantenimiento y la legibilidad del código CSS haciendo que sea modular, reutilizable y estructurado. BEM consiste en diferenciar los componentes de la página e ir seleccionando el qué sería un bloque, cuáles son los elementos que hay en ese bloque y qué modificadores puede tener. Un vez hecha la distinción, a la hora de escribir el diseño para el bloque, se crea una clase con el nombre del bloque (por ejemplo: **.menu**). Para escribir el diseño del elemento del bloque se escribe el

nombre del bloque seguido de dos guiones bajos y el nombre del elemento (por ejemplo: **.menu__logo**) y, por último, para diseñar los diferentes estados de ese elemento se sigue con dos guiones y el modificador que se requiera (por ejemplo: **.menu__logo--small**). Si el nombre del bloque del elemento o del modificador consta de más de una palabra se une mediante guiones. Hay que recalcar que el modificador se puede asignar al bloque (por ejemplo: **.menu—dark**), y que un elemento dentro de un bloque se puede convertir en bloque independiente que tenga elementos, pero no se puede referencia en una misma clase varios elementos de un bloque.

3.3. Desarrollo del back end

El *back end* sirve a modo de API para el *front end*. Está creado con Laravel, un *framework* de PHP que facilita el desarrollo, puesto que tiene integrado ya un sistema de enrutamiento, un ORM, llamado Eloquent, y ciertas funcionalidades que hace que desarrollar una API sea muy rápido. Además, tiene mucha potencia comparado con otros *frameworks* como Symfony, CodeIgniter Yii, CakePHP o Zend. En este desarrollo no se hace uso de su sistema de renderizado de vistas con Blade, puesto que al funcionar como API solo enviará objetos en formato JSON.

Junto a Laravel se hace uso de varias librerías: Passport, php-unit, phpcpd, phpcs y phpcbf. **Passport** viene integrada con Laravel aunque hay que instalarla con el gestor de paquetes por defecto de PHP (Composer), y sirve para la autenticación con *tokens*, creándolos y verificando el acceso a la aplicación. **Php-unit** nos proporciona herramientas para crear test automáticos, ya sean test unitarios o e2e. **Phpcpd** detecta código duplicado especificando cuántas lineas duplicadas se quieren detectar. **Phpcs** detecta errores de código estandarizado, por ejemplo si la linea está mal tabulada, si se usan comillas dobles o simples, o cualquier regla de sintaxis que se quiera introducir; ya hay varios estándares de código creado por asociaciones y grupos de desarrollo como por ejemplo el PSR-1 o PSR-2. Por último, **phpcbf** soluciona los fallos más fáciles detectados por la librería phpcs. Estas tres últimas librerías se utilizan para mejorar la calidad del código.

El *scaffolding* del *back end* está orientado a arquitectura hexagonal y DDD, explicados anteriormente, juntando partes de la estructura de carpetas que da Laravel.

Explicado todo lo importante sobre los conceptos y las tecnologías usadas, se procede a explicar el desarrollo y todos sus pasos.

3.4. Explicación del desarrollo

Antes de empezar a desarrollar hay que hacer un estudio previo para averiguar varias cosas. Una es cómo será nuestro dominio, qué lógica tendrá, qué entidades constituirán el *software*… La otra cosa importante a meditar es cómo sera el flujo del usuario, que sea

coherente, y que la experiencia de usuario sea intuitiva y agradable. Cada uno de estas cosas se soluciona de diversas formas. En el primer caso se diseña un modelo UML con las entidades y cómo se relacionan.

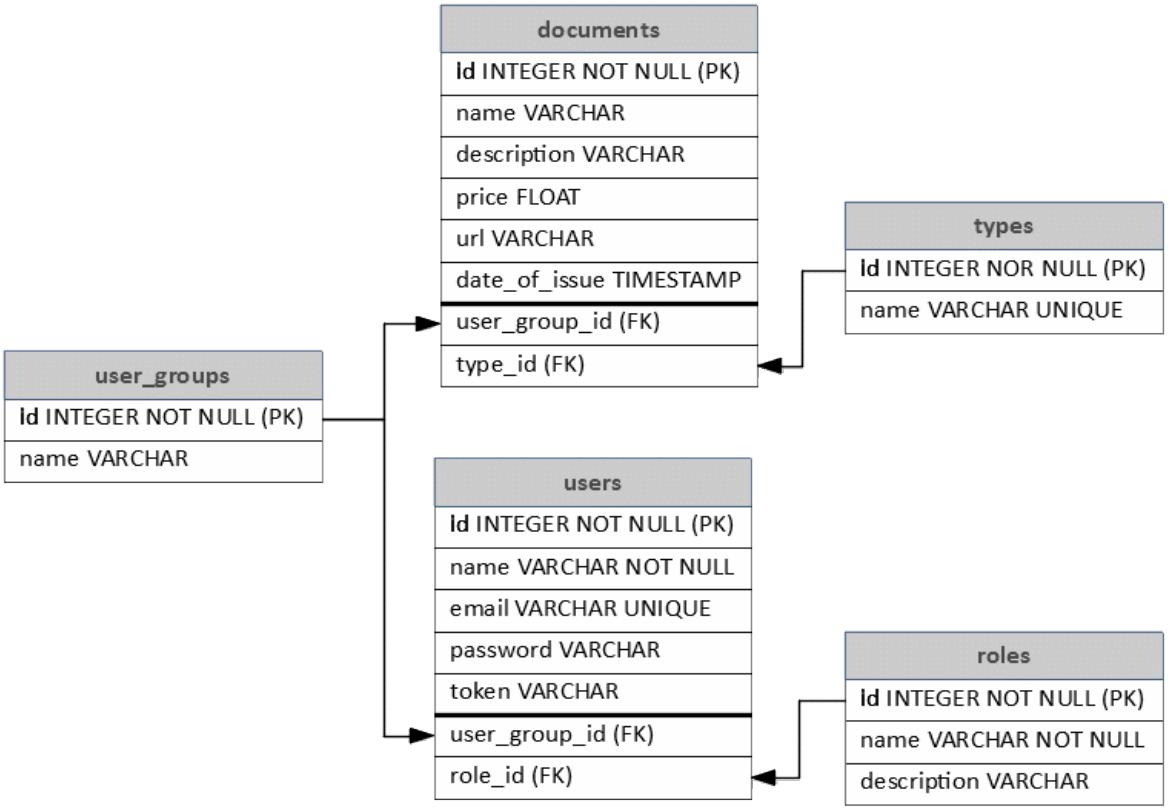

Figura 10: Diagrama UML de la estructura de base de datos

En cuanto a la experiencia de usuario se realizan diferentes *wireframes* para tener las ideas claras de dónde situar cada componente, de tal forma que antes de invertir el tiempo en el desarrollo se sepa con seguridad cuál es el funcionamiento que se requiere. Para hacerlo se ha utilizado la página web https://mockflow.com/.

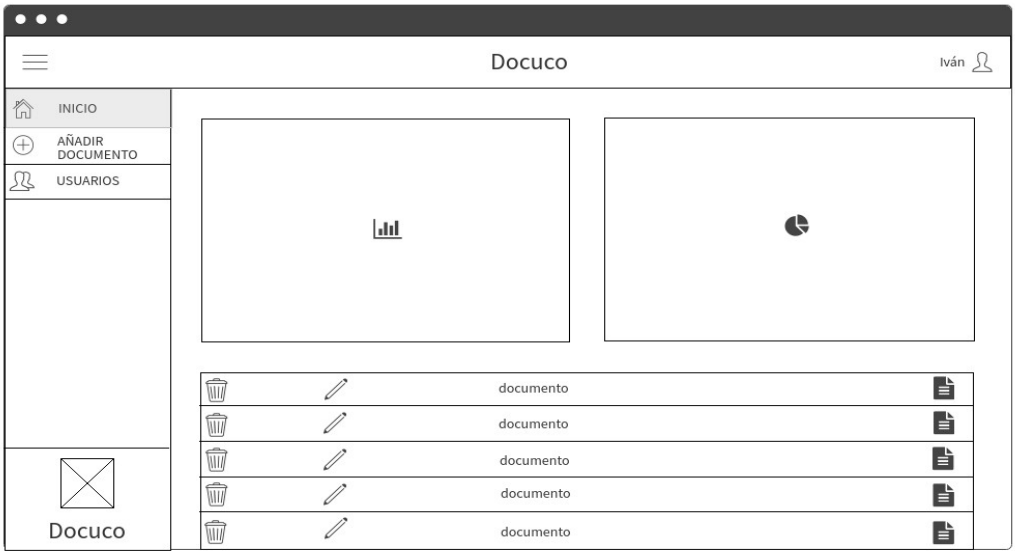

Figura 11: wireframe del dashboard

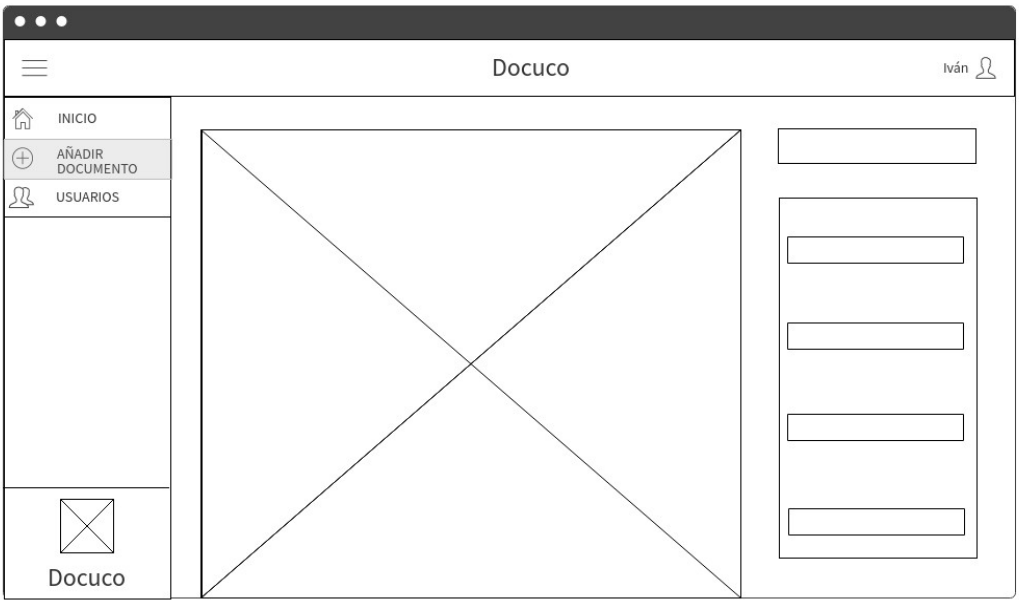

Figura 12: wireframe de añadir documento

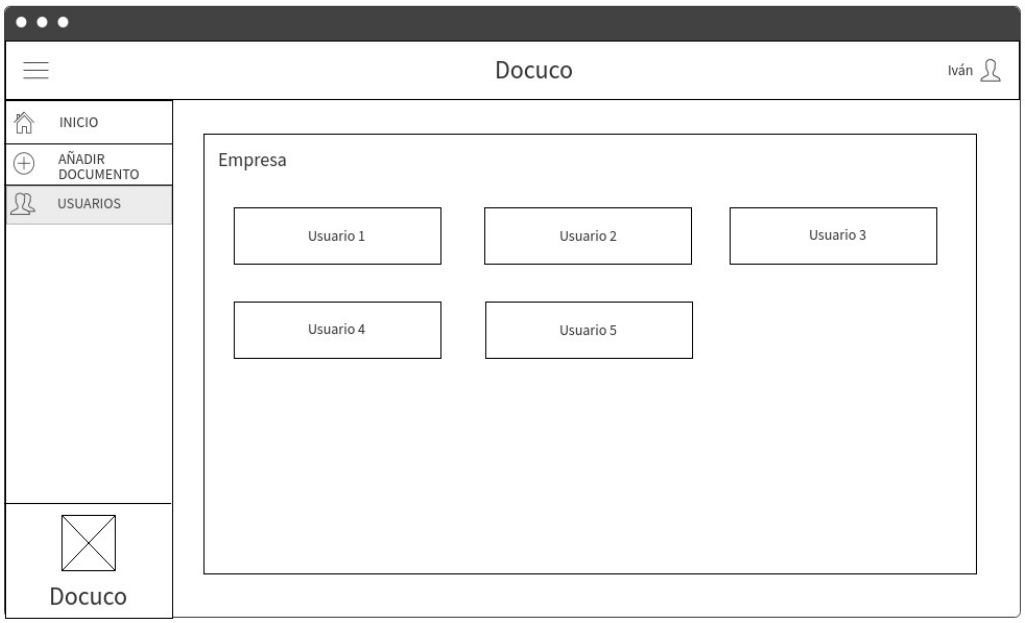

Figura 13: wireframe de administración de usuarios

Una vez hechos el diagrama de UML y los *wireframes* del *software* lo mejor es crear las diferentes **historias de usuario**. Consta de especificar las acciones que se pueden hacer con qué tipo de usuario y qué es lo que se espera al realizarlas (por ejemplo: "Como usuario administrador puedo editar un usuario"). Para ello se hace un *brainstorm* con todas las ideas y se apuntan en un papel, indicando a qué módulo del dominio pertenece. Las acciones que finalmente se ven claras en el *software* son:

- Usuarios
  - Cualquier usuario registrado puede iniciar sesión
  - Cualquier usuario puede cerrar sesión
  - Un usuario con rol administrador puede añadir más usuarios a su grupo
  - Un usuario con rol administrador puede ver los usuarios de su grupo
  - Un usuario con rol administrador puede modificar un usuario de su grupo
  - Un usuario con rol administrador puede eliminar un usuario de su grupo

- Documentos
  - Un usuario con rol administrador o editor puede añadir un documento a su grupo
  - Un usuario con rol administrador o editor puede modificar un documento de su grupo
  - Un usuario con rol administrador o editor puede eliminar un documento de su grupo
  - Cualquier usuario puede ver todos los documentos del grupo
  - Cualquier usuario puede ver datos para las graficas del grupo

Hecho esto, se plantea el nombre del proyecto. Para ello se escriben palabras relacionadas con el *software*, como documentos, ingresos, gastos, ficheros, análisis, administración, facturas, datos, colección… Con esas palabras como referencia hay que encontrar un nombre que simbolice el funcionamiento de la herramienta, que sea fácil de recordar. Después de varias propuestas se elige el nombre **Docuco,** enlazando las palabras documentos y colección.

Con todas estas ideas bajadas al papel se puede empezar a desarrollar. La idea de este desarrollo es que se pueden hacer todas las cosas en paralelo, de tal forma que se pueda avanzar sin dependencias entre las distintas partes del proyecto. Para hacer la API sin necesidad del *front end* se usa Insomnia, una aplicación que simula las peticiones que realizará el *front end*, similar a Postman. Para que el *front end* no dependa del *back end* se *mockea* la respuesta que realiza la API en formato JSON. En cuanto al diseño, una vez hechos los *wireframes* solo quedaría mejorar la estética de los componentes utilizados de Angular Material.

Para la creación de la base de datos, desde el diagrama UML, se utilizan las migraciones de Laravel. Cada migración es un fichero que internamente lo que hace es conectarse a la base de datos y hacer lo que ponga en el fichero, como puede ser: crear tablas, modificarlas, eliminarlas… Esto facilita mucho el trabajo a la hora de hacer un desarrollo **iterativo incremental**, ya que lo más probable es que al principio no se tenga toda la estructura que se va a usar y conforme la aplicación crezca se pueden ir añadiendo tablas sin perder la consistencia. Se puede hacer que solo se ejecuten las migraciones que falten porque se hayan añadido más tarde. Acompañando a esto están los *seeds*, datos de prueba que se añaden a la base de datos con los cuales podemos comprobar el funcionamiento.

Después de realizar estos pasos previos empezamos con lo primero de todo de la aplicación, el sistema de login.

3.4.1. Login

En primer lugar, el *front end*, a través de un formulario, realiza una petición HTTPS con el método POST, donde en el cuerpo de la petición se encuentran las credenciales con las que se quiere acceder a la aplicación. El *back end* recibe esa petición, se comprueba en base de datos que existe el usuario y, si este existe, se crea un *token* personalizado con los datos del usuario como son el id, el rol al que pertenece y el grupo de usuarios en el que se encuentra. Una vez creado el *token* se devuelve en la petición, y si no existe el usuario se envía un error 401, indicando que no está autorizado. El *front end*, tras recibir el *token*, lo guarda en *session storage* (el *session storage* permite almacenar datos mientras esté la pestaña, y en el momento que se elimina la pestaña del navegador se eliminan los datos del *session storage*). Se almacena el *token* para poder realizar todas las demás peticiones con él en la cabecera de la petición, puesto que todas las rutas de la API están aseguradas con un *middleware* que

comprueba si la petición viene con un *token* y, además, si el *token* que recibe es correcto; de no ser así envía un error 401. Con esto, está la API asegurada, pero hay que realizar algo en el *front end* para que el usuario no ponga la url del dashboard, por ejemplo, y se le muestre la pantalla. Para ello, en Angular, existen los **Guards**, funciones que se ejecutan al acceder a una ruta, una especie de *middleware* que permite comprobar cosas como si el usuario ha iniciado sesión o si tiene cierto rol, y ejecutar acciones al respecto.

3.4.2. Cerrar sesión

Para cerrar sesión se elimina el *token* del *session storage* y se redirige a la pantalla de login. De esa forma, al no tener el *token*, las peticiones a la API solo devuelven 401 y los Guards de Angular no permiten entrar a las diferentes página.

3.4.3. Dashboard

Esta pantalla es la principal una vez se ha accedido a la plataforma. En ella se muestran dos gráficas y un listado. Las gráficas se realizan en el *front end* mediante una librería llamada Chart.js, a la cual se le pasa un objeto con los datos y unas opciones de visualización. El objeto de datos lo trae directamente la API mediante una petición, ya que así resulta más rápido que si tuviera que hacer los cálculos el lado del *front end*. Al hacerlo así se podría pensar que estamos acoplando la *back end* con una librería del *front end*, pero la verdad es que el dato que manda la API es un estándar en cuanto a gráficas, por tanto estamos dando datos estandarizados sin que se acople a qué librería se usa. La primera gráfica es dinámica y puede variar lo que se quiere ver, por un lado se ven los ingresos totales y los gastos totales de cada mes en el año actual, y por otro lado se puede ver el beneficio de cada mes en el año actual. La segunda gráfica muestra de una forma más visual qué porcentaje del total corresponde a ingresos y cuál a gastos. Por último, la lista de documentos permite ver los datos relevantes del documento, además de realizar ciertas acciones sobre cada uno, como por ejemplo editar el documento, eliminarlo o ver el PDF del documento para poder descargarlo. A esta pantalla puede acceder cualquier usuario que tenga acceso a la aplicación pero se le restringe las acciones que pueda hacer dependiendo del rol. El administrador tiene acceso completo a todo, el editor puede ver los documentos, editarlos y borrarlos, por último el rol de visualizador solo puede ver las gráficas y los documentos pero no puede editar los documentos ni borrarlos, por tanto los botones para editar o borrar no le aparecerán a este rol de usuario.

3.4.4. Añadir documento

En esta pantalla se pueden añadir documentos clicando en la zona central de la página o arrastrando una imagen a esta zona. Para esta funcionalidad no se hace uso de ninguna librería sino que se implementa a través de una directiva de Angular la cual detecta cuando se hace la acción de arrastrar y soltar (*drag and drop*) y poder cargar el fichero en memoria

para utilizarlo. Una vez el fichero está en memoria se visualiza, gracias a la librería JavaScript llamada ng2-pdf-viewer, en el lado izquierdo de la pantalla, y en la derecha se muestra un formulario. La posición de dónde mostrar cada cosa no es casual, y es que a través de un pequeño estudio se analiza que es la mejor posición, puesto que el formulario se rellena con los datos del PDF y normalmente cuando queremos copiar algo de un sitio a otro, la gente diestra se pone la referencia en la izquierda para no entorpecerse con el brazo, por eso los diestros (la gran mayoría de la población) prefiere esta posición. El formulario pide unos datos obligatorios como son la fecha del documento, el precio, el nombre o indicar que tipo de documento es, si ingreso o gasto.

Completado el formulario se envía mediante HTTPS a la API. La API comprueba mediante el *middleware* si el usuario ha iniciado sesión y si tiene permisos para realizar esa acción; si no ha iniciado sesión se envía un 401 como se ha explicado en la pantalla del login, pero si el caso es que no tiene permiso, se envía un código 423 indicado de esta manera que no tiene permiso para realizar la acción. Sin embargo, si todo es correcto, se subirá el documento a la carpeta pública de Laravel con un nombre único y se asocia el archivo PDF subido con los datos del formulario. Posteriormente se crea en la base de datos y se asocia al grupo de usuarios del usuario que envió la petición (esto se sabe mediante la información del *token* en la cabecera de la petición). Hecho esto se devuelve el documento creado como respuesta a la petición.

3.4.5. Actualizar documento

Para actualizar un documento hay que ir a la pantalla principal y en la lista de documentos se clica el icono de lápiz para editarlo. Al editar el documento se reutilizan en Angular los mismos componentes que al crear un documento, pero cambiándolos de orden, primero se muestra el formulario y a la derecha el PDF. Con esto conseguimos que el usuario no crea que está añadiendo, sino que, como la pantalla es diferente, la acción que realiza también lo es. El funcionamiento es el mismo que añadir, se manda el *token* en la cabecera de la petición y los datos en el body, la diferencia es que esta vez se manda en una petición PUT con el id como ***path param*** de la url. El *back end* comprueba si el usuario tiene permisos y si es así realiza la acción, si no devuelve un error con el código 423.

3.4.6. Eliminar documento

Para eliminar un documento concreto hay que ir a la pantalla principal y en la lista de documentos clicar el icono de papelera. Se verifica con un cartel si está seguro que se quiere eliminar el documento. El funcionamiento es el mismo que en las demás acciones respecto al *token*, la verificación en la API del rol de usuario y la respuesta si no tiene permiso. La diferencia está en que el método HTTP es DELETE.

3.4.7. Gestión de usuarios

La gestión de usuarios es exactamente igual que con documentos, todas las acciones típicas de un CRUD, Create Read Update Delete (Crear Leer Actualizar Eliminar). En Angular todo esto se realiza en un módulo, el cual se divide en varios componentes. Este módulo, al igual que los demás módulos, se carga de forma perezosa cuando el usuario demanda ver esa parte de la aplicación.

3.4.8. Logo

Puesto que el diseño de la aplicación web es minimalista, sencilla e intuitiva, el logo tenía que demostrar el mismo estilo, siendo un símbolo del *software* fácil de reconocer y que se asociara rápidamente. Para ello, los colores siguen el mismo esquema que en la aplicación, siendo mayoritariamente monocromáticos en azul, aunque en la aplicación existe algo de rosa para destacar ciertas cosas en concreto.

Una vez elegido el color, se piensa en la forma que tendrá. Siendo un símbolo referente de la aplicación, el logo tiene que expresar de forma sencilla lo que se realiza en ella. Como es una gestión de documentos los iconos que vienen a la mente son iconos de ficheros o de carpetas. Se elige la segunda opción ya que de esta forma se señala que en una carpeta hay un conjunto de ficheros como ocurre en la aplicación, y se ve mejor ese símil, ganando de esta forma coherencia con el funcionamiento y el diseño del *software*. Para darle ese toque que se caracterice con la aplicación se hace una mezcla entre la **D** de **Docuco** y el icono de carpeta, haciendo una línea redondeada en la base de este y girando el icono, dando esa forma llamativa.

El diseño esta hecho de forma vectorial con Inkscape para no perder calidad en reducciones o ampliaciones de tamaño, y con pocos detalles pequeños para que se distinga bien en la web aunque se use de forma reducida como en el favicon, por ejemplo.

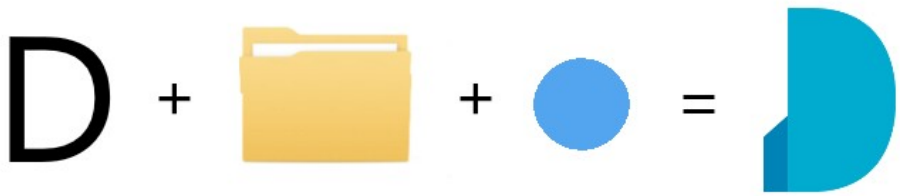

Figura 14: Explicación del logo

3.5. Explicación del despliegue

De momento con todo lo explicado anteriormente se ha conseguido crear un servidor local con Docker y Docker Compose, se ha desarrollado la plataforma y se ha creado todo lo

necesario para su funcionamiento: base de datos, *back end*, *front end*, diseño, etc. Pero con esto solo se puede ejecutar el programa en local, solo el usuario del ordenador donde se esté ejecutando el *software* tiene acceso. Para solucionar esto hace falta desplegar la aplicación en un servidor que esté las 24 horas del día encendido para que cualquier persona pueda acceder a cualquier hora. Entre todas las opciones de servidores en la nube, OVH, Linode, IONOS, etc. hay tres que destacan: AWS, Amazon Web Service, GCP, Google Cloud Platform y Azure de Microsoft. Entre ellas se elige finalmente los servicios de AWS. Se hace uso de varios servicios que tiene que se explican a continuación.

3.5.1. EC2 y relación con el proyecto

EC2, Elastic Compute Cloud (nube de servidores elásticos), es un servicio de AWS el cual básicamente proporciona servidores virtuales con posibilidad de escalar su capacidad de forma sencilla sin perder los datos. Una vez creado el servidor se le asigna una clave SSH para que se pueda tener acceso.

El siguiente paso es configurar NGINX para que cuando se despliegue la aplicación se tenga acceso. Se configuran dentro del mismo servidor dos sitios virtuales, uno para el *front end* y otro para el *back end*, de tal forma que en el mismo servidor se alojan varios proyectos con nombres de subdominio diferentes. En este caso el subdominio *docuco.icordoba.es* está asociado al *front end* y el subdominio *api.docuco.icordoba.es* al *back end*. Al alojar en el mismo servidor el *back end* y el *front end,* hay que configurar en el sitio virtual del *back end* la configuración del **CORS** para que el *front end* pueda acceder al *back end.* Si se hubiera hecho en servidores diferentes esto no haría falta. Hecho esto queda hacer el despliegue.

Para desplegar el *front end,* se compila mediante el comando de angular *ng build --prod*, lo cual genera una carpeta llamada *build* que contiene el index.html y los JavaScript necesarios. Estos ficheros tal cual se dejan en la carpeta del servidor correspondiente que se haya especificado en el sitio virtual de NGINX.

En cuanto al *back end* es aún más sencillo porque no hay que compilar nada, se pueden coger todas las carpetas y ficheros tal cual y ponerlas dentro de la carpeta, especificada en su sitio virtual de NGINX. Sin embargo, para que no tarde tanto la transferencia de archivos, es preferible no pasar la carpeta *vendor*, ya que es la que contiene todas las dependencias de librerías que se usan por debajo, y una vez pasado lo demás al servidor se ejecuta el comando *composer install,* que lee del fichero *composer.json* las dependencias y las instala. Otra cosa a tener en cuenta en la parte del *back end* es el fichero *.env,* que hay que crearlo con la configuración correspondiente.

3.5.2. RDS y relación con el proyecto

Para la base de datos se usa el servicio de AWS llamado RDS, Relational Database Service (servicio de base de datos relacionales). Este servicio está enfocado a la creación y gestión de base de datos. Está la posibilidad de crear una base de datos con MySQL, MariaDB o cualquier gestor de base de datos que sea relacional, en este caso se ha desarrollado el *software* con PostgreSQL y es el que se elige.

Una vez creada la base de datos con las credenciales que se hayan puesto hay que volver al *back end* que se ha desplegado para poder crear las tablas desde ahí con el comando de artisan *php artisan migrate:fresh* y de esta forma cogerá las credenciales de la base de datos que se hayan puesto en el fichero *.env,* y creará las tablas que estén en las migraciones de Laravel.

3.5.3. Route53 y relación con el proyecto

Con todo esto ya se tiene el *software* desplegado y la base de datos montada, pero solo se puede acceder a él mediante la IP. Esto ocurre porque no se ha configurado ninguna ruta desde el dominio que se usa hacia ese servidor en concreto. Para ello, se usa el servicio Route53 de AWS, que vincula los dominios o subdominios con servidores mediante las Ips. El problema es que los servidores creados, por defecto, tienen una IP dinámica; para cambiar esto se tiene que crear lo que AWS llama una Elastic IP y vincularla con el servidor. Esta Elastic IP es una IP fija que se asocia a la máquina y, aunque esta se reinicie, cambiará su IP pero no su Elastic IP vinculada, así que en el servicio Route53 se asocia el subdominio con la Elastic IP que corresponda.

De esta manera cuando en la url se ponga el nombre del subdominio se accederá al servidor, que este, a su vez, internamente redirigirá la ruta entre los sitios virtuales de NGINX.

4. Evaluación

Actualmente este proyecto se puede visitar en https://docuco.icordoba.es (las credenciales están como *placeholder*). El código completo, como se ha mencionado anteriormente, se encuentra en github, https://github.com/nabby27/docuco.

Los objetivos finales planteados se han cumplido exitosamente, realizando todas las tareas necesarias. Estos objetivos han variado desde el planteamiento de la idea. Al principio se pensó en implementar un OCR, en concreto Tesseract (https://tesseract.projectnaptha.com/) en el lado cliente, para que directamente este reconocimiento de caracteres leyera el documento PDF e introdujera automáticamente los datos necesarios que se piden en el formulario, gracias a haber guardado las coordenadas de donde se encuentra cada dato previamente. Finalmente esta idea no se introdujo en los objetivos finales puesto que al

empezar el planteamiento y crear los *wireframes* se vio que podían darse muchas casuísticas que no se podían tener en cuenta, como por ejemplo que el mismo tipo de PDF tuviera más páginas o simplemente que una linea (como la de precio total) fuera más larga en un documento que en otro, y por ese motivo que los datos estuvieran en otras coordenadas del documento. De todas maneras es una idea de mejora que queda pendiente mirar cómo implementar de la mejor manera.

Por otro lado, se han añadido otros objetivos conforme avanzaba el proyecto, como son el despliegue en los servidores de AWS o la implementación de test automáticos.

Una vez acabado el proyecto hay algunas mejoras o implementaciones nuevas, tanto técnicamente como de funcionalidades, que se pueden llevar a cabo.

Técnicamente, en cuanto al *front end*, se puede implementar **Redux** con ng-redux (https://github.com/angular-redux/ng-redux) para el manejo de estados en Angular, facilitando el tema de mantener datos vivos y llamar a la API para recuperarlos, manteniendo la consistencia entre ellos y así poder gestionarlos de una mejor manera. También se podría añadir la internacionalización que posee Angular con ngx-translate (https://github.com/ngx-translate/core) en su core, con los cambios necesarios en el diseño, para poder visualizar la plataforma en diferentes idiomas. Por el lado del *back end*, puede evolucionar la estructura hacia **CQRS**, Command Query Responsability Segregation (segregación de responsabilidad en comando y consulta), que básicamente es la misma estructura que existe, añadiendo una capa más entre el controlador y el caso de uso, en la cual se encuentra el command/query que se lanzan a un bus, que puede ser sincrono o asíncrono, y el handler que ejecuta este command/query. Con este cambio a CQRS se cambiarían acciones como la de crear, la cual se haría mediante el método HTTP PUT y pasándole el id desde el cliente con un formato **UUID**, sin depender de la base de datos (infraestructura) para que los autogenere. Esto, además, facilita el testing y se quitaría deuda técnica. Otra mejora en la parte del *back end* es la implementación de **Swagger** (https://swagger.io/) para documentar los *endpoints*, sus parámetros y sus posibles respuestas. Por último, una mejora general en toda la aplicación sería la implementación de **CI/CD**, continuos integration/continuos deployment (integración continua/despliegue continuo), con lo que se consigue lo que se llama *continuos delivery* (entrega continua). Esto se podría realizar mediante las GitHub Actions comprobando antes de desplegar que todo esté correcto, como son los test, el estándar de código y el linter en cada commit que se haga a la rama master del repositorio. Con esto se tendría siempre la última versión en el servidor a cada cambio que se haga de forma automática.

Entre las funcionalidades que se pueden añadir para mejorar el *software* estaría el multiidioma, como se ha explicado en la mejora técnica. Tener cada usuario unas preferencias de la aplicación (modo oscuro, preferencia en el tema de colores, graficas preferidas, idioma por defecto, etc.), y poder realizar acciones mediante las diferentes

gráficas, como son moverse en el tiempo de la gráfica o poder filtrar la lista de documentos desde ahí. Otra mejora puede ser el registro de usuarios desde la pantalla principal, pero se tendría que mirar cómo se implementaría con el plan de precios añadiendo una pasarela de pago y que caduque el acceso a la plataforma. Con esto también habría que ver cómo manejar las contraseñas de usuario cuando se crean, ya que actualmente siempre se pone la misma contraseña 123456 para el funcionamiento de la plataforma como prueba, puesto que el objetivo no era en ningún momento el tema de usuarios y su gestión ni los roles, son solo añadidos para este proyecto.

5. Conclusión

Considero que el proyecto es muy completo, proporcionando buenas prácticas en cada parte del desarrollo e integrando todos los módulos necesarios para cumplir los objetivos, haciendo que sea estable gracias a los test, mantenible y escalable gracias a una buena estructura base y los principios usados. No solo se ha desarrollado un *software* para gestión documental, sino que también se ha puesto en producción siendo totalmente funcional creando una marca de *software* con nombre propio, logo y estilo.

6. Referencias

Front end

- https://angular.io/docs
- https://www.typescriptlang.org/docs/home.html
- https://material.angular.io/components/categories
- https://docs.cypress.io/guides/overview/why-cypress.html
- https://www.chartjs.org/docs/latest/
- https://momentjs.com/docs/
- https://github.com/VadimDez/ng2-pdf-viewer

Back end

- https://laravel.com/docs/7.x
- https://phpunit.readthedocs.io/en/9.1/
- https://github.com/sebastianbergmann/phpcpd
- https://github.com/squizlabs/PHP_CodeSniffer

SOLID

- https://profile.es/blog/principios-solid-desarrollo-software-calidad/
- https://es.wikipedia.org/wiki/SOLID
- https://enmilocalfunciona.io/principios-solid/

Arquitectura Hexagonal

- https://www.franciscougalde.com/2019/09/17/introduccion-a-la-arquitectura-hexagonal/
- https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f
- https://codely.tv/blog/screencasts/arquitectura-hexagonal-ddd/
- https://apiumhub.com/es/tech-blog-barcelona/arquitectura-hexagonal/
- https://our-academy.org/posts/arquitectura-hexagonal

DDD

- https://es.wikipedia.org/wiki/Dise%C3%B1o_guiado_por_el_dominio
- https://justdigital.agency/que-es-domain-driven-design-ddd/
- https://devexperto.com/domain-driven-design-1/
- https://github.com/jatubio/5minutos_laravel/wiki/Resumen-sobre-DDD.-Domain-Driven-Design
- https://medium.com/@jonathanloscalzo/domain-driven-design-principios-beneficios-y-elementos-primera-parte-aad90f30aa35
- https://franiglesias.github.io/ddd-is-not-what-they-said/
- http://www.icorp.com.mx/blog/domain-driven-design/

Testing

- https://es.wikipedia.org/wiki/Pruebas_de_software
- https://es.wikipedia.org/wiki/Prueba_unitaria
- https://www.yeeply.com/blog/que-son-pruebas-unitarias/
- https://apiumhub.com/es/tech-blog-barcelona/beneficios-de-las-pruebas-unitarias/
- https://www.etnassoft.com/2011/07/27/tests-unitarios-cuando-usarlos-y-pistas-para-conseguir-un-sistema-robusto/
- https://www.mtp.es/pruebas-e2e-la-forma-de-garantizar-que-un-negocio-esta-asegurado/
- https://blog.irontec.com/introduccion-automatizacion-tests-e2e-cypress-io/

TDD

- https://es.wikipedia.org/wiki/Desarrollo_guiado_por_pruebas
- https://www.paradigmadigital.com/dev/tdd-como-metodologia-de-diseno-de-software/

Otros

- https://aws.amazon.com/
- https://docs.docker.com/

- https://docs.docker.com/compose/
- https://pro.codely.tv/home/
- https://openwebinars.net/