GRADO EN INGENIERÍA INFORMÁTICA

TRABAJO DE FINAL DE GRADO

Gestor de Contraseñas

Autor:
Adelin Octavian DUBANOAIA

Supervisado por:
Bogdan Costel MOCANU

Tutorizado por:
Cristina CAMPOS SANCHO

13 de septiembre de 2023

Resumen

Este documento describe la creación y el desarrollo de un gestor de contraseñas por parte de un estudiante con ganas de aprender nuevas tecnologías y adquirir nuevos conocimientos. A lo largo del documento se detallan las motivaciones, el contexto en el que se encuentra el estudiante, el proceso de diseño y una descripción muy detallada del producto en sí y de las dificultades técnicas superadas.

Palabras clave

.NET Core, React, Cifrado, Desarrollo Web, RESTfull.

Keywords

.NET Core, React, Encryption, Web Development, RESTfull.

Contenido

1. Introducción

Se pretende diseñar y desarrollar un gestor de contraseñas seguro y cifrado donde el cliente pueda crear y almacenar contraseñas únicas y complejas para cada servicio en línea, sin necesidad de recordarlas todas. Además, este gestor puede generar contraseñas aleatorias y robustas para garantizar la máxima seguridad.

1.1 Contexto y motivación del proyecto

El contexto del estudiante es sumamente importante para entender la motivación de este proyecto. Este ha decidido enriquecer su experiencia realizando una estancia de movilidad Erasmus de un año de duración durante el cuarto curso de Ingeniería Informática.

El destino ha sido la Universidad Politécnica de Bucarest en la Facultad de Ciencias de la Computación. En la normativa de la universidad de destino las prácticas se realizan durante el período de vacaciones justo después de acabar tercer curso por eso el estudiante no ha podido realizarlas durante el primero o segundo semestre de cuarto, sino que las ha realizado en verano al finalizar el curso académico.

Esto explica por qué el trabajo de final de grado también ha seguido un proceso diferente al que se realiza en la Universitat Jaume I. En primer lugar, el estudiante, al igual que todos los estudiantes de la universidad de destino, ha buscado un profesor supervisor para que le guíe sobre el proyecto que tiene que realizar. El alumno piensa en un tema para el TFG que suponga un reto tecnológico para éste y en el que se puedan ver las habilidades aprendidas. Además de que debe existir una motivación útil para la sociedad, el entorno, para él mismo, monetaria, etc. Es decir, no debe ser un proyecto trivial.

Una vez propuesto el tema del proyecto y con el visto bueno del supervisor se queda inscrito en un "registro de temas de TFG ". Después, se procede a la realización del proyecto, proceso que se detalla en el capítulo: 2. Planificación de proyecto. Una vez realizado, el estudiante redacta la memoria con la ayuda y supervisión del tutor y se presenta el TFG en la universidad de origen.

El tema elegido por el estudiante es un gestor de contraseñas y las motivaciones son varias. La primera es que este proyecto resuelve el problema que se describe a continuación.

Con el auge de la tecnología que se ha vivido durante estas últimas décadas, y la creciente cantidad de servicios en línea y plataformas que se utilizan a diario, es más difícil recordar contraseñas únicas y seguras para cada cuenta. La mayoría de los usuarios reutilizan la misma contraseña para todos los sitios web, incluso contraseñas que dan acceso a información sensible

o financiera. Muchas veces se olvida incluso el nombre de usuario y para muchos es un calvario pasar por el proceso de recuperación de contraseña vía email, donde los servicios ya no dejarán poner una contraseña similar. De esta manera obligan al usuario a crear una diferente, la cual volverá a olvidar o la apuntará en un sitio inseguro como una notita al lado del portátil o un archivo de texto en el escritorio con todos los usuario y contraseñas de las páginas web.

Aquí es donde el gestor de contraseñas desarrollado por el estudiante entra en juego. Esta aplicación ofrece una solución segura y conveniente para administrar y proteger las contraseñas de los usuarios de forma eficiente. Aunque existen varias soluciones buenas en el mercado, no todas son las más seguras ni las más accesibles ya que, o son totalmente de pago o la versión gratis es bastante reducida. Por eso se presenta la motivación del estudiante de ofrecer públicamente y sin ánimo de lucro un gestor de contraseñas distinto de los demás, lo más seguro posible y que pueda ser utilizado tanto por el usuario final como en el ámbito corporativo, además de contar con funcionalidades clave.

La otra motivación para desarrollar un gestor de contraseñas es que, a diferencia de otras aplicaciones web, ésta debe estar cifrada y cumplir unos estándares de seguridad. Esto pone unos obstáculos tecnológicos desafiantes en el camino que dotan al estudiante de habilidades y conocimientos para seguir creciendo y ser un activo valioso en el mundo laboral.

1.2 Objetivos del proyecto

El gestor de contraseñas se fundamenta en tres pilares esenciales:

- **Seguridad:** Su objetivo principal es proteger de forma sólida las contraseñas con técnicas de encriptación avanzada.
- **Privacidad:** El usuario de esta aplicación debe confiar en que sus datos serán gestionados de forma privada, recolectándose lo menos posible y sin que se compartan con terceros bajo ninguna forma.
- **Conveniencia.** Se busca facilitar la vida del usuario mediante diversas funcionalidades (que se establecerán en el siguiente capítulo) y el acceso conveniente a sus credenciales con una interfaz fácil de usar e intuitiva.

1.3 Descripción del proyecto

El proyecto consta de una aplicación web con un servidor y una base de datos para guardar la información y una aplicación del lado del cliente que se ejecuta en el navegador para poder ser utilizada desde cualquier dispositivo. Se usan tecnologías modernas y potentes (React.js, ASP.NET Core) tanto para la implementación como para el desarrollo.

Ésta permitirá gestionar cuentas con una sola contraseña maestra ofreciendo productividad al usuario y confianza al utilizar la aplicación.

1.4 Estructura de la memoria

En primer lugar, en la memoria se presenta el capítulo de planificación del proyecto que detalla la metodología seguida y el seguimiento que se ha hecho de las tareas. En el capítulo siguiente se especifica el proceso de definición de producto y un análisis del entorno.

En el capítulo de la solución propuesta se detallan las tecnologías usadas, el porqué de estas tecnologías y cómo se ha llevado a cabo la implementación. Por último, se explican algunos detalles técnicos que tienen que ver con la seguridad del gestor de contraseñas.

2. Planificación del proyecto

2.1 Metodología

El proyecto que se ha realizado tiene algunas características peculiares frente a otros proyectos que se suelen presentar. En primer lugar, no se desarrolla en ninguna empresa, es un proyecto propio e individual. Esto ofrece mucha flexibilidad al estudiante para organizarse y planificarse de la manera más adecuada posible.

Debido a que el proyecto se tiene que entregar en su totalidad con una fecha límite, se dispone de un proyecto bien definido y con unos requisitos estables que no van a cambiar antes de tiempo y además es individual (no se puede trabajar en paralelo). La metodología de trabajo que se ha seguido y la más acertada es Waterfall.

Esta es una metodología secuencial y lineal en la que cada etapa del proyecto se completa antes de pasar a la siguiente. Se tienen etapas bien definidas y se trabaja completándolas de una manera secuencial puesto que la mayoría de las tareas dependen de la anterior. Además, cada quince días desde la fecha de comienzo hasta la de finalización, el estudiante se ha reunido con el profesor supervisor mostrándole los avances, hablando sobre las dificultades técnicas y obteniendo una retroalimentación de su progreso actual.

A continuación, se muestra en la Figura 2.1 una tabla de las etapas y tareas establecidas y un coste en semanas orientativo de lo que debería durar cada etapa.

Número de etapa	*Tarea*	*Semanas*
1	"Branstoming" y selección de requisitos	0,5
2	Encuesta y entrevistas	0.5
3	Estudio de mercado y definición de requisitos	0.5
4	Estudio y elección de Arquitectura, base de datos, tecnologías Backend, Frontend.	0,5
5	Estudio del cifrado	1
6	Aprendizaje Backend	1
7	Desarrollo y Modelado Base de Datos	1
8	Implementación API y servicios	2

9	Implementación de cifrado	2
10	“Testing” del Backend	1
11	Aprendizaje Frontend	1
12	Implementación Frontend de autenticación, registro, usuario y bienvenida.	1
13	Implementación Frontend del gestor de contraseñas	4
14	Iconos y estilizado	2
15	Redacción de la memoria	3

Figura 2.1. Tabla de planificación. Cada semana se estima que el estudiante deberá trabajar 4 horas al día de lunes a viernes.

2.2 Riesgos, coste y recursos.

Al ser un proyecto que se desarrolla en la facultad no presentan ningún riesgo más importante que no llegar a entregar el proyecto. Esto tiene una serie de ventajas y desventajas a la hora de calcular los costes:

- **Ventajas:** No hay ninguna obligación de realizar el proyecto, el tema se puede cambiar, se puede empezar con mucho tiempo de antelación y no se debe realizar en un período finito de prácticas. Además de tener un horario muy flexible y libertad a la hora de organizarse, desarrollar y escoger requisitos y tecnologías para trabajar.

- **Desventajas:** el estudiante debe asumir los costes de electricidad usados para alimentar la estación de trabajo y disponer de un equipo en buenas condiciones para desarrollar código. Deberá planificar muy bien qué es lo que quiere hacer y empezar desde cero con todo el proyecto. Al ser individual no podrá pedir ayuda a la hora de desarrollar, sólo consejo a su supervisor. Se debe hacer en tiempo libre y compaginar con los estudios del segundo semestre de cuarto curso.

Por tanto, más allá de la fatiga no ha habido más costes ya que los costes de electricidad se consideran ínfimos y el equipo para trabajar ya se tenía.

En cuanto a los recursos físicos, el proyecto se ha desarrollado en su totalidad en el domicilio del estudiante con un Laptop Lenovo Legion Y540 i7 16Gb RAM, 512 Gb de almacenamiento, gráfica RTX 2060 como equipo de trabajo y un monitor LG de 27’ IPS 4K para una mejor productividad. El rendimiento del equipo ha sido impecable para soportar todo el software de desarrollo. En cuanto a las herramientas que se han utilizado:

- **Visual Studio Community:** para desarrollar el Backend y hacer testing.

- **Visual Studio Code:** con algunas extensiones de JavaScript y la extensión de React para desarrolladoras en el navegador Chrome.

- **Git y GitHub:** para mantener un control de versiones y alojar el código en la nube como copia de seguridad en caso de que el equipo fallara.

- **Docker:** para desplegar la base de datos en un entorno controlado.

2.3 Seguimiento de proyecto

Gracias a que el estudiante tiene libertad para organizarse, se han cumplido las tareas de las etapas en el período especificado. La forma en que se ha planificado ha dejado margen para algunas etapas, pero para otras ha sido necesario invertir más horas extra para llegar a completar la tarea en el plazo, ya que, como las tareas dependen la una de la otra no era posible invertir muchos más días porque se ponía en riesgo la fecha de entrega.

A continuación, se muestra una tabla con el seguimiento que se ha hecho y las etapas de trabajo. Como se ha mencionado, un día de trabajo se considera 4 horas. En las etapas que ha sido necesario invertir más horas por día se muestran las horas por día extra que se han tenido que invertir para llegar al plazo.

Numero de etapa	*Tarea*	*Periodo*	*Horas extra por día*
1	"Branstoming" y selección de requisitos	27/02- 05/03	0
2	Encuesta y entrevistas		
3	Estudio de mercado y definición de requisitos	06 /03 - 12/03	0
4	Estudio y elección de Arquitectura, base de datos, tecnologías Backend, Frontend.		
5	Estudio del cifrado	13/03 - 19/03	0
6	Aprendizaje Backend	20/03 – 26/03	0
7	Desarrollo y Modelado Base de Datos	27/03 – 02/04	0
8	Implementación API y servicios	03/04 – 16/04	1
9	Implementación de cifrado	17/04 – 30/04	1
10	"Testing" del Backend	01/05 – 07/05	5
11	Aprendizaje Frontend	08/05 – 14/05	0
12	Implementación Frontend de autenticación, registro, usuario y bienvenida.	15/05 – 21/05	0
13	Implementación Frontend del gestor de contraseñas	22/05 – 11/ 06	1
14	Iconos y estilizado	12/06 – 25/06	0
15	Redacción de la memoria	26/06 – 21/07	0

Figura 2.2. Tabla de seguimiento.

Como se puede observar la estimación ha sido bastante acertada a pesar de que en los períodos de desarrollo se ha necesitado una hora más.

Donde sí la estimación ha fallado estrepitosamente ha sido en la etapa de testing. Ha sido muy difícil depurar después de implementar el cifrado ya que las herramientas de depuración convencionales no han sido efectivas. Por eso ha sido necesario invertir muchas más horas para quitar los bugs y hacer que el Backend funcione.

3. Análisis y especificación de requisitos

En este capítulo se detalla la obtención de requisitos y se especifican las funcionalidades que va a tener la aplicación. Se realizan entrevistas y encuestas para afinar este proceso junto con un estudio de mercado.

3.1 "Brainstorming"

Primeramente, se realiza una etapa de "brainstorming" donde el estudiante genera ideas de requisitos que cree que podrían encajar con los objetivos del proyecto y que serían útiles tanto en el ámbito personal como empresarial. El resultado se muestra en la siguiente lista:

- Generación automática de contraseñas seguras
- Indicador para contraseñas débiles
- Recordatorios para cambiar las contraseñas periódicamente
- Cifrado de datos sensibles
- Guardar tarjetas bancarias
- Guardar notitas
- Almacenar datos de contacto
- Guardar identidades
- Enviar mensajes entre usuarios
- Autocompletado
- Sincronización multiplataforma
- Autentificación de dos factores
- Compartir contraseñas seguramente
- Políticas de seguridad
- Administración centralizada
- Descargar datos en formato CSV
- Bloqueo de cuenta después de varios intentos fallidos de inicio de sesión
- Selfi en caso de autentificación fallida
- Acceso solo a determinados dispositivos autorizados
- Autentificación con llave de seguridad física
- Política de privacidad y "no-logging"

3.2 Estudio de requisitos: selección y exclusión.

En el segundo paso el estudiante (junto con el consejo del supervisor) hace un estudio de cada uno de los requisitos para determinar si serán relevantes, si no suponen un riesgo para la seguridad, si se pueden implementar en un tiempo razonable, etc. A continuación, se muestran los requisitos excluidos y el motivo de la exclusión:

a) **Autocompletado:** Fueron descubiertas dos vulnerabilidades publicadas en el IFIP SEC 2020 [1] por las cuales una página web maliciosa podría hacerse con las claves de inicio de sesión que el usuario tenia guardadas en la bóveda si tenían la función de autocompletado activa en su gestor de contraseñas. La primera consiste en que "*los campos de inicio de sesión se rellenan con un nombre de usuario y una contraseña, a pesar de que las URL de origen y destino no coinciden*". La segunda expone que *"las políticas de autocompletado no distinguen entre HTTP (Hypertext Transfer Protocol) y HTTPS (Secure Hypertext Transfer Protocol) cuando intentan rellenar una credencial que ha sido almacenada con HTTPS en una versión HTTP del sitio Esto permitiría a un atacante 'man-in-the-middle' hacerse pasar por una versión HTTP de un sitio web popular y robar las credenciales de usuario almacenadas originalmente para la versión HTTPS."*

b) **Acceso sólo a determinados dispositivos autorizados** y **Bloqueo de cuenta después de varios intentos fallidos:** Estas son medidas de seguridad muy estrictas para un usuario común, además de resultar engorrosas.

c) **Selfi en caso de autentificación fallida**: Es una medida poco conveniente porque es posible que al autentificarse siempre se invada al usuario con una foto borrosa o mal hecha de algún familiar o amigo curioso y no mejora significativamente la seguridad.

d) **Autentificación con llave de seguridad física**: Lo mismo pasa con esta medida ya que muy pocas personas van a invertir sus recursos en comprar tales dispositivos.

e) **Sincronización multiplataforma** y **Autentificación de dos factores**: Estas son sin duda buenas funcionalidades, pero son muy costosas para una primera versión de la aplicación y es muy posible que no dé tiempo de realizar una aplicación web junto con dos aplicaciones móviles: para Android y iOS.

f) **Políticas de seguridad:** Estas políticas podrían parecer una buena medida y son fáciles de implementar, pero siempre resultan engorrosas en un gestor de contraseñas y la mayoría de servicios tienen las suyas propias. Además, el usuario se quedaría desconcertado si la aplicación dejara guardar una contraseña alfanumérica de 8 caracteres, pero otro servicio considera que 8 caracteres no es suficiente o que es obligatorio algún carácter especial y provocaría situaciones incomodas. Por eso es preferible que el usuario tenga flexibilidad a la hora de usar la aplicación además de que hoy día muchos sitios web "saltean" la contraseña según recomienda el Reglamento General de Protección de Datos (RGPD).

Una vez se tiene los requisitos excluidos se muestra también el motivo por el cual se han escogido de momento dos requisitos esenciales:

a) **Política de privacidad y "no-logging":** Brinda seguridad al usuario de que, aunque la aplicación esté cifrada, no se estén haciendo copias de sus datos en logs ni haciendo un seguimiento de su actividad.

b) **Administración centralizada**: aunque esta aplicación se use en ámbito empresarial (con su propio despliegue de base de datos y servidores) o personal (con un servidor mantenido por la comunidad) debe haber un administrador que dé de baja los usuarios que realicen un comportamiento fraudulento o dañino. Esto no quiere decir que el administrador pueda ver las contraseñas de los usuarios, pero sí tendrá acceso para listar los usuarios y borrarlos en caso de que sea necesario.

3.3 Encuesta

Después de esta criba el producto va tomando forma hacia una aplicación muy completa con varias funcionalidades muy interesantes. Pareciera que este gestor de contraseñas además de manejar contraseñas también permite guardar tarjetas bancarias, notitas, contactos, identidades y dispone de un chat entre usuarios, convirtiendo a este gestor en una aplicación completa de productividad.

Sin embargo, el siguiente paso es preguntar a potenciales usuarios que es lo que quieren. Para eso se realiza una encuesta preguntando varios grupos de personas principalmente estudiantes entre 16 y 26 años. La muestra de estudio también se puede dividir demográficamente entre rumanos o españoles ya que se han utilizado dos formularios iguales en contenido, pero distintos en idioma. En la Figura 3.1 se muestran los resultados obtenidos preguntando a los usuarios por el resto de los requisitos que no han sido ni excluidos ni seleccionados.

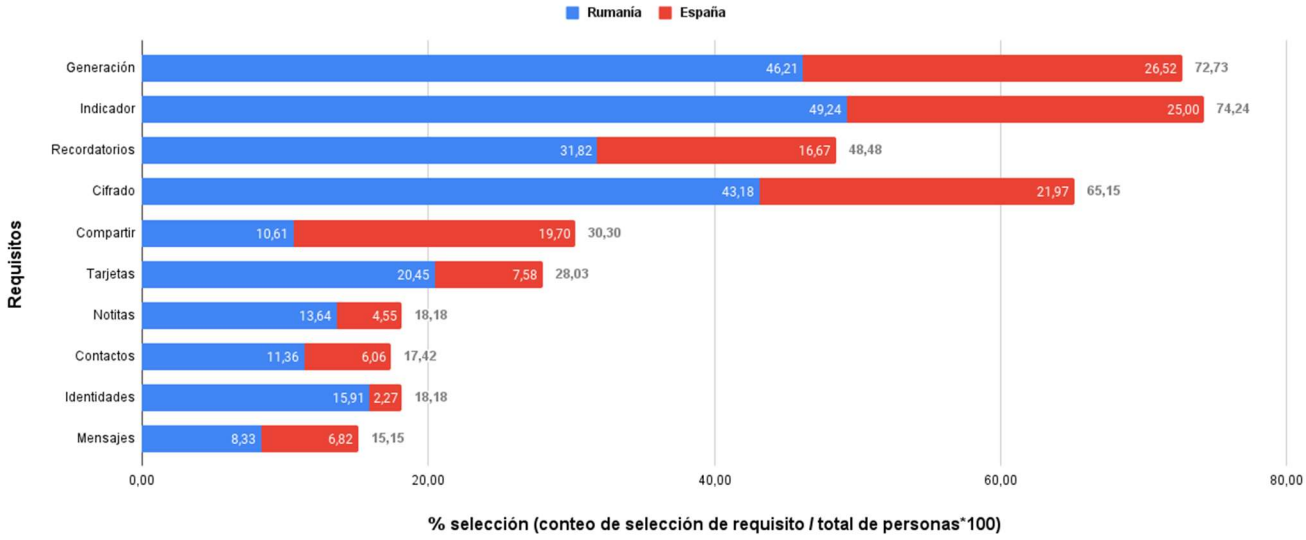


Figura 3.1: En el eje Y se observan los requisitos expresados de forma abreviada para que se pueda dibujar el gráfico. En el eje X se muestra el porcentaje de selección que ha tenido un requisito tanto para España, como Rumanía como el total (que se muestra al exterior del todo). El número de españoles encuestados es de 52 mientras que el número de rumanos asciende a 82 resultando en un total de 132 personas.

Fácilmente se puede observar que a los usuarios de ambas regiones no les importa tanto las funcionalidades extra que pueda tener un gestor de contraseñas como las notitas, contactos o demás si no que disponga de herramientas útiles para la gestión de contraseñas y que sus datos estén cifrados y seguros. Por eso se ha decido eliminar estos últimos 5 requisitos menos votados y dejarlos para implementar en siguientes versiones del producto.

El requisito de compartir contraseñas es el menos votado del bloque de requisitos que se va a implementar ya que es poco común que un estudiante comparta sus contraseñas privadas con más usuarios, pero lo cierto es que es una funcionalidad muy útil en el ámbito corporativo por eso se va a descartar.

3.4 Entrevista

Por último, se han realizado 4 entrevistas a personas que usan gestores de contraseñas en su día a día en el ámbito empresarial y todas han hecho hincapié en 3 puntos importantes:

- El primero es la importancia del cifrado de datos. El administrador del servidor no debería tener acceso a los datos de los usuarios ni a la base de datos, solo el propio usuario debería tener acceso a sus datos. Gracias a este punto reforzamos la confianza en los objetivos y funcionalidades elegidas.
- El segundo punto se refiere al hecho de que compartir contraseñas con los demás usuarios es importante en el mundo empresarial ya que facilita el flujo de trabajo en muchas ocasiones. Además, la inclusión de permisos de acceso es una idea magnífica para el control y la seguridad.
- El tercer punto describe la necesitad de los usuarios de organizar sus contraseñas en capetas o grupos, funcionalidad básica que se ha pasado por alto. Viendo el gran peso que tiene esta característica y el hecho de que se puede completar en un tiempo razonable se añadirá a la lista de requisitos.

Así pues, gracias a las entrevistas se ha reforzado el nivel de confianza que existía en la selección de características del producto, se han refinado algunos y además se ha descubierto una funcionalidad que no existía y que era sumamente importante.

3.5 Estudio de mercado

En esta sección se analizarán algunas características y funcionalidades de los gestores de contraseñas considerados más seguros hoy día. Como primer enfoque se ha probado la versión gratuita de cada gestor de contraseñas, además de leer el *whitepaper* [2][3][4] para extraer la información sobre seguridad y cifrado que hace falta. Con estos datos se rellena la Figura 3.2 para poder hacer una comparación de qué es lo que ofrece cada uno de los gestores de contraseñas. Hay que destacar que las características mostradas forman parte del plan gratuito ya que, a excepción de KeePass, todos son de pago.

	LastPass	KeePass	NordPass	Dashlane	Bitwarden
Algoritmos de cifrado	AES-256 PBKDF2	AES-256 Argon2	xChaCha20 Argon2id	AES-256 PBKDF2	AES-256 PBKDF2
Indicador de fortaleza de contraseña	SI	SI	NO	SI	SI
Generador de contraseña	SI	SI	SI	SI	SI
Compartir contraseña	1 persona	NO	NO	SI	1 persona
Código Abierto	NO	SI	NO	NO	SI
Lugar de Almacenamiento	Nube	Local	Nube	Nube	Nube
Lugar de cifrado	Cliente	Cliente	Cliente	Cliente	Cliente

Figura 3.2: Tabla comparativa.

Como se puede observar la mayoría de los gestores de contraseñas, como LastPass, KeePass, Dashlane y Bitwarden, utilizan algoritmos de cifrado fuertes como AES-256 y PBKDF2. NordPass utiliza el algoritmo xChaCha20 junto con Argon2id, lo que también es una opción segura. Esto ofrece una guía sobre que algoritmos de cifrado usar en la aplicación.

Las opciones de indicador de fortaleza y generador de contraseña están presentes en casi todos los gestores de contraseñas cosa que refuerza nuestra confianza en las funcionalidades elegidas en la sección anterior.

Todos los gestores de contraseñas guardan los datos en la nube a excepción de KeePass que los guarda en la propia máquina cliente. Este es un enfoque mucho más seguro, pero menos conveniente ya que no es posible compartir contraseñas u otros datos entre usuarios de la aplicación. Cabe destacar que el resto de los gestores de contraseñas no tienen límite de usuarios a la hora de compartir datos, pero esta funcionalidad solo está disponible en la versión de pago. Como el gestor de contraseñas que desarrolla el estudiante es gratis esta funcionalidad no va a tener límites.

Todos los gestores de contraseñas utilizan el cifrado del lado del cliente, lo que proporciona una capa adicional de seguridad y privacidad, pero la aplicación del estudiante realizará el proceso de cifrado del lado del servidor. Esto es así debido a que se busca un equilibrio entre seguridad y conveniencia y es posible que el rendimiento de la aplicación baje considerablemente si el usuario

cuenta con un equipo con pocos recursos: como pueden ser móviles de baja calidad u ordenadores viejos. Esto lo ha podido comprobar el estudiante de primera mano ya que ha usado Bitwarden y Dashlane durante más de dos años y ha tenido la oportunidad de comprobar el bajo rendimiento que ofrecen en equipos de bajas capacidades. También hay que mencionar que en equipos de gama media y alta el rendimiento es impecable.

Otra de las motivaciones para cifrar del lado del servidor es que pone el camino del desarrollador más retos tecnológicos que deberá superar, dotándolo de más habilidad y experiencia para su futuro laboral además de realizar un producto distinto de lo que se encuentra en el mercado actual.

Debido al cifrado en el servidor hay que añadir dos características para mantener la seguridad. La primer es el cifrado durante la transmisión de mensajes entre el cliente y el servido que se puede conseguir con la implementación de un certificado SSL. La segunda es el cifrado de la memoria del servidor ya que, si un atacante obtuviera acceso a la máquina, no podría extraer los datos de los usuarios.

Esto ya le pasó a la aplicación LastPass [5] que fue víctima de dos ataques: una en el año 2015 donde vieron comprometidas las direcciones de correo electrónico de los usuarios, los recordatorios de contraseñas, las diferentes sales de cada usuario y los algoritmos de autenticación; y otra en 2022 donde además de robar datos de usuarios (que por suerte estaban cifrados) robaron también partes del código fuente e información técnica sobre el gestor , y otra información no cifrada que podría ser usada para realizar ataques de phishing dirigidos.

2.6 Casos de uso y requisitos definitivos

A continuación, se presenta un diagrama de casos de uso útil para entender las acciones que distintos actores pueden realizar en la plataforma. En el gráfico 3.3 se muestran algunas funcionalidades con la palabra CRUD delante. Esto se ha hecho para tener un gráfico simplificado y quiere decir que se podrán realizar acciones para crear, leer, modificar o borrar el objeto.

Por ejemplo, el caso de uso CRUD Cuentas significa que el usuario puede crear una nueva cuenta, acceder a ella, modificarla o borrarla.

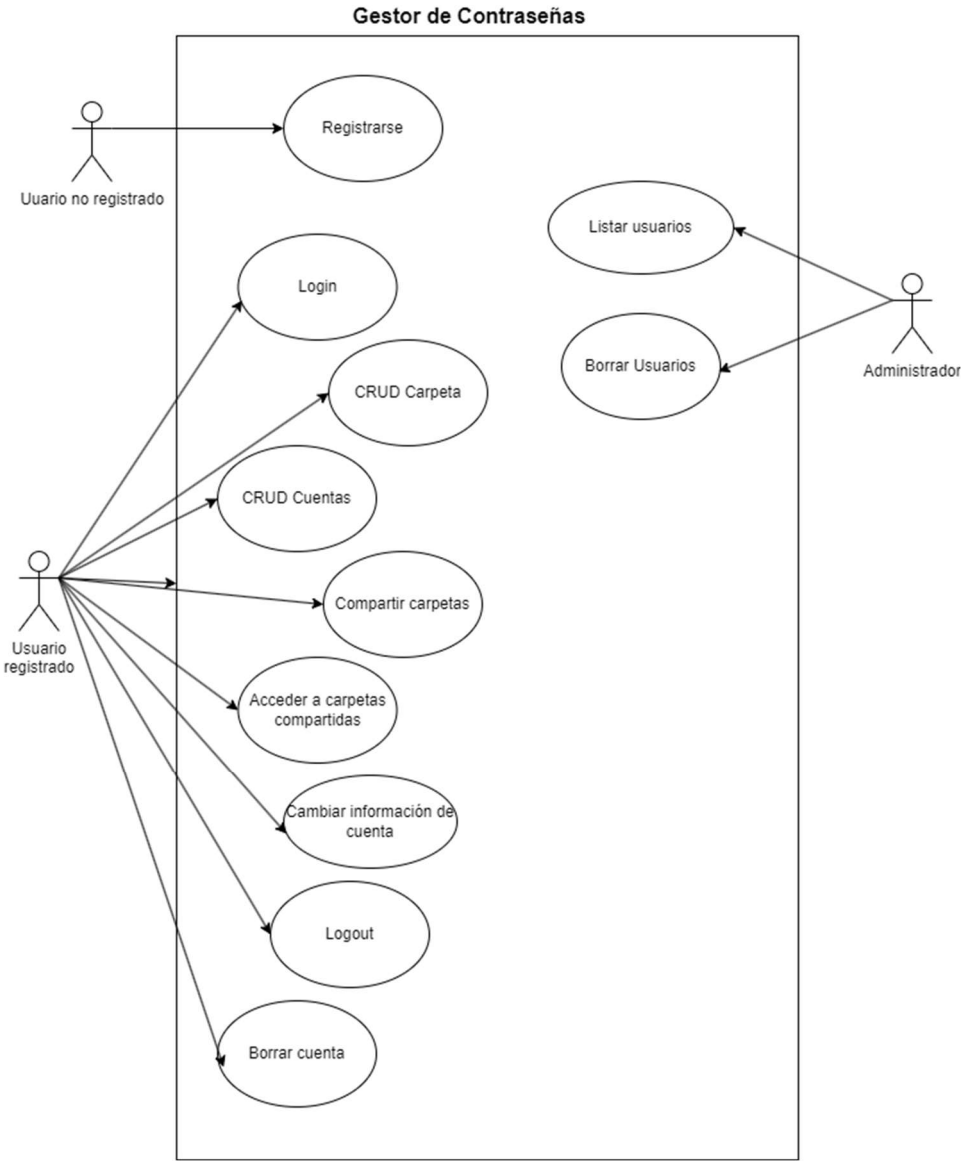


Figura 3.3: Diagrama de casos de uso

Con esta información ya se tiene la lista final de requisitos que incluye: los dos requisitos elegidos en la fase de selección, los cinco que sobresalieron en la fase de encuesta, la funcionalidad que surgió en base a las entrevistas y los dos últimos debido al análisis de mercado de esta sección como se muestra en la Figura 3.4

Numero de requisito	Descripción	Categoría
1	CRUD Cuentas	Conveniencia
2	Administración centralizada	Conveniencia
3	Política de privacidad y "no-logging"	Conveniencia
4	Generación de contraseñas	Conveniencia
5	Indicador de contraseña débil	Conveniencia
6	Recordatorios para cambiar la contraseña	Conveniencia
7	Cifrado de datos en la base de datos	Seguridad
8	Compartir carpetas de cuentas	Conveniencia
9	Permisos a las carpetas compartidas	Seguridad
10	Fecha de expiración de permisos	Seguridad
11	Organización en carpetas	Conveniencia
12	Cifrado de memoria	Seguridad
13	Interfaz amigable centrada en la productividad	Conveniencia

Figura 3.4: Requisitos finales.

4. Solución propuesta

Con toda esta información la mejor opción y la más utilizada para un gestor de contraseñas es crear una aplicación web. Una aplicación web es un software accesible a través de un navegador web que permite a los usuarios interactuar y realizar diversas tareas en línea. No requiere instalación y funciona en múltiples dispositivos. Se aloja en servidores remotos y ofrece una experiencia interactiva, brindando servicios, información o funcionalidades a los usuarios a través de internet.

En esta sección, se describirá la arquitectura tecnológica de la aplicación web junto con el proceso de comunicación y mensajes. Se detallará la estructura del Frontend, desarrollado con tecnologías modernas para proporcionar una experiencia de usuario intuitiva. Además, se explorará el Backend, encargado de la lógica de negocio y la comunicación con la base de datos encriptada, asegurando la protección de la información confidencial. También se describirá el por qué se han utilizado estas tecnologías.

4.1 Arquitectura

La aplicación web se sustentará sobre una arquitectura de capas. Este es un enfoque de diseño para el desarrollo de aplicaciones que organiza su estructura en diferentes niveles o capas, cada una con una responsabilidad específica. Cada capa tiene funciones bien definidas y se comunica con las capas adyacentes, pero está independizada de las demás. La capa de presentación (frontend) se encarga de la interfaz de usuario, la capa de lógica de negocio (backend) gestiona la funcionalidad y reglas del negocio, y la capa de acceso a datos (base de datos) maneja el almacenamiento y recuperación de la información.

Se ha elegido esta arquitectura porque en el desarrollo de un gestor de contraseñas, la elección de una arquitectura de capas se presenta como una opción altamente ventajosa. Gracias a su capacidad para separar claramente las responsabilidades, esta arquitectura garantiza una estructura organizada y mantenible. Además, esta arquitectura favorece la reutilización de componentes, permitiendo un desarrollo eficiente y reduciendo la duplicación de código. Por último, la capacidad de escalar la aplicación de manera sencilla y la posibilidad de adaptarse a nuevos requisitos la convierten en una opción versátil y adaptable a las necesidades cambiantes del gestor de contraseñas.

En la Figura 4.1 se muestran los componentes que forman la aplicación web y el proceso de comunicación para acceder a un determinado recurso por parte del usuario. Después se hará una breve descripción de los pasos del proceso comunicativo:

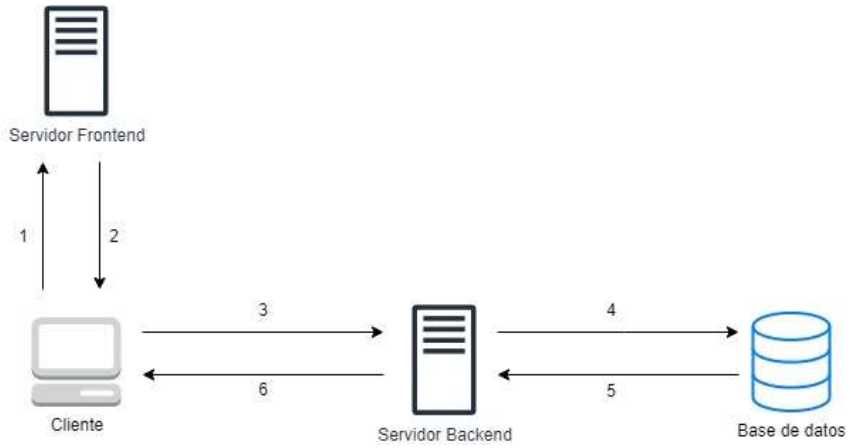


Figura 4.1: Componentes y comunicación.

Paso 1: El navegador del cliente solicita al Servidor Frontend el código de la aplicación web.

Paso 2: El Servidor Frontend envía el código de la aplicación al navegador y este lo ejecuta mostrando al cliente la interfaz.

Paso 3: El cliente al interactuar con la interfaz hace una petición (o varias) al Servidor Backend para acceder a un recurso.

Paso 4: El Servidor Backend procesa la petición y hace otra petición (o varias más) a la Base de datos en busca del recurso.

Paso 5: La base de datos devuelve el recuso al Servidor Backend donde es procesado según la lógica de la aplicación.

Paso 6: Por último, el servidor Backend envía el recurso procesado al navegador donde este interpreta la respuesta según el código que recibió previamente del Servidor Frontend y lo muestra al usuario.

4.2 Estilo arquitectónico

Para especificar cómo se van a comunicar los componentes de la aplicación web es necesario adoptar un estilo arquitectónico que se ajuste a la aplicación web. A continuación, se muestra una breve descripción de cada uno:

a) **RESTful**:[6] Se basa en el protocolo HTTP y utiliza sus métodos (GET, POST, PUT, DELETE) para acceder a recursos identificados por URLs únicas. Es simple y eficiente, permitiendo a los clientes obtener y manipular datos con facilidad.

b) **SOAP (Simple Object Access Protocol)**:[7] Es un protocolo basado en XML (eXtensible Markup Language) que define una estructura rígida para intercambiar mensajes entre aplicaciones. Emplea el protocolo HTTP, SMTP o TCP para la comunicación y ofrece características avanzadas pero mayor complejidad.

c) **GraphQL:** [8]Permite a los clientes especificar exactamente los datos que necesitan, evitando solicitudes innecesarias. Utiliza una única URL y es más flexible que RESTful, lo que lo hace ideal para aplicaciones con requisitos de datos complejos.

d) **gRPC:** [9]Basado en HTTP/2, es rápido y eficiente para la comunicación entre servicios distribuidos. Utiliza Protocol Buffers para describir los servicios y mensajes, siendo ideal para aplicaciones con alta demanda de rendimiento.

e) **WebSockets:** [10]Permite conexiones bidireccionales y en tiempo real entre clientes y servidores, lo que lo hace adecuado para aplicaciones que requieren actualizaciones continuas, como chats y juegos en tiempo real.

Para empezar, se puede descartar la comunicación con Web Sockets y gRPC porque no se adecuan a las necesidades de la aplicación: no es necesaria una continua conexión con el servidor ni estamos ante una aplicación de alto rendimiento.

Seguidamente, el uso de SOAP en una aplicación web hoy día es una opción desaconsejable debido a su complejidad y sobrecarga. SOAP utiliza XML, lo que puede resultar pesado y difícil de leer. Además, requiere más recursos y tiempo de procesamiento en comparación con tecnologías más modernas como RESTful o GraphQL, que ofrecen una comunicación más liviana, flexible y eficiente.

Por último, para una aplicación relativamente pequeña como un gestor de contraseñas, el uso de GraphQL puede resultar innecesariamente complejo. GraphQL es ideal para aplicaciones con requisitos de datos complejos y necesidades de flexibilidad en la obtención de información específica. Sin embargo, para una aplicación como un gestor de contraseñas, donde la interacción con la base de datos es relativamente directa, utilizar GraphQL agregaría una capa adicional de complejidad y aumentaría el tiempo de desarrollo.

Se busca un estilo arquitectónico que ofrezca una comunicación directa y eficiente con la base de datos. Esto simplificará la implementación y el mantenimiento de la aplicación. Por lo cual en este caso la opción que más se adecua es una aplicación RESTful.

En una aplicación RESTful, los mensajes se organizan y estructuran utilizando un conjunto de prácticas y convenciones bien definidas. La comunicación se basa en el protocolo HTTP, donde los mensajes se intercambian entre el cliente y el servidor a través de solicitudes y respuestas. Las solicitudes HTTP son enviadas por el cliente al servidor, mientras que las respuestas son generadas por el servidor y enviadas al cliente como resultado de una solicitud previa.

Cada mensaje HTTP tiene una estructura específica. Las solicitudes incluyen una línea de solicitud que indica el método HTTP (GET, POST, PUT, DELETE, etc.), la URL del recurso objetivo y la versión del protocolo. Además, pueden contener encabezados que especifican metadatos adicionales para la solicitud.

Las respuestas HTTP incluyen una línea de estado que indica el código de estado de la respuesta (por ejemplo, 200 para éxito, 404 para recurso no encontrado, etc.), la versión del protocolo y un mensaje de estado asociado al código. Además, también pueden contener encabezados que proporcionan información adicional sobre la respuesta.

Los mensajes en una aplicación RESTful generalmente se intercambian en formato JSON (JavaScript Object Notation) o XML. JSON es ampliamente utilizado debido a su legibilidad y facilidad de uso, y es el formato que se ha elegido para enviar y recibir datos en la aplicación web del estudiante.

4.3 Base de Datos

Una base de datos es una colección estructurada de información que se organiza y almacena en un sistema informático, permitiendo el acceso, búsqueda, modificación y gestión eficiente de los datos. En el contexto de una aplicación web como es el gestor de contraseñas que desarrolla el estudiante, una base de datos es esencial para almacenar y gestionar de forma segura la información confidencial de los usuarios, como nombres de usuario, contraseñas cifradas y otra información relevante.

La base de datos en un gestor de contraseñas juega un papel crucial en la seguridad y funcionamiento de la aplicación. Al utilizar una base de datos, se garantiza que los datos de los usuarios estén organizados y protegidos de forma adecuada. La información almacenada puede ser recuperada rápidamente cuando un usuario inicia sesión y necesita acceder a sus contraseñas almacenadas.

A la hora de implementar una base de datos en el proyecto será necesario escoger un motor de gestión de bases de datos. La diferencia entre una base de datos y un motor de gestión de bases de datos (DBMS) es que la base de datos es el lugar donde se almacenan los datos, mientras que el motor de gestión de bases de datos es el software que se encarga de administrar y gestionar esa base de datos para que las aplicaciones puedan acceder y trabajar con los datos de manera eficiente. El DBMS es responsable de manejar aspectos como la seguridad, la integridad de los datos, la recuperación de fallos y el rendimiento de la base de datos. Así pues, será necesario un BDMS de licencia libre y gratuito.

En el mercado actual los dos DBMS más utilizados son MySQL y PostgreSQL. PostgreSQL es conocido por ser más ANSI SQL compatible y tener una sintaxis más robusta y rica en comparación con MySQL, aunque las diferencias entre los dos para un proyecto como este son irrelevantes. Por eso se elige casi arbitrariamente el motor PostgreSQL.

4.3.1 Diseño

El siguiente paso es establecer las tablas, campos y relaciones que va a tener la base de datos para que pueda cumplir con los requisitos como se muestra en la Figura 4.2.

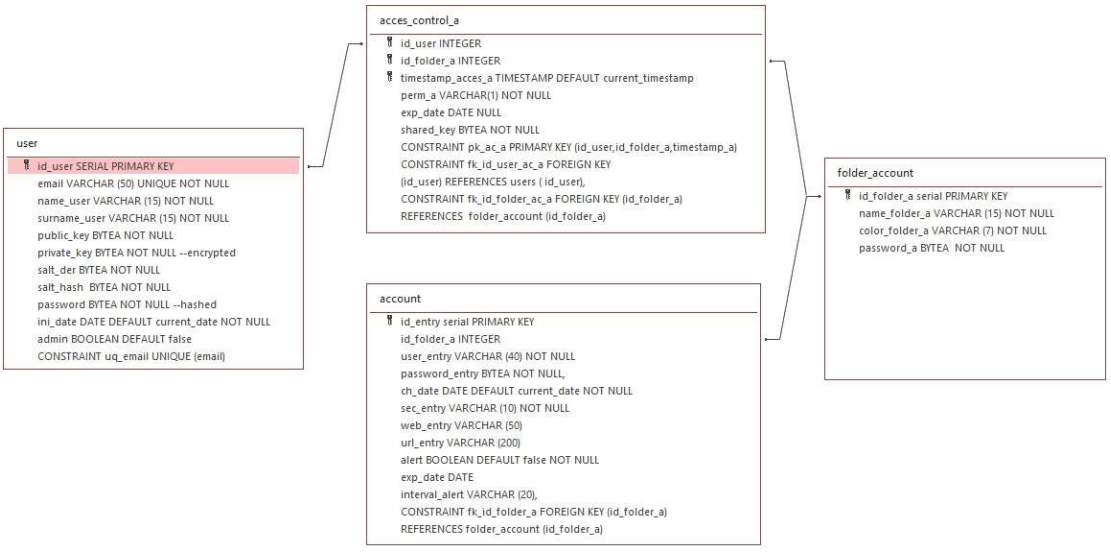


Figura 4.2: Esquema de tablas, campos y restricciones de integridad de la base de datos.

Como se puede observar hay algunos campos con el tipo de dato BYTEA. Este es un tipo de dato adecuado para guardar datos en binario y será utilizado para guardar los "salt" y los datos cifrados tanto llaves como datos del usuario. El esquema de acceso a los datos cifrados se detallará en el apartado de seguridad en capítulos posteriores.

La tabla *"folder_account"* almacena información sobre las carpetas de cuentas creadas por los usuarios, y la tabla *"account"* almacena información sobre las cuentas individuales que se encuentran dentro de cada carpeta. La relación entre ambas tablas se establece mediante la columna *"id_folder_a"* en la tabla *"account"* que es una clave foránea que referencia la tabla "*folder_account*". De esta manera una carpeta de cuentas puede contener varias cuentas, pero cada cuenta solo puede pertenecer a una única carpeta.

La tabla *"acces_control_a"* tiene una columna llamada *"id_user"* que actúa como una clave foránea que referencia la tabla usuarios. También tiene una columna *"id_folder_a"* que actúa como clave foránea que referencia a la tabla *"folder_account"*. Esta relación establece que un registro de control de acceso está asociado a un usuario específico y a una carpeta específica. Así es posible controlar qué usuarios tienen acceso a determinadas carpetas de cuentas, permitiendo que varios usuarios puedan compartir acceso a una misma carpeta. Además de tener un campo permisos ORWA (propietario, leer, modificar, agregar respectivamente) y el poder establecer una fecha de expiración para que el permiso ya no sea válido pasada esa fecha.

También se ha añadido a la clave primaria el campo *"timestamp_acces_a"* para asegurar compatibilidad con futuras funcionalidades de esta aplicación, aunque de momento no es necesario porque el acceso se otorga en base a la existencia de un registro en dicha tabla.

Otro campo que se ha añadido es el de *"color_folder_a"* ya que será útil en la interfaz de usuario poder asignar colores a las carpetas.

Las únicas reglas de identidad que existen a nivel de base de datos es que el campo email debe ser único y que la gran mayoría de atributos no pueden ser nulos.

Gracias a la ayuda del DBSM se pueden declarar los campos *"id_user"*, *"id_folder_a"* y *"id_entry"* como *"serial"*. Cuando se define un campo como *"serial"* en PostgreSQL, se crea una secuencia asociada a ese campo. La secuencia es un objeto de base de datos que genera automáticamente valores numéricos secuenciales y únicos cada vez que se inserta un nuevo registro en la tabla. Por lo tanto, el campo *"serial"* actúa como una clave primaria autoincremental y es necesario incluir un id cuando se va a crear un nuevo usuario, carpeta o entrada.

Para mejorar la compatibilidad futura se ha se ha establecido añadir el sufijo "_a" a las tablas y a los campos de dichas tablas. Así cuando se añadan tablas como *"acces_control_c"* y *"folder_card"* con campos como *"id_folder_c"*, o tablas *"folder_note"* con campos *"id_folder_n"* la estructura del proyecto será mucho más clara y limpia.

4.3.2 Despliegue

Para poder trabajar de una forma cómoda se utiliza un contenedor de Docker. Este es una unidad de software ligera, portátil y autónoma que incluye todo lo necesario para ejecutar una aplicación, incluidas las bibliotecas, dependencias y configuraciones del sistema. Estos contenedores se basan en la tecnología de virtualización a nivel de sistema operativo, lo que les permite ejecutarse de manera aislada en un sistema anfitrión compartiendo los recursos del kernel del sistema operativo subyacente.

En lugar de crear máquinas virtuales completas, los contenedores Docker comparten el núcleo del sistema operativo del host, lo que los hace más eficientes y rápidos de iniciar y detener. Cada contenedor tiene su propio sistema de archivos aislado, lo que permite que múltiples contenedores se ejecuten en el mismo sistema sin interferir entre sí.

Primeramente, se crea el archivo “create.sql” (Figura 4.3) donde se especifican las tablas los campos y las relaciones con lenguaje SQL exactamente igual al diseño.

```sql
CREATE TABLE users (
    id_user serial PRIMARY KEY,
    email VARCHAR (50) UNIQUE NOT NULL,
    name_user VARCHAR (15) NOT NULL,
    surname_user VARCHAR (15) NOT NULL,
    public_key BYTEA NOT NULL,
    private_key BYTEA NOT NULL, --encrypted
    salt_der BYTEA NOT NULL,
    salt_hash BYTEA NOT NULL,
    password BYTEA NOT NULL, --hashed
    ini_date DATE DEFAULT current_date NOT NULL,
    admin BOOLEAN DEFAULT false NOT NULL,
    CONSTRAINT uq_email UNIQUE (email)

);
CREATE TABLE folder_account (
    id_folder_a serial PRIMARY KEY,
    name_folder_a VARCHAR (15) NOT NULL,
    color_folder_a VARCHAR (7) NOT NULL,
    password_a BYTEA NOT NULL
);

CREATE TABLE account (
    id_entry serial PRIMARY KEY,
    id_folder_a INTEGER,
    user_entry VARCHAR (40) NOT NULL,
    password_entry BYTEA NOT NULL, --encrypted
    ch_date DATE DEFAULT current_date NOT NULL,
    sec_entry VARCHAR (10) NOT NULL,
    web_entry VARCHAR (50) NULL,
    url_entry VARCHAR (200) NULL,
    alert BOOLEAN DEFAULT false NOT NULL,
    interval_alert VARCHAR (20) NULL,
    exp_date DATE NULL,
    CONSTRAINT fk_id_folder_a FOREIGN KEY (id_folder_a) REFERENCES folder_account (id_folder_a)
);

CREATE TABLE acces_control_a (
    id_user INTEGER,
    id_folder_a INTEGER,
    timestamp_a TIMESTAMP DEFAULT current_timestamp,
    perm_a VARCHAR (1) NOT NULL,
    exp_date_a DATE NULL,
    shared_key_a BYTEA NOT NULL,
    CONSTRAINT pk_ac_a PRIMARY KEY (id_user, id_folder_a, timestamp_a),
    CONSTRAINT fk_id_user_ac_a FOREIGN KEY (id_user) REFERENCES users ( id_user),
    CONSTRAINT fk_id_folder_ac_a FOREIGN KEY (id_folder_a) REFERENCES  folder_account (id_folder_a)
);
```

Figura 4.3: Contenido del fichero create.sql

Después se configura un archivo Docker Compose (Figura 4.4) que es el que realiza toda la configuración del contenedor.

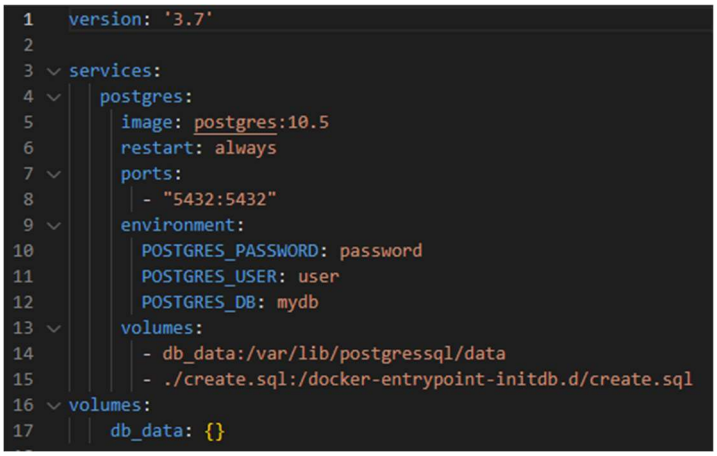


Figura 4.4: Contenido del fichero docker-compose.yml

En este fichero se establece que hay un servicio llamado postgres y se utiliza la imagen postgres10.5 (miniatura del sistema operativo Linux más el motor de base de datos PostgreSQL). Se especifica que el puerto 5432 del huésped se mapea al puerto 5432 del anfitrión. Luego se proporcionan las variables de entorno necesarios para configurar la base de datos.

Por último se crea un volumen persistente (línea 16 y 17) llamado "db_data" y se guarda el contenido de la base de datos (`/var/lib/postgressql/data`) en él. Así se consigue que, aunque la base de datos caiga o el contenedor se pare los datos no se pierdan.

También se coloca el script de inicialización que se ha creado en la ruta especial: `docker-entrypoint-initdb.d/`

Finalmente se ejecuta el comando `docker-compose -f docker-compose.yml up -d` para poner el contenedor en marcha.

4.4 Backend

En esta sección, se presentará el Backend del Gestor de Contraseñas, una pieza fundamental de la aplicación. El Backend actúa como un motor central que gestiona la lógica de negocio, procesa las solicitudes del cliente y se comunica con la base de datos para almacenar y recuperar información confidencial de manera segura. Se describirá en detalle la arquitectura utilizada, las tecnologías implementadas y las principales funcionalidades que aseguran un rendimiento óptimo y la protección de los datos de los usuarios. Gracias a un diseño cuidadoso y el uso de prácticas modernas de desarrollo, se ha creado un Backend robusto y escalable que respalda las operaciones del Gestor de Contraseñas, garantizando la confidencialidad y privacidad de los usuarios.

4.4.1 Tecnología utilizada

Hoy día, existen varias tecnologías populares para desarrollar el Backend de una aplicación web. En la Figura 4.5 se muestra una estadística de tecnologías más usadas para desarrollar Backend.

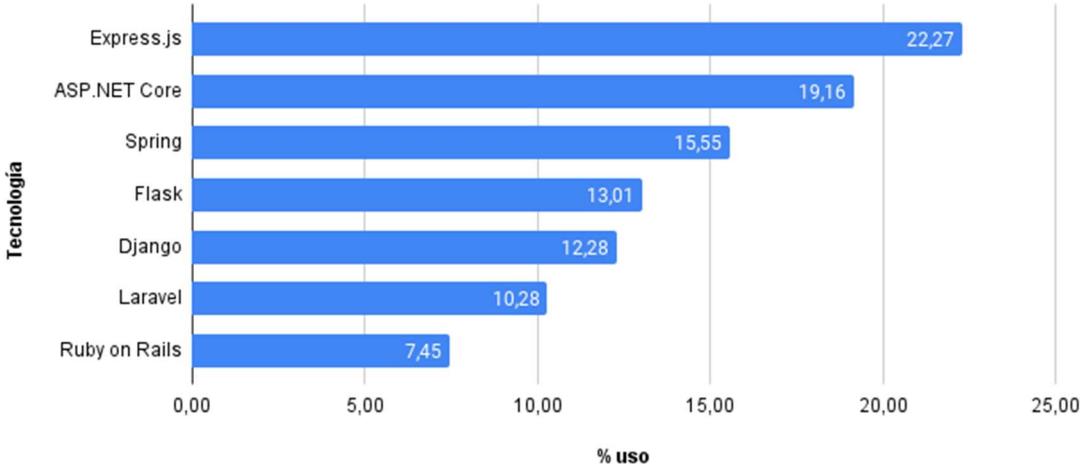


Figura 4.5: Tecnologías Backend más usadas por desarrolladores profesionales. Datos extraídos de la encuesta anual de Stackoverflow Insights 2021 con un total de 49 941 respuestas.[11]

Uno de los criterios para escoger la tecnología es que sea la más usada, así el estudiante se habrá formado en algo que pueda serle lo más útil posible en el mundo laboral. La opción que lidera la encuesta es Express.js. Sin embargo, el lenguaje de programación de este es JavaScript al igual que el Frontend (ya que es casi un estándar usar JavaScript para aplicaciones web del lado del cliente) y el estudiante persigue formarse y aprender cosas nuevas.

La siguiente opción es ASP.NET Core. Este es un framework de código abierto desarrollado por Microsoft que se utiliza para construir aplicaciones web y servicios en el lado del servidor. Es una evolución de ASP.NET y está diseñado para ser más ligero, modular y de alto rendimiento. ASP.NET Core es multiplataforma, lo que significa que puede ejecutarse tanto en sistemas Windows como en Linux y macOS. Esta es la opción escogida para realizar el desarrollo Backend. Además, es una de las tecnologías mejor pagadas en este ámbito de desarrollo.

Elegir Spring o Flask tampoco es una opción muy atractiva ya que el estudiante ha desarrollado en proyectos anteriores de la universidad APIs con Spring y Flask y no supondría aprender algo nuevo. Por no mencionar que están por debajo de ASP.NET Core[12]

La versión de .NET que se utiliza es .NET 6 ya que esta es una versión estable que incluye varias novedades y cambios importantes en la plataforma.NET:

a) Una de las mejoras más destacadas es un aumento de rendimiento de hasta el 90% (casi el doble) en varios casos. Esto consolida la posición de la plataforma como una de las de mayor rendimiento y, probablemente, la plataforma convencional no especializada de mayor potencia.

b) Nueva versión (10) del lenguaje de programación C#. En esta versión se introducen multitud de novedades como las cláusulas *"using"* globales e implícitos, mejoras en funciones lambda, los archivos con ámbito de espacio de nombres, mejoras en cadenas interpoladas, mejoras en estructuras, etc.

c) "*Hot Reload*", que permite cambiar el código "en caliente", mientras se está depurando sin necesidad de parar y reiniciar las aplicaciones.

d) Introducción de las "Minimal APIs", que permiten crear APIs sencillas de manera veloz y sin apenas código.

e) Añadido de nuevas APIs para dar soporte a HTTP/3, APIs para matemáticas, para trabajar con JSON o para manejar fechas y horas de manera independiente.

f) Nuevo soporte para OpenSSL 3. inclusión del esquema de cifrado ChaCha20Poly1305, y varias mitigaciones en tiempo de ejecución, en concreto W^X y CET en procesadores Intel.

4.4.2 ORM

Un ORM (Object-Relational Mapping) es una técnica de programación que permite a los desarrolladores interactuar con una base de datos relacional utilizando objetos y lenguajes de programación orientados a objetos, como Java, C#, Python, entre otros.

El objetivo principal de un ORM es abstraer la complejidad de las operaciones de base de datos y facilitar la manipulación de datos utilizando código más familiar y cercano a la lógica de la aplicación. En lugar de escribir consultas SQL directamente, el ORM mapea las tablas de la base de datos a clases y los registros a instancias de esas clases.

De esta manera, se pueden realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) y consultas de base de datos utilizando métodos y propiedades de las clases, lo que facilita el desarrollo y mantenimiento del código. Por eso si se va a desarrollar en .NET el mejor ORM es Entity Framework desarrollado por la propia Microsoft.

Para ello se necesita instalar el paquete NuGet *"Npgsql.EntityFrameworkCore.PostgreSQL".* A continuación, es necesario escribir en el archivo "appsettings.json" la cadena de conexión que en caso de esta aplicación será la de la figura 4.6:

```
"ConnectionStrings": {
  "WebApiDatabase": "Host=localhost; Database=mydb; Username=user; Password=password"
},
```

Figura 4.6: Cadena de conexión de la base de datos PostgreSQL.

Después es necesario crear los modelos que mapeen las entidades de la base de datos. Esto se hace modernamente con una migración inversa: un Scaffold como aparece en la Figura 4.7. Esto crea en el proyecto un directorio llamado "Model" (véase Figura 4.8) que contiene una clase por cada entidad mas una clase de configuración del contexto de la base de datos donde se detalla la información sobre como se relacionan las tablas entre ellas.

```
> dotnet ef dbcontext scaffold "Host=localhost;Database=mydb;Username=user;Password=password" Npgsql.EntityFrameworkCore.PostgreSQL -o Model
```

Figura 4.7: Comando que genera el código del modelo.

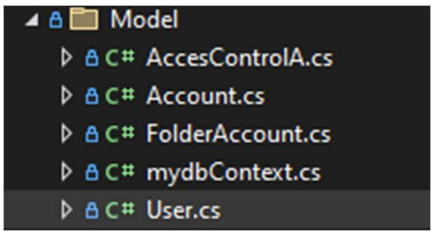


Figura 4.8: Estructura del directorio Model.

Por último, se configura la inyección de dependencias en el fichero "Program.cs" como se muestra en la Figura 4.9

```
builder.Services.AddDbContext<mydbContext>(options =>
    options.UseNpgsql(builder.Configuration.GetConnectionString("WebApiDatabase")));
```

Figura 4.9: Configuración del servicio mydbContext para que se pueda solicitar una clase de este servicio en cualquier parte del código y se pueda acceder a la base de datos.

Ahora ya es posible usar los métodos de "mydbContext" desde un controlador como en la Figura 4.10.

```
[ApiController]
[Route("[controller]")]
public class AuthController : ControllerBase
{
    private readonly mydbContext _context;

    public AuthController(mydbContext context)
    {
        _context = context;
    }
```

Figura 4.10: Se define un atributo privado y se inicializa con la case. La inyección de dependencias se encargará de pasarle una instancia de mydbContext al constructor del controlador.

4.4.3 DTO

Un DTO (Data Transfer Object) es un patrón de diseño utilizado en el desarrollo de software para transferir datos entre distintas capas o componentes de una aplicación. Su principal objetivo es encapsular y agrupar datos de forma estructurada y desacoplada, permitiendo el intercambio de información de manera eficiente.

Mas sencillamente, un DTO es una clase simple que contiene atributos que representan los datos que se desean transferir. Estas clases suelen ser objetos ligeros y con poca o ninguna lógica de negocio, ya que su principal función es transportar datos de un lugar a otro. En este proyecto se usará para la comunicación entre capas. El Frontend envía un objeto JSON como solicitud y este se mapea automáticamente a un DTO especifico. Así los datos están en un objeto fáciles de acceder y no en formato cadena.

Se han creado dos directorios: "Request" y "Response" que albergaran cada uno de ellos las clases DTO para peticiones y respuestas respectivamente. No todas las peticiones que se hacen del Frontend al Backend tienen un DTO asociando necesariamente. Es posible hacer un GET a un recurso pasando el id como parámetro en la ruta.

Tampoco es necesario un DTO para todas las respuestas ya que es muy común que después de hace un POST se devuelva simplemente un código de estado HTTP.

Incluso un DTO no tiene que representar obligatoriamente solo los datos de un objeto de dominio o una tabla de la base de datos, sino que puede tener varios campos diversos como pasa con el DTO "SpecFolderAResponse". Como se observa en la Figura 4.11 este tiene además de los propios datos de una carpeta, un campo calculado que indica si la carpeta esta compartida y otro campo que es una lista con los usuarios que esta compartida esa carpeta (si hay).

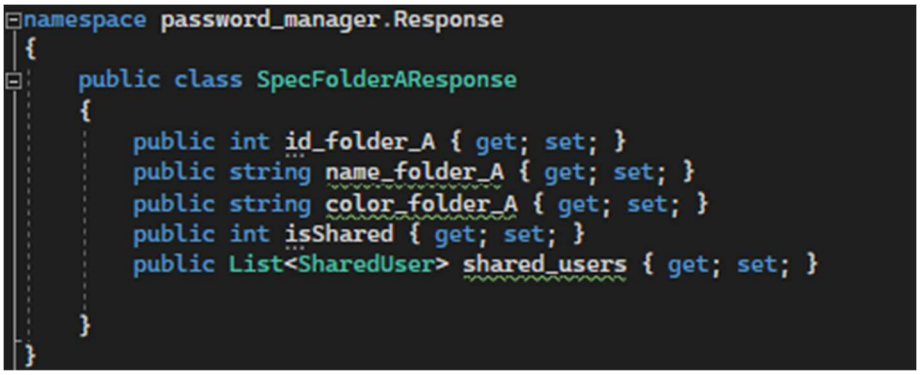


Figura 4.11: Contenido del fichero SpecFolderAResponse.cs del directorio Response.

4.4.3 Controladores

Un controlador es una clase que actúa como intermediario entre las solicitudes del cliente y la lógica de negocio de una aplicación web. La responsabilidad principal de los controladores de esta aplicación son recibir las solicitudes HTTP del Frontend (llamadas a APIs) y procesarlas para generar una respuesta adecuada. Esto implica llevar a cabo la lógica necesaria para manejar la solicitud y preparar los datos que serán utilizados para construir la respuesta, en formato JSON.

Un controlador tiene varias rutas que por lo general se agrupan bajo un mismo recurso. Cada ruta es una URL que se puede acceder y que acepta parámetros o texto en formato JSON en el cuerpo de la petición. En la aplicación las entidades usuarios capetas y cuentas tiene cada uno un controlador llamado "UsersController", "FolderAController" y "AccountController"

Cada controlador tiene la una colección de rutas que abarca las acciones: crear recurso, listar todos los recursos, obtener recurso, actualizar recurso, borrar recurso. También existen rutas especiales por ejemplo en el controlador "AccountController" el de obtener contraseña o actualizar contraseña. Todas las rutas permiten realizar las acciones necesarias para cumplir con los requisitos del gestor de contraseñas.

Debido a motivos de optimización de rendimiento (que se detallarán en el capítulo de especificaciones técnicas) el gestor de contraseñas desarrollado no permite compartir una contraseña en específico, sino que esta más orientado a compartir las carpetas. Este es un enfoque diferente ya que en vez de ir compartiendo una a una las contraseñas lo que se tiene que hacer es crear una carpeta específica para esa persona o ese grupo de personas y todas las contraseñas que se añadan a dicha carpeta serán visibles para todos los usuarios que tengas acceso.

En entornos corporativos es perfecto y aunque pudiera parecer una solución un poco tediosa para el usuario promedio no lo es en absoluto debido a dos razones importantes. La primera consta en que si un usuario ha compartido su contraseña con alguien es muy probable que comparta otras

contraseñas también ya que seguramente sean familia o amigos. La segunda es que una vez creado la carpeta para la/s persona/s en cuestión siempre va a tener que acceder a ella a través de esa carpeta y si por alguna razón cambia la contraseña y no quiere que los demás entren no se olvidará de que está compartida.

Existe un controlador específico y dedicado para compartir las carpetas llamado "ShareAccountController". Este presenta todas las rutas CRUD para que un usuario pueda acceder a una carpeta compartida, además de las rutas CRUD para acceder a la información de las carpetas compartidas y rutas especiales para compartir, dejar de compartir, acceder a la contraseña cifrada, cambiar contraseña cifrada, etc.

Como se está desarrollando una aplicación web cada respuesta de la API devolverá, además del contenido, un código de estado HTTP. En caso de éxito se devuelve el código 200. En caso de error se seguirán los códigos de estado convencionales cada uno usado adecuadamente para el propósito para el cual fue concebido. Los usados son: 400, 401, 403, 404, 204. Además, a veces el código 400 (petición errónea) es muy general. Por eso cuando sea necesario y se devuelva un 400 la respuesta llevará en el cuerpo un JSON con el siguiente formato: {error = '<frase>'} donde <frase> será una frase relevante para saber después en el Frontend qué mensaje de error mostrar al usuario. Por ejemplo, al registrarse el servidor puede devolver como repuesta el código 400 con el JSON {error = 'EMAIL_EXIST'}.

Para finalizar este apartado se muestra en la Figura 4.12 un controlador y se analiza su estructura ya que todos los controladores siguen la misma lógica.

```
64          [Authorize]
65          [HttpGet]
66          [Route("getFolderA/{id_folder_A}")]
67          public async Task<IActionResult> GetFolderA(int id_folder_A)
68          {
69
70              int id_user = int.Parse(HttpContext.User.FindFirstValue(ClaimTypes.NameIdentifier));
71              if (!_storage.ExistUserKey(id_user))
72              {
73                  return Unauthorized();
74              }
75              if (!await _folderAService.FolderAExistsForUser(id_user, id_folder_A))
76              {
77                  return StatusCode(403);
78              }
79              var folder = await _folderAService.GetFolderA(id_folder_A);
80              if (folder != null)
81              {
82                  return Ok(folder);
83              }
84              else
85              {
86                  return NotFound();
87              }
88          }
```

Figura 4.12: Líneas 64 - 88 del archivo FolderAController.cs. La ruta corresponde a la acción de obtener una carpeta en base a un id.

Al inicio en la línea 64 existen 3 cabeceras antes del método. La cabecera [Authorize] indica solo los usuarios que estén autentificados podrán acceder a este método. En caso contrario se devolverá el error "Unauthorized" automáticamente. En secciones posteriores se explicará el proceso de autentificación.

La cabecera [HttpGet] es un atributo en ASP.NET Core que se utiliza para decorar un método en un controlador de API para indicar que el método debe responder a las solicitudes HTTP GET. Cuando un cliente realiza una petición GET a una ruta que coincide con la ruta asociada al método decorado con [HttpGet], el método será invocado para manejar la solicitud.

Por último, la cabecera [Route] indica que el método se invocará cuando el cliente acceda a la ruta especificada. Además, esta ruta tiene un parámetro que hace referencia al id de la carpeta la cual se quiere acceder.

La línea 67 es la firma del método. Lleva la palabra reservada *"async"* que quiere decir que este método se ejecuta de forma asincrónica. Casi todos los métodos de todos los controladores son asincrónicos para proveer un mejor rendimiento a la aplicación. Los únicos métodos que no son asincrónicos son los que están en el controlador "UsersController.cs" que se encargan básicamente del registro de usuarios, cambio en el perfil de los usuarios y todas las operaciones que puede hacer un administrador. Esto se ha hecho así para evitar conflictos y condiciones de carrera en el servidor.

En la primera línea dentro del método (línea 70) se extrae del token de autentificación recibido por el usuario el id de usuario. Después, en líneas posteriores se comprueba que si la llave privada de este usuario está en la memoria del servidor significa que este usuario se ha autentificado y existe una sesión. Tener un token válido no es suficiente (después se explica el porqué). Si no es así se devolverá el código 401.

A continuación (línea 75), se utiliza un servicio propio también asíncrono para saber si ese usuario tiene acceso a la carpeta que quiere acceder. Si el método devuelve false, el usuario está intentando acceder a un recurso que no tiene permiso y le será devuelto el código 403 que quiere decir "prohibido".

Ahora que las comprobaciones estas hechas se ejecuta en la línea 79 la llamada al servicio que devolverá o no los datos requeridos. Como este también es un método asíncrono se espera con la palabra reservada *"await"*. Si hay datos para devolver el servicio devuelve una instancia del DTO "FolderAResponse" con los datos que se piden. En caso contrario el servicio devuelve un nulo y la API devolver el código 404.

Como se ha mencionado anteriormente todos los contralores siguen la misma estructura de tres pasos: comprobar si hay sesión vigente, comprobar que el usuario tiene permisos al recurso ejecutar la acción y devolver una respuesta.

4.4.4 Servicios propios

Para ayudar a la compatibilidad, seguir los principios de "Clean Arquitecture" y como buenas prácticas de programación se ha separado la lógica de la aplicación y el acceso a la base de datos en unos servicios propios. Así los controladores no acceden a la base de datos y sólo trabajan con los servicios propios (a excepción de controlador login) resultando en el flujo de información representado en la Figura 4.13.

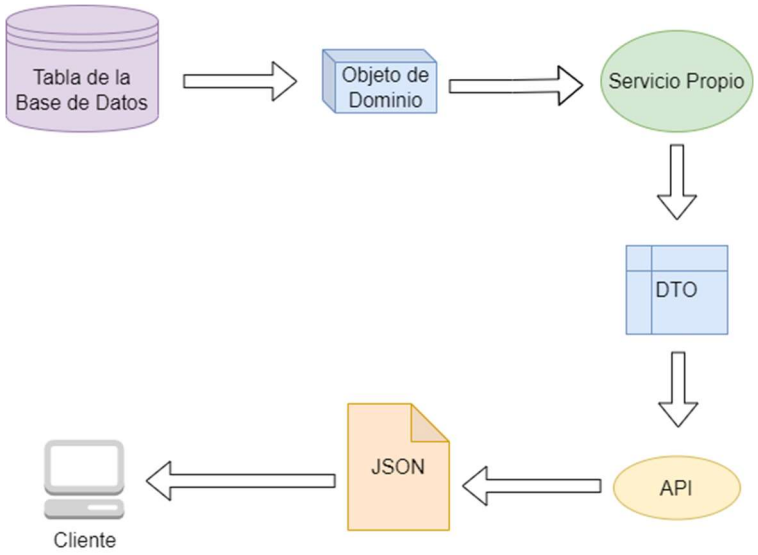


Figura 4.13: Flujo de información desde la base de datos al cliente.

Existe un servicio para cada controlador. Además, todos los métodos que usa el controlador "ShareAccountcontroller.cs" se encuentran en el servicio "AccountService" Estos servicios siguen la misma filosofía de inyección de dependencias que la clase "mydbContext" y también son asincrónicos. Cuando se necesita una instancia de una clase el motor de .NET la proporciona. Se configura como en la Figura 4.14.

```
builder.Services.AddScoped<UserService>();
builder.Services.AddScoped<FolderAService>();
builder.Services.AddScoped<AccountService>();
builder.Services.AddScoped<CryptoService>();
builder.Services.AddSingleton<UserStorage>();
```

Figura 4.14: Sección del archivo Program.cs. Tres Servicios para los controladores mencionados. Los dos últimos se detallarán en capítulos posteriores.

Cuando se utiliza AddScoped, el servicio estará disponible para toda la duración de la solicitud actual. Si hay varias dependencias que requieren el mismo servicio, todas recibirán la misma instancia durante el ciclo de vida de la solicitud.

Siguiendo con el ejemplo anterior se mostrará como un servicio provee a al método "GetFolderA" del controlador "FolderAController.cs" anterior la información necesaria. Para eso se muestran las Figuras 4.15 y 4.16.

```
public async Task<bool> FolderAExistsForUser(int id_user, int id_folder_A)
{
    var access_entry = await _dbContext.AccesControlAs
        .FirstOrDefaultAsync(a => a.IdUser == id_user && a.IdFolderA == id_folder_A && a.PermA == "o");
    return access_entry != null;
}
```

Figura 4.15: Método FolderAExistForUser del archivo FolderAService.cs. llamado por el controlador.

Este método accede a la base de datos y con una consulta LINQ verifica que existe en la tabla "acces_control_a" una entrada con el id del usuario, el id de la carpeta que se quiere acceder. Si

el atributo “perm_a” contiene una “o” significa que el usuario es propietario de la carpeta y se le da permiso. Si no es una “o” o no existe la entrada se deniega el acceso.

Esto es muy importante ya que desde el Frontend no se podría acceder a id’s de carpetas que no corresponden al usuario porque este no sabe cuáles son los id’s ni si existen ya que el Frontend sólo muestra carpetas a las que el usuario si debería tener acceso. Pero un atacante malicioso que esté autentificado puede enviar solicitudes con Postman u otro software hacia el Backend pidiendo información de carpetas y cuentas que no le pertenecen, adivinando el id o haciendo varias solicitudes de prueba y error. Por eso todos los servicios implementados tienen métodos similares para comprobar el acceso. Una vez que se le da acceso se procede a recabar información de la base de datos y expedirla.

```
public class FolderAService
{
    private readonly mydbContext _dbContext;
    private readonly UserStorage _storage;
    private readonly IConfiguration _config;
    private readonly IDataProtector _protector;
    private readonly CryptoService _crypto;

    public FolderAService(mydbContext dbcontext, IConfiguration config, IDataProtectionProvider dataProtectionProvider, UserStorage storage, CryptoService cs)
    {
        _dbContext = dbcontext;
        _storage = storage;
        _config = config;
        _protector = dataProtectionProvider.CreateProtector(_config.GetValue<string>("JwtConfig:ProtectorName"));
        _crypto = cs;

    }

    public async Task<bool> CreateFolderA(int id_user, CreateFolderARequest request)...

    public async Task<List<FolderAResponse>> GetFoldersForUser(int id_user)...



    public async Task<SpecFolderAResponse> GetFolderA(int id_folder_A)
    {
        try
        {
            var folder = await _dbContext.FolderAccounts
                .Where(fa => fa.IdFolderA == id_folder_A)
                .FirstOrDefaultAsync();

            if (folder == null)
                return null;

            var specFolderA = new SpecFolderAResponse
            {
                id_folder_A = folder.IdFolderA,
                name_folder_A = folder.NameFolderA,
                color_folder_A = folder.ColorFolderA,
                isShared = 0, // Default value is not shared
                shared_users = new List<SharedUser>()
            };

            var sharedUsers = await _dbContext.AccesControlAs
                .Where(aca => aca.IdFolderA == id_folder_A && aca.PermA != "o")
                .Join(_dbContext.Users,
                    aca => aca.IdUser,
                    u => u.IdUser,
                    (aca, u) => new { AccessControl = aca, User = u })
                .ToListAsync();

            if (sharedUsers.Count > 0)
            {
                specFolderA.isShared = 1;

                foreach (var sharedUser in sharedUsers)
                {
                    var sharedUserObj = new SharedUser
                    {
                        email_shared = sharedUser.User.Email,
                        perm = sharedUser.AccessControl.PermA,
                        exp_date = sharedUser.AccessControl.ExpDateA?.ToString("dd-MM-yyyy") ?? string.Empty
                    };

                    specFolderA.shared_users.Add(sharedUserObj);
                }
            }

            return specFolderA;
        }
        catch (Exception ex)
        {
            Console.WriteLine($"An error occurred while getting FolderA with id {id_folder_A}: {ex.Message}");
            return null;
        }
    }
```

Figura 4.16: Definición de clase y Método GetFolderA del archivo FolderAService.cs. llamado por el controlador.

En este ejemplo se usa la información de la clase de domino para llenar la información del DTO “SpecFolderAResponse”. Seguido se usa una consulta LINQ para saber si existe en la tabla “acces_control_a” alguna entrada de algún otro usuario con algún permiso que no sea el de propietario. Si hay, se modifica la variable “isShared” a verdadera (mediante el 1) y se itera sobre la lista para extraer la siguiente información: a quién se le ha compartido, con que permisos y hasta cuándo. Esta información va dentro de un DTO “SharedUser” que va dentro del DTO principal que se devulve al controlador.

Cuando un usuario comparte una carpeta con otro usuario y establece una fecha de expiración y esta caduca, el servidor tiene que borrar de la tabla “acces_control_a” la entrada asociada para quitar el permiso a ese usuario. Es necesario crear una tarea en el servidor (como se muestra en la Figura 4.17) para que se conecte a la base de datos y elimine estas entradas. Después hay que programar la tarea para que se ejecute periódicamente (Figura 4.18). Para lograr esto se ha usado la biblioteca Quartz instalado con el gestor de paquetes NuGet.

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Options;
using password_manager.Model;
using Quartz;

namespace password_manager.Services
{
    public class DenyAccessJob : IJob
    {
        private readonly mydbContext _dbContext;


        public DenyAccessJob(mydbContext dbcontext)
        {
            _dbContext = dbcontext;
        }
        public async Task Execute(IJobExecutionContext context)
        {
            Console.WriteLine("TAREA PROGRAMADA");
            DateOnly today = DateOnly.FromDateTime(DateTime.Today);

            var entriesToDelete = _dbContext.AccesControlAs
                .Where(entry => entry.ExpDateA < today)
                .ToList();

            _dbContext.AccesControlAs.RemoveRange(entriesToDelete);
            await _dbContext.SaveChangesAsync();
        }
    }
}
```

Figura 4.17: Fichero DenyAccessjob.cs Primero se obtiene la fecha de hoy. Segundo se ejecuta una consulta LINQ para obtener todas las fechas anteriores a hoy. Por último, se borran de la base de datos las entradas encontradas.

```
builder.Services.AddQuartz(q =>
{
    q.UseMicrosoftDependencyInjectionJobFactory();

    var jobKey = new JobKey("DenyAccessJob");

    q.AddJob<DenyAccessJob>(opts => opts.WithIdentity(jobKey));

    q.AddTrigger(opts => opts
    .ForJob(jobKey)
    .WithIdentity($"{jobKey}-trigger")
    .WithCronSchedule("0 1 0 * * ?"));
});

builder.Services.AddQuartzHostedService(q => q.WaitForJobsToComplete = true);
```

Figura 4.18: Configuración simple de Quartz mediante cronSchedule para ejecutrase (de izquierda a derecha) a los 0 segundos, al minuto 1, a la hora 0 de cualquier día de cualquier mes, no importa el día de la semana. En resumen, se ejecuta todos los días a las 00:01.

4.4.5 Autentificación

Para este proyecto se ha escogido la autenticación por tokens JWT. [13] Los tokens JWT (JSON Web Tokens) son una forma compacta y segura de representar información entre dos partes en forma de objetos JSON. Estos tokens son utilizados para autenticar y autorizar usuarios en aplicaciones web y servicios API. La estructura básica de los tokens JWT que se van a utilizar consta de tres partes separadas por puntos: la cabecera (header), el cuerpo (payload) y la firma (signature).

a) **Cabecera:** Contiene información sobre el tipo de token y el algoritmo de firma utilizado. Es un objeto JSON codificado en Base64url que define cómo se debe procesar el token.

b) **Cuerpo:** Contiene los datos que se van a transmitir: el identificador del usuario, el email de usuario, el remitente del token, el destinatario y una fecha y hora de caducidad que se ha establecido en 16 horas. Pasadas estas horas el token ya no será válido. También es un objeto JSON codificado en Base64url.

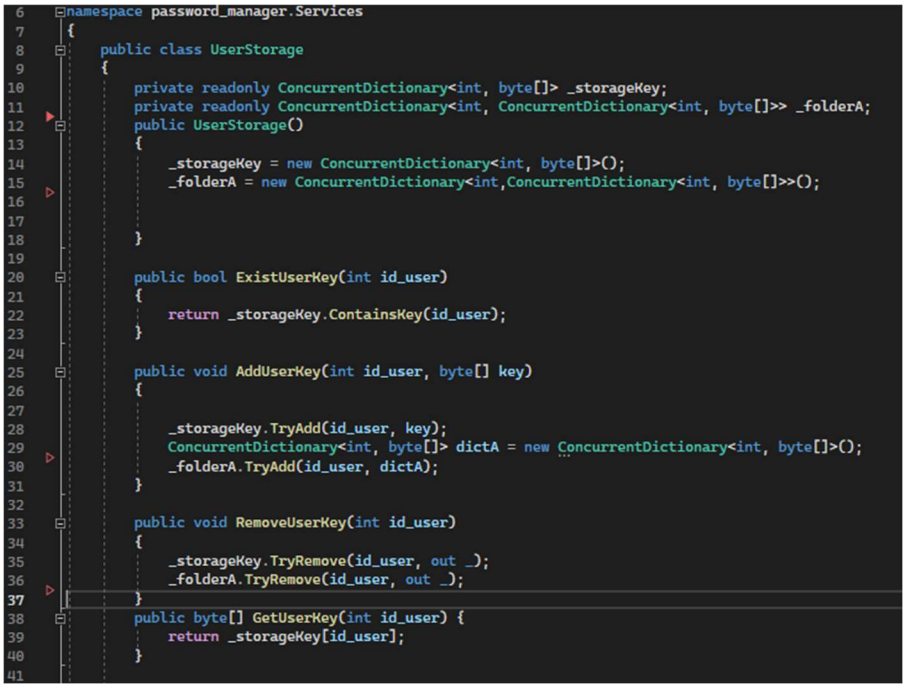
c) **Firma:** Es la parte final del token y se utiliza para verificar la integridad y autenticidad del mensaje. La firma se crea combinando la cabecera y el cuerpo con una clave secreta conocida solo por el servidor, utilizando el algoritmo de firma digital, HMAC (Hash-based Message Authentication Code). De esta forma el servidor puede confiar en que nadie ha modificado el token cambiando por ejemplo el id de usuario para hacerse pasar por otra persona. Si alguno de los campos cambia el hash cambiará y la firma ya no será válida.

El token JWT es autónomo y autocontenido, lo que significa que toda la información necesaria para validar y procesar el token se encuentra dentro de él. Eso significa que el servidor no recuerda los tokens que ha autorizado, sólo los sabe validar. Esto crea un problema y es que el usuario cuando hace “logout” lo único que pasa es que deshecha ese token y se olvida de él. Pero si un atacante malicioso consiguiera acceso a la máquina del usuario después de que este la usara y se hiciera con el token podría hacerse pasar por él.

Por eso es va a optar guardar en el servidor la información sobre la sesión del usuario. La forma tradicional es guardar en un HashMap de los tokens rechazados o los tokens emitidos. Cuando el usuario cierra la sesión se guarda el token en la lista de tokens rechazados o se quita de la lista de tokens emitidos.

Parece correcto, pero en esta aplicación se va a buscar un enfoque diferente y más ingenioso. Como además el servidor debe tener en memoria la clave privada del usuario para poder descifrar los datos de la base de datos lo que se ha hecho es crear un diccionario con el par (clave, valor): (id de usuario, clave privada).

Así cuando un usuario se autentifica se guarda en el diccionario su clave privada y cuando cierra su sesión se borra su clave. Este diccionario forma parte del servicio “UserStorage” que expone métodos públicos de acceso. Se muestra la clase en la Figura 4.19 y el registro del servicio en la figura 4.20



```
namespace password_manager.Services
{
    public class UserStorage
    {
        private readonly ConcurrentDictionary<int, byte[]> _storageKey;
        private readonly ConcurrentDictionary<int, ConcurrentDictionary<int, byte[]>> _folderA;
        public UserStorage()
        {
            _storageKey = new ConcurrentDictionary<int, byte[]>();
            _folderA = new ConcurrentDictionary<int,ConcurrentDictionary<int, byte[]>>();


        }

        public bool ExistUserKey(int id_user)
        {
            return _storageKey.ContainsKey(id_user);
        }

        public void AddUserKey(int id_user, byte[] key)
        {

            _storageKey.TryAdd(id_user, key);
            ConcurrentDictionary<int, byte[]> dictA = new ConcurrentDictionary<int, byte[]>();
            _folderA.TryAdd(id_user, dictA);
        }

        public void RemoveUserKey(int id_user)
        {
            _storageKey.TryRemove(id_user, out _);
            _folderA.TryRemove(id_user, out _);
        }
        public byte[] GetUserKey(int id_user) {
            return _storageKey[id_user];
        }

```

Figura 4.19: Parte del archivo UserStorage.cs del directorio Services. Muestra el diccionario storageKey, el constructor y los métodos públicos ExistUserKey, AddUserKey, RemoveUserKey y GetUserKey. El otro diccionario es con fines de optimización y se detalla en siguiente capitulo.

```
builder.Services.AddSingleton<UserStorage>();
```

Figura 4.20: Se registra en el archivo Program.cs como AddSingleton porque se desea una sola instancia del servicio en todo el ciclo de vida de la aplicación. Así solo existe un único diccionario de claves al que comprobar si un usuario tiene una sesión activa.

Ahora que se ha especificado cómo el servidor mantiene la sesión del usuario se muestra cómo se configura el servicio de autenticación (Figura 4.21 y 4.22). Este comprobará que en las cabeceras de las solicitudes exista un token JWT válido.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        /*byte[] keyBytes = new byte[16];
        byte[] protectedKeyBytes = new byte[16];
        using (var serviceProvider = builder.Services.BuildServiceProvider())
        {
            var dataProtectionProvider = serviceProvider.GetService<IDataProtectionProvider>();
            var protector = dataProtectionProvider.CreateProtector(builder.Configuration["JwtConfig:ProtectorName"]);
            using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
            {
                rng.GetBytes(keyBytes);
            }
            protectedKeyBytes = protector.Protect(keyBytes);
            SealedVariables.Instance.InitializeJWT(protectedKeyBytes);
        }*/
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,

            ValidIssuer = builder.Configuration["JwtConfig:Issuer"],
            ValidAudience = builder.Configuration["JwtConfig:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(builder.Configuration["JwtConfig:Secret"]))
            //IssuerSigningKey = new SymmetricSecurityKey(keyBytes),
        };/*
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(keyBytes);
            rng.GetBytes(keyBytes);
        }*/

    });
```

Figura 4.21: Se configura para comprobar todos los campos y que el emisor, el receptor y la clave con la que se firmó sean idénticas los valores del objeto "JwtConfig" contenido en el archivo de configuración appSettings.json.

```
"JwtConfig": {
  "Secret": "mysecretkeyissosososososososososecret",
  "Issuer": "password-manager-server-v1",
  "Audience": "password-manager-client",
  "AccessTokenExpirationInMinutes": 960,
  "ProtectorName": "ProtectorTokenInstance"
}
```

Figura 4.22: Objeto JwtConfig del archivo de configuración appSettings.json

Ahora cuando el usuario se autentifica el servidor tiene que crear un token válido para expedirlo. En la Figura 4.23 se muestra este proceso.

```
    var tokenHandler = new JwtSecurityTokenHandler();
    var tokenDescriptor = new SecurityTokenDescriptor
    {

        Subject = new ClaimsIdentity(new Claim[]
        {
            new Claim(ClaimTypes.NameIdentifier, user.IdUser.ToString()),
            new Claim(ClaimTypes.Email, user.Email)
        }),
        Audience = _config.GetValue<string>("JwtConfig:Audience"),
        Issuer = _config.GetValue<string>("JwtConfig:Issuer"),
        Expires = DateTime.UtcNow.AddMinutes(_config.GetValue<double>("JwtConfig:AccessTokenExpirationInMinutes")),
        //SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(_protector.Unprotect(SealedVariables.Instance.GetImmutableArray())), SecurityAlgorithms.HmacSha256Signature)
        SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(Encoding.UTF8.GetBytes(_config.GetValue<string>("JwtConfig:Secret"))), SecurityAlgorithms.HmacSha256Signature)

};
    var token = tokenHandler.CreateToken(tokenDescriptor);

    return Ok(new
    {
        access_token = tokenHandler.WriteToken(token)
    });
}
catch (Exception ex)
{
    var errorResponse = new { error = "2" };
    return StatusCode(500, errorResponse);
}
```

Figura 4.23: Parte del código del archivo LoginController.cs

4.4.6 Testing

Para probar todas las rutas y comprobar que el Backend funciona como es debido se ha usado "Swagger". Esta es una herramienta de código abierto para describir, documentar y visualizar APIs RESTful. Swagger utiliza la especificación OpenAPI para describir los endpoints, los parámetros, las respuestas y otra información relevante de la API al mismo tiempo que ofrece una interfaz visual para enviar peticiones a las rutas con plantillas json especificas y mostrar las respuestas. Se muestra un ejemplo de esta herramienta en acción en la Figura 4.24.

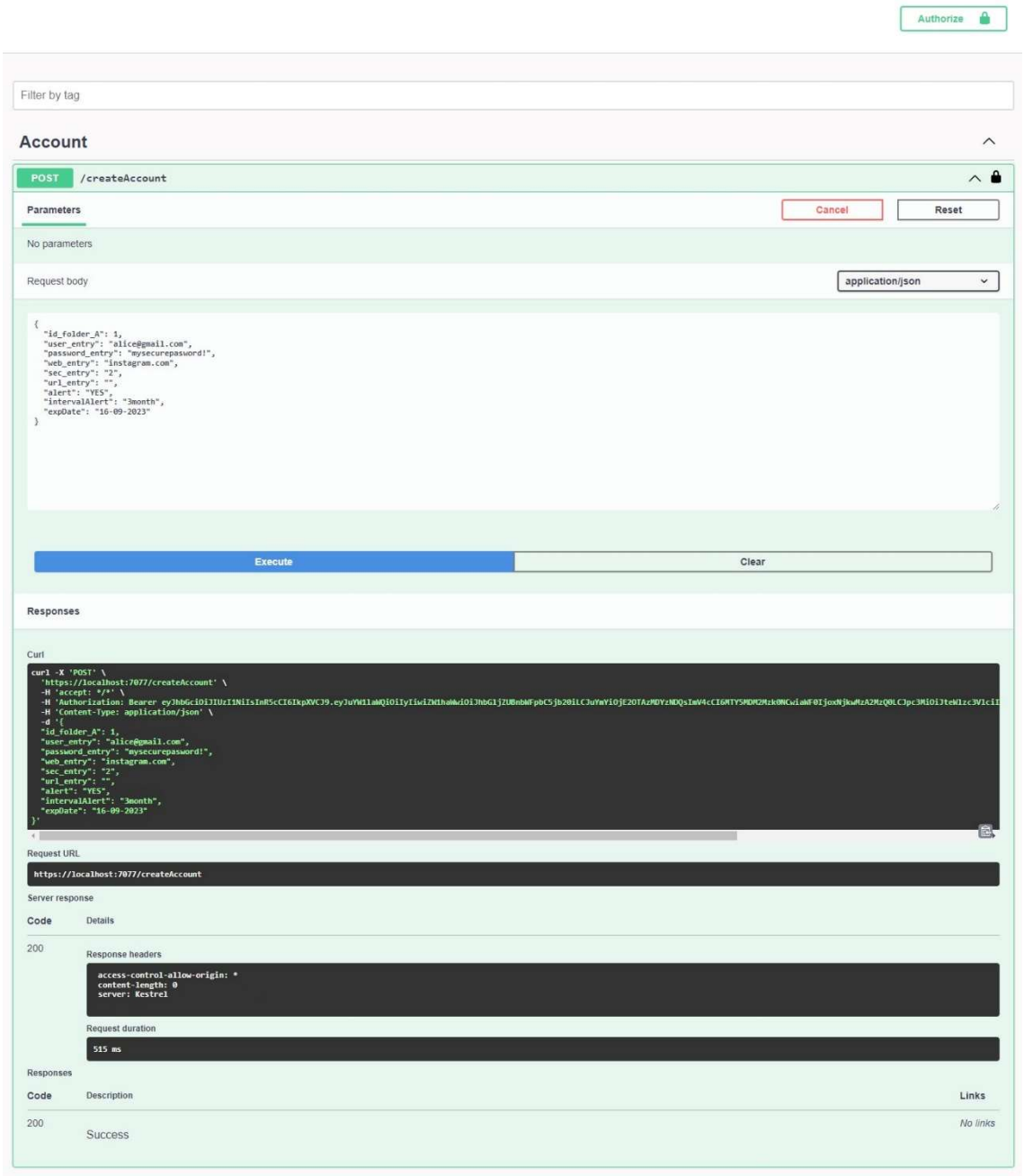
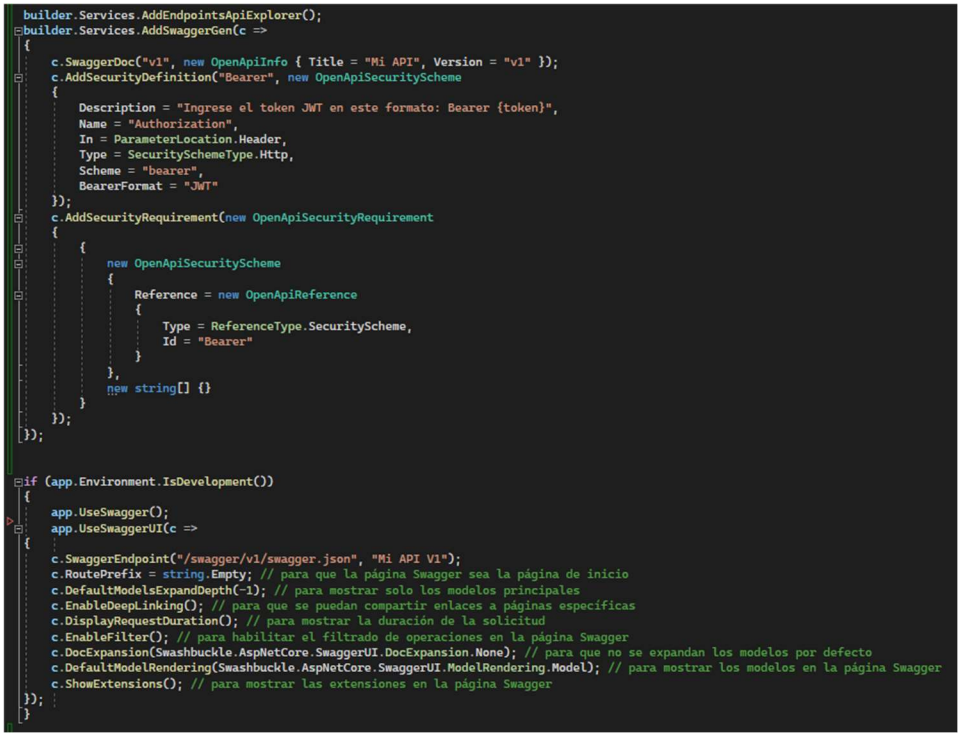


Figura 4.24 Ejemplo de creación de cuenta desde Swagger.

Para poder tener el botón de "Authorize" y mandar el token de autentificación con cada petición ha sido necesaria una configuración adicional que se muestra en la Figura 4.25.



```
builder.Services.AddEndpointsApiExplorer();
builder.Services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo { Title = "Mi API", Version = "v1" });
    c.AddSecurityDefinition("Bearer", new OpenApiSecurityScheme
    {
        Description = "Ingrese el token JWT en este formato: Bearer {token}",
        Name = "Authorization",
        In = ParameterLocation.Header,
        Type = SecuritySchemeType.Http,
        Scheme = "bearer",
        BearerFormat = "JWT"
    });
    c.AddSecurityRequirement(new OpenApiSecurityRequirement
    {
        {
            new OpenApiSecurityScheme
            {
                Reference = new OpenApiReference
                {
                    Type = ReferenceType.SecurityScheme,
                    Id = "Bearer"
                }
            },
            new string[] {}
        }
    });
});

if (app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI(c =>
{
    c.SwaggerEndpoint("/swagger/v1/swagger.json", "Mi API V1");
    c.RoutePrefix = string.Empty; // para que la página Swagger sea la página de inicio
    c.DefaultModelsExpandDepth(-1); // para mostrar solo los modelos principales
    c.EnableDeepLinking(); // para que se puedan compartir enlaces a páginas específicas
    c.DisplayRequestDuration(); // para mostrar la duración de la solicitud
    c.EnableFilter(); // para habilitar el filtrado de operaciones en la página Swagger
    c.DocExpansion(Swashbuckle.AspNetCore.SwaggerUI.DocExpansion.None); // para que no se expandan los modelos por defecto
    c.DefaultModelRendering(Swashbuckle.AspNetCore.SwaggerUI.ModelRendering.Model); // para mostrar los modelos en la página Swagger
    c.ShowExtensions(); // para mostrar las extensiones en la página Swagger
});
}
```

Figura4.25: Configuración del UI de Swagger en el archivo Program.cs

4.5 Frontend

Antes de desarrollar el Frontend se va a establecer el tipo general de aplicación web: SPA o MPA.

Una Single-Page Application (SPA) es un tipo de aplicación web que carga una única página HTML y luego actualiza el contenido de forma dinámica, sin requerir una carga completa de la página en cada interacción del usuario. Esto se logra mediante el uso de JavaScript para manejar las interacciones y la manipulación del DOM. Las SPAs ofrecen una experiencia más fluida y rápida al usuario, ya que sólo necesitan cargar los recursos necesarios una vez y luego actualizan la interfaz de usuario de manera dinámica, reduciendo los tiempos de espera y mejorando la interactividad. Algunos ejemplos de SPAs populares incluyen Gmail y Facebook.

En contraste, una Multi-Page Application (MPA) es una aplicación web que consta de múltiples páginas HTML, cada una con su propia URL y contenido. Cada vez que el usuario interactúa con la aplicación y navega a una nueva página, se realiza una solicitud al servidor para obtener la nueva página. A diferencia de las SPAs, las MPAs requieren una carga completa de la página cada vez que se realiza una nueva solicitud, lo que puede llevar a tiempos de espera más largos y una experiencia menos fluida para el usuario. Sin embargo, las MPAs son más sencillas de desarrollar y pueden ser más adecuadas para aplicaciones con contenido más estático o estructurado en páginas independientes. Ejemplos de MPAs incluyen Wikipedia y Reddit.

Una vez definidos los términos la opción más evidente es desarrollar una SPA ya que estar cargando páginas separadas para una contraseña no es lo más adecuado y entorpecería mucho la experiencia de usuario. Además de que casi todas las páginas web actualmente que ofrecen servicios en línea como redes sociales, correo electrónico, plataformas de música y video,

aplicaciones bancarias y financieras, plataformas de comercio electrónico, herramientas de productividad, plataformas de aprendizaje en línea, etc. son SPA.

4.5.1 Tecnología utilizada

Actualmente todas las páginas se programan con JavaScript o TypeScript. Estos lenguajes son los más utilizados en el desarrollo web Frontend debido a su capacidad para manipular el DOM (Modelo de Objetos del Documento) y gestionar eventos de forma eficiente, lo que es esencial para crear una experiencia de usuario fluida e interactiva en una SPA.

Existen varios frameworks para desarrollo Frontend. En la Figura 4.26 se muestra una estadística de tecnologías más usadas para este propósito.

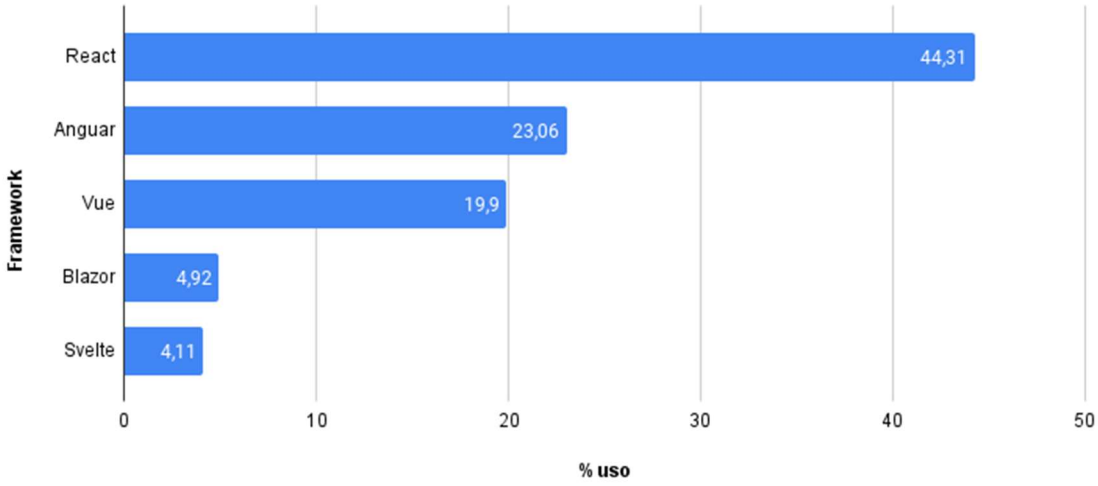


Figura 4.26: Frameworks Frontend más usados por desarrolladores profesionales. Datos extraídos de la encuesta anual de Stackoverflow Insights 2022 con un total de 45 297 respuestas [14]

Como se observa React es la tecnología mayormente utilizada y es escogida para desarrollar esta aplicación. Es una biblioteca JavaScript de código abierto que se utiliza para construir interfaces de usuario interactivas y dinámicas. Fue desarrollada por Facebook y se ha convertido en una de las tecnologías más populares para el desarrollo Frontend.

El funcionamiento de React se basa en la creación de componentes reutilizables que representan partes individuales de la interfaz. Estos componentes están escritos en JavaScript y pueden contener código HTML (JSX) para definir la estructura de la interfaz. React utiliza un modelo de programación declarativo, lo que significa que los desarrolladores sólo deben preocuparse por la lógica y el estado de los componentes, mientras que React se encarga de la actualización eficiente de la interfaz en función de los cambios de estado.

Como herramienta de creación de proyecto se utiliza Vite. Se utiliza para desarrollar aplicaciones web modernas y rápidas, centrándose especialmente en proyectos basados en Vue.js, aunque también es compatible con otras bibliotecas como React. Proporciona un entorno de desarrollo de alto rendimiento con las siguientes características principales:

a) **Tiempo de compilación rápido:** Vite utiliza el esquema de **"**importación bajo demanda**"** para lograr un tiempo de compilación rápido, lo que permite una experiencia de desarrollo en tiempo real mientras se edita el código.

b) **Servidor de desarrollo con recarga rápida:** Vite incluye un servidor de desarrollo con recarga rápida (hot module replacement), lo que significa que los cambios

realizados en el código se reflejan instantáneamente en el navegador sin necesidad de recargar la página.

c) **Enfoque centrado en los módulos:** Vite se basa en el sistema de módulos nativo de JavaScript (ES modules) para administrar las dependencias y proporcionar una experiencia de desarrollo más eficiente.

d) **Construcción optimizada para producción:** Vite es capaz de generar paquetes de producción altamente optimizados, lo que resulta en tiempos de carga más rápidos y una mejor experiencia para los usuarios.

4.5.2 Páginas y enrutado

Como se ha mencionado el gestor de contraseñas será SPA, pero esto no quiere decir que sólo tendrá una página. De hecho, es una buena práctica separar las páginas por propósitos del usuario y grupos de acciones. Así pues, las páginas que tiene el gestor de contraseñas son las siguientes:

Una página sencilla de portada que aparece nada mas acceder el dominio con información introductoria sobre el gestor de contraseñas. Desde esta página se puede acceder a la página de registro o a la página de autentificación (llamada de ahora en adelante login).

La página de registro donde el usuario introduce sus datos y se completa el registro. Debido a las políticas de privacidad no será necesario verificar el correo ya que no tiene que ser uno real. Esto ofrece anonimato en internet, algo difícil de encontrar últimamente para un usuario común, además de ofrecer la posibilidad de cambiar el dominio del email y tener una mejor organización, aunque no se dispongan de correos con dominio personalizado.

La otra página que se puede acceder desde la página de portada es la de login. Esta es también muy simple donde el usuario sólo tiene que introducir sus credenciales. Después de página el usuario será redirigido a la pagina del gestor de contraseñas propiamente dicho.

La siguiente página es la del gestor de contraseñas. Tiene un botón en la parte superior para acceder a la página de cuenta de usuario. El resto de la página se describirá en la siguiente sección junto con los componentes.

La última es la página de cuenta de usuario donde este puede consultar su información personal, cambiarla, darse de baja o hacer logout.

Como todo es un componente en React las páginas también los son, pero hay que asociarlos con una ruta. En la figura 4.27 se muestra cómo se consigue esto.

```
15  function App() {
16    return (
17      <Router>
18        <Routes>
19          <Route path="/" element={<Welcome />} />
20          <Route path="/login" element={<LoginPage />} />
21          <Route path="/register" element={<RegisterPage />} />
22          <Route path="/accounts" element={<AccountsPage />} />
23          <Route path="/user" element={<UserPage />} />
24        </Routes>
25      </Router>
26    );
27  }
28
29  export default App
```

Figura 4.27: Contenido del archivo App.jsx excepto los imports.

Si en alguna parte de la aplicación se necesita redirigir a una página en base a unas condiciones se puede hacer con el hook de navegación proporcionado por la librería React Router. Por ejemplo, en la Figura 4.28 se muestra cómo se redirige a login si el servidor responde que el usuario no está autentificado (código de estado 401).

```
} else if (response.status === 401) {
  sessionStorage.removeItem('accessToken');
  navigate('/login');
```

Figura 4.28: Contenido del archivo FolderPanel.jsx.

4.5.3 Diseño y componentes

Antes de explicar el diseño el estudiante reconoce que carece de conocimientos de UI/UX. Su formación y su itinerario no le han proporcionado habilidades para diseñar y realizar un trabajo en condiciones y es por eso por lo que el enfoque más simple que se va a adoptar es no realizar este diseño. Es consiente de que esta parte no se puede obviar, pero en el mundo laboral no será el estudiante quien tenga esta responsabilidad.

Para llegar al resultado final este se ha inspirado de su experiencia con otros gestores de contraseñas otras aplicaciones web y el propio sentido común y los elementos se han ido colocando a medida que se iba desarrollando el Frontend cambiando los atributos según la retroalimentación que iba recibiendo de su entorno.

Como el gestor es una herramienta de productividad se ha copiado en la distribución de espacio de servicios de email como Outlook o Protonmail. El concepto de tener toda la información visible en una sola página es útil y poderoso y es una estrategia que siguen muchas aplicaciones web. Esto se consigue anidando componentes como muestra el diseño de la Figura 4.29.

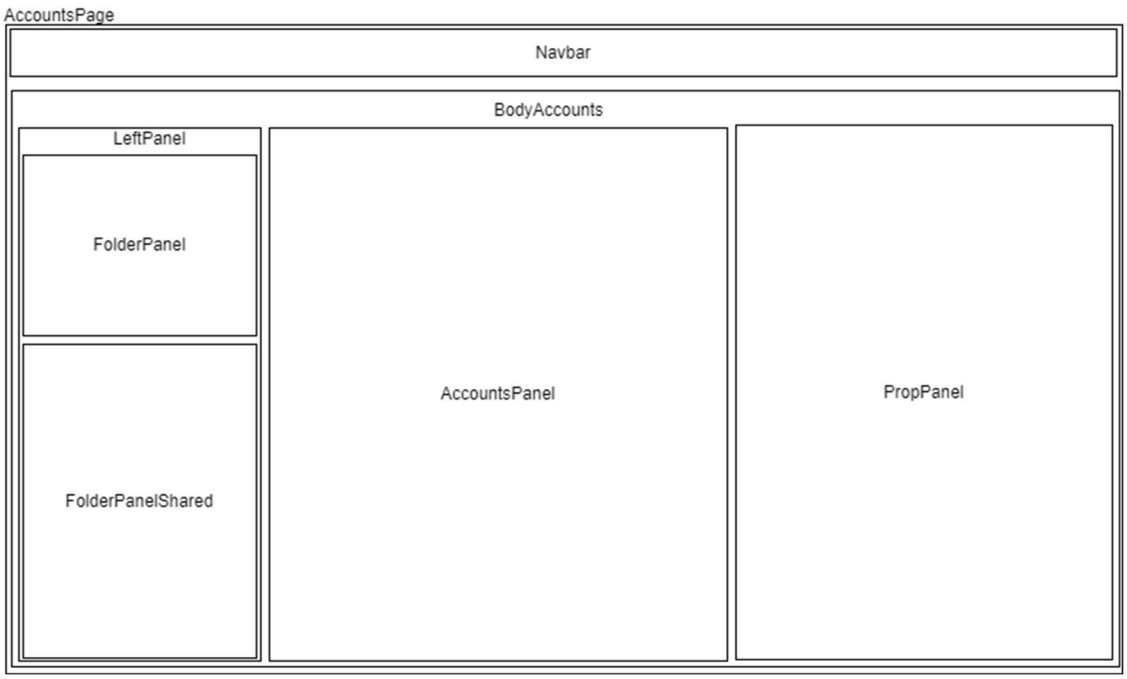


Figura 4.29: Diseño y disposición de componentes principales del gestor de contraseñas.

El usuario ve una lista desplegable con todas sus carpetas en el componente “FolderPanel” y además otra lista con las carpetas que le han compartido a él en el componente “FolderPanelShared”. Donde es posible seleccionarlas.

En el componente del medio “AccountPanel” se muestra una lista con todas las cuentas que hay en la carpeta seleccionada.

El componente de la derecha es el panel de propiedades. Desde aquí se realizan todas las operaciones de creación y modificación de carpetas y cuentas incluso el proceso de compartir carpetas. Dentro de este componente se renderiza dinámicamente otros componentes en función de los botones que se hayan pulsado o en función del ítem (carpeta o cuenta) que esté seleccionado. El funcionamiento se describe en las secciones posteriores.

4.5.3 Obtención de Datos

Como se ha explicado con anterioridad la API devuelve un JSON. Para obtener ese JSON, extraer los datos y mostrarlos cuando se cargan los componentes, se utiliza un useEffect. El hook “useEffect” se utiliza para ejecutar código en respuesta a cambios en el componente, como cuando se monta (se agrega al DOM), se actualiza o se desmonta. Esto permite realizar tareas adicionales de forma controlada y evitar problemas de rendimiento y de sincronización. El “useEffect” toma dos argumentos: una función y una matriz de dependencias.

En la aplicación la función representa la llamada a la API, y se ejecuta cada vez que el componente se monta, se actualiza o se desmonta.

La matriz de dependencias es una serie de valores que, al cambiar, el efecto se ejecuta nuevamente.

En la Figura 4.30 se muestra el código que obtiene una lista las carpetas del usuario y las guarda en una variable folders.

```
const [folders, setFolders] = useState([]);
const [isEmpty, setIsEmpty] = useState(true);
const navigate = useNavigate();

useEffect(() => {
  const fetchData = async () => {
    try {
      const accessToken = sessionStorage.getItem('accessToken');
      if (!accessToken){
        navigate('/login');
        return;
      }
        const response = await fetch(`${API_URL}/listFoldersA`, {
        headers: {
          Authorization: `Bearer ${accessToken}`,
        },
      });

      if (response.status === 200) {
        const data = await response.json();
        data.sort((a, b) => a.name_folder_A.localeCompare(b.name_folder_A));
        setFolders(data);
        setIsEmpty(false);
      } else if (response.status === 204) {
        setFolders([]);
        setIsEmpty(true);
      } else if (response.status === 401) {
        sessionStorage.removeItem('accessToken');
        navigate('/login');
      } else {
        console.log('Error FolderPanel Lista');
      }
    } catch (error) {
      console.log('Error de red', error);

    }
  };

  fetchData();
```

Figura 4.30: Contenido del archivo FolderPanel.jsx. Con "fetch" se realiza la llamada a la API y se comprueba el código de estado HTP recibido. Antes de guardar los datos en la variable local folders se hace una ordenación por nombre de carpeta para que todas ellas se muestren siempre en la misma posición.

4.5.4 Gestión de Estado

Para tener una aplicación que actualiza contenido dinámicamente haciendo peticiones a la API en segundo plano es necesario saber si los componentes que ve y utiliza el usuario han cambiado. Para eso se utiliza el hook useState de React. Este es un hook que permite a los componentes de función tener su propio estado local.

Al invocarlo se obtienen un par de valores: el estado actual y una función para actualizar ese estado. Al declarar el estado con useState, se proporciona un valor inicial. Luego, se puede usar el estado y la función de actualización dentro del componente para manejar el estado de manera reactiva.

Cuando la función de actualización se llama con un nuevo valor, React re-renderiza el componente con el nuevo estado y actualiza la interfaz de usuario en consecuencia. Esta característica permite desarrollar componentes interactivos y dinámicos en React, ya que puedes manejar el estado local sin la necesidad de crear una clase.

Si se combina estos valores con la matriz de dependencias del useEffect se consigue una herramienta poderosa para mostrar la información dinámicamente. Además, el estado y la función de actualización se pueden pasar a los componentes hijos para que estos sean quienes actualicen el estado y cambiar así con una variable un componente distinto. Para conseguir la funcionalidad

que se desea y que los componentes se rendericen a tiempo real se muestra el esquema de la figura 4.31.

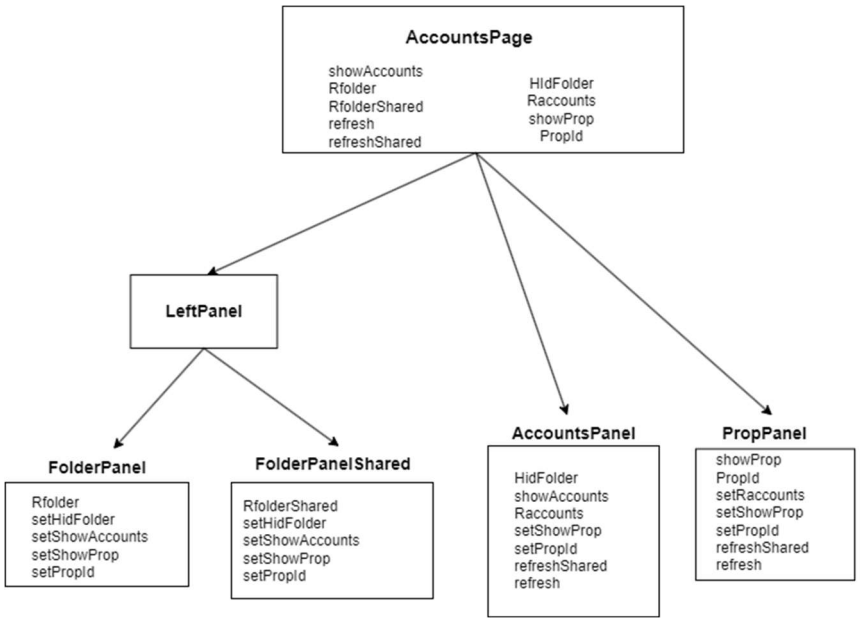


Figura 4.31: Diagrama de herencia. El padre AccountsPage define los hooks. En la figura no se muestran las funciones de actualización por una cuestión de limpieza. Refresh y refreshShared no son hooks, son funciones que alternan de true a false los valores de Rfolder y RfolderShared respectivamente. Los hijos reciben los hooks del padre mediante paso de propiedades. Los hooks heredados están en la matriz de dependencia del componente hijo para poder ser actualizado a placer.

"showAccounts" y "HidFolder" los verifican AccountsPanel. Puede tomar el valor "none", "my" o "shared" en función de si no hay cuentas para mostrar, si se muestran las cuentas propias o las compartidas respectivamente. Al pulsar sobre una carpeta se cambia este valor en función de si se ha pulseado en una carpeta propia o compartida. También se actualiza el id de "HidFolder" para que AccountsPanel sepa las cuentas de qué carpeta mostrar.

Si mientras se está navegando sobre una carpeta el usuario pierde el derecho a ella la API contestará con un 403 y se tendrá que hacer un refresh de AccountsPanel, pero también de FolderPanelShared. Por eso se utilizará la función "refreshShared" para actualizar también el componente FolderPanelShared.

Cada carpeta tiene un botón para mostrar los ajustes de la carpeta. Lo mismo pasa con las cuentas. Además, hay un botón para añadir carpetas y añadir cuentas. Todas estas acciones cambian el "showProp" a una cadena con la acción asociada. También en los casos en los que hace falta (mostrar las propiedades de la carpeta tal) se actualiza el valor de "PropId" con el id de la carpeta o la cuenta. Después se filtra el showProp para saber que componente hijo montar en este componente. El componente hijo realiza acciones en la API y es necesario que los otros componentes se actualicen. Si se necesita actualizar FolderPanel (porque se ha borrado una carpeta, por ejemplo) o SharedPanel se usan las funciones refresh y refreshShared respectivamente. Si se necesita actualizar AccountsPanel (porque se ha borrado una cuenta, por ejemplo) se usa "setRaccounts"

De esta forma y gracias a la matriz de dependencia del useEffect y al hook useState, todos los componentes se pueden actualizar entre sí.

5. Detalles de implementación

Este capítulo trata del cifrado de la aplicación y de las técnicas que ofrecen seguridad y rendimiento. El algoritmo de cifrado simétrico escogido es AES-256 CBC con padding PKCS7. [15] Este es considerado seguro debido a su robustez matemática y a su diseño resistente a varios ataques criptográficos conocidos. Utiliza operaciones matemáticas bien establecidas y estudiadas, como sustituciones no lineales y mezclas de columnas y filas, que han demostrado ser altamente resistentes a intentos de descifrado no autorizados. Cuanto más larga es la clave, mayor es la resistencia frente a ataques de fuerza bruta, ya que aumenta exponencialmente el número de posibles combinaciones de claves que un atacante tendría que probar. Por eso AES con longitud de clave de 256 bits se considera un estándar seguro.

CBC es un modo de operación que añade una capa de seguridad adicional para cifrar bloques de datos. En este modo, cada bloque de datos se cifra usando la clave y el bloque anterior de datos cifrados. Esto ayuda a eliminar patrones repetitivos en el cifrado y proporciona una mayor seguridad en comparación con modos de operación más simples.

El relleno PKCS#7 (Padding Scheme #7 de los Public Key Cryptography Standards) es una técnica criptográfica utilizada para ajustar la longitud de datos antes de ser cifrados. En criptografía simétrica, como el cifrado AES, los datos que se van a cifrar deben tener una longitud que sea un múltiplo del tamaño del bloque de cifrado utilizado (por ejemplo, 128 bits para AES-128 o 256 bits para AES-256). Sin embargo, en muchos casos, los datos que se desean cifrar no tienen una longitud que sea un múltiplo exacto del tamaño del bloque de cifrado.

El relleno PKCS#7 resuelve este problema agregando bytes adicionales al final de los datos para que la longitud total sea un múltiplo del tamaño del bloque de cifrado. Estos bytes adicionales tienen un valor que indica cuántos bytes se han agregado, lo que permite que el receptor sepa cuántos bytes deben eliminarse después de descifrar los datos para recuperar los datos originales sin el relleno.

A continuación, cuando se hable de “cifrarAES” o “descifrarAES” se refiere a cifrar/descifrar con clave simétrica AES-256 CBC y padding PKCS7 a menos que se especifique lo contrario.

El algoritmo que se usa para cifrado asimétrico es RSA (Rivest-Shamir-Adleman). Este es un algoritmo de cifrado asimétrico ampliamente utilizado en criptografía. Fue propuesto por Ron Rivest, Adi Shamir y Leonard Adleman en 1977 y se basa en la dificultad de factorizar grandes números enteros en sus números primos componentes.

A continuación, cuando se hable de “cifrarRSA” o “descifrarRSA” se refiere a cifrar/descifrar con clave asimétrica RSA a menos que se especifique lo contrario.

Otra técnica que es necesario conocer antes de explicar el funcionamiento del cifrado es el proceso de derivación de clave. Esto es una técnica criptográfica utilizada para generar una clave secreta o un conjunto de claves a partir de una entrada inicial, como una contraseña o una clave maestra. En esta aplicación se usa PBKDF2 ya que es más rápida y usa menos memoria que otros algoritmos como Scrypt o Argon2.

La derivación de clave con PBKDF2 se realiza mediante el uso de una función de hash criptográfico (en la aplicación HMAC). Cada paso consiste en mezclar la contraseña con un valor aleatorio llamado "salt" y aplicar la función de hash. El resultado pasa a ser la entrada del paso siguiente donde se volverá a mezclar con el salt y se volverá a aplicar la función de hash. Esto se realiza un numero grande de veces. El número de iteraciones se puede configurar para hacer que el proceso sea más lento y costoso, dificultando así los intentos de descifrar la contraseña mediante ataques de fuerza bruta o diccionario. Al final de las iteraciones, se obtiene una clave derivada que es más segura y adecuada para su uso en algoritmos de cifrado.

A continuación, cuando es hable de derivar la clave se hará mediante el algoritmo PBKDF2 con un salt especificado, hash HMAC y con 1000 iteraciones.

5.1 Cifrado de base de datos

En este gestor de contraseñas en la versión 1 se cifra sólo la contraseña de cada cuenta. Un primer enfoque podría ser cifrarAES el campo “password_entry” de la tabla de “accounts” (véase figura 4.2) la contraseña maestra del usuario. Así nadie excepto el usuario podría descifrarAES el contenido. El problema es que de este modo no se puede compartir la cuenta.

Por ejemplo, el campo “password_entry” del usuario A esta cifrado con la clave maestra de A. Cuando B se conecte no podrá descifrar el contenido del campo “password_entry” ya que no sabe la clave maestra de A. La única forma de conseguirlo es que A y B estén conectados al mismo tiempo y que el servidor descifre en ese momento el campo “password_entry” con la clave A y lo vuelva a cifrar de nuevo con la clave de B. En ese momento el usuario A ya no podrá acceder a su propia cuenta porque “password_entry” esta cifrado con la clave de otro usuario.

El enfoque correcto es usar una técnica de cifrado asimétrico llamado Key Group Encryption y el cifrado RSA. Esto consiste en que cada usuario tiene una clave pública y otra clave privada. La calve pública sirve para cifrar y la clave privada para descifrar. Si algo está cifrado con la clave pública sólo se puede descifrar con la clave privada. La clave pública la conoce todo el mundo porque es pública y la clave privada sólo la conoce el usuario. Aclarados estos términos se muestra un pre-ejemplo de cómo se comparten los datos:

Alice quiere compartir password_entry a Bob lo primero que hace es establecer una llave compartida y cifrarAES el contenido de password_entry con esa llave compartida. Después lo que va a hacer es cifrarRSA con la llave pública de Bob la llave compartida y la guarda en la base de datos. Nadie más puede acceder ahora a la clave excepto Bob

Cuando Bob se conecte y quiera ver el contenido de password_entry descifrarRSA la llave compartida con su llave privada. Acto seguido descifraraAES el contenido de password_entry con la llave compartida.

Esto describe el comportamiento del cifrado asimétrico. En la aplicación se ha hecho una pequeña optimización añadiendo una contraseña de más creando una cadena de

cifrado/descifrado. El esquema de cómo funciona el gestor de contraseñas realmente se muestra a continuación en la figura 5.1.

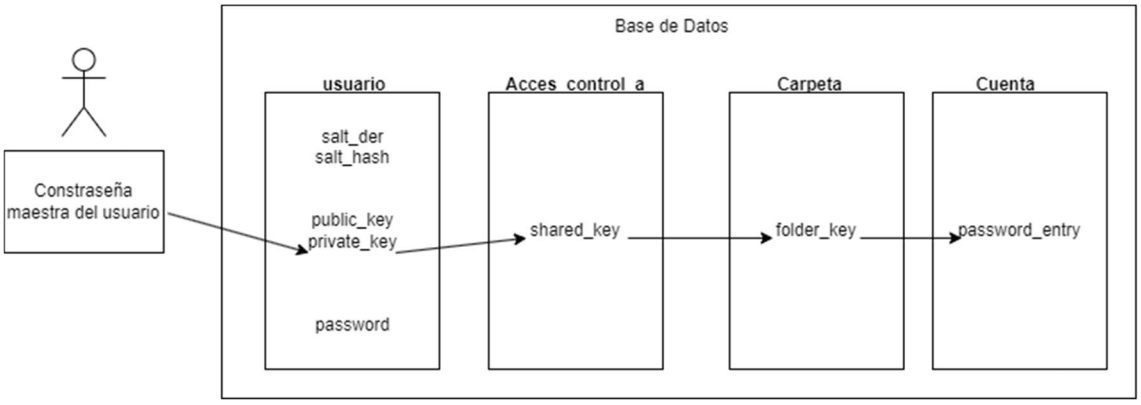


Figura 5.1. Esquema de cifrado y como un usuario accede a su contraseña.

Cuando un usuario se registra se realizan 3 cosas:

1. Se crea un par de llaves pública/privada.
2. Se deriva la contraseña maestra utilizando el salt_hash y se guarda en password.
3. Se deriva la contraseña maestra utilizando salt_der y se cifraAES la llave privada que se guarda en el campo private_key.

Después cuando el usuario quiere guardar una contraseña en password_entry sigue el servidor sigue los siguientes pasos:

1. CifrarAES el password_entry con la contraseña de folder_key.
2. CifrarAES la folder_key descifrada con la shared_key.
3. Derivar la contraseña maestra del usuario con salt_der y obtener la contraseña de servidor personal.
4. Con esa contraseña descifrarAES la clave privada.
5. CifrarRSA la shared_key descifrada con su clave privada.

Si se quiere compartir la contraseña con otro usuario lo único que se debe hacer es:

1. DescifrarRSA la shared_key con la clave privada.
2. CifrarRSA la shared_key descifrada con la clave pública del usuario con quien se quiere compartir.
3. Crear una entrada en acces_control_a con el id del usuario en cuestión, el id de la carpeta en cuestión los permisos que se el quiere dar y una fecha de expiración y guardar en el campo shared_key esa clave compartida cifrada.

Cuando el usuario al que se le compartió la carpeta quiera acceder al recurso seguirá los siguientes pasos:

1. Se deriva con salt_hash la contraseña maestra del usuario y se comprueba que coincide con password.

2. Una vez verificado se deriva la contraseña maestra con salt_der y con esta se descifraAES la clave privada. Esta se guarda en el almacenamiento “storage” como se menciona en capítulos anterior para que no se le tenga que pedir al usuario siempre la contraseña.
3. Se mira la tabla acces_control_a y se verifica que existe una entrada con el id del usuario y el id folder y que además tiene permisos para realizar la acción que desea. Si es así se descifraRSA la clave compartida guardada con la llave privada del usuario.
4. Se descifraAES la folder_key con la shared_key.
5. Se descifraAES el campo password_entry con la folder_key.

De esta manera se puede compartir recursos en la aplicación y no es posible acceder a los datos de los usuarios a menos que se tenga la contraseña maestra de este o que este usuario haya decidido compartir su contraseña con otro usuario en específico. Nadie ajeno puede leer los datos cifrados de los usuarios ni siquiera el propio administrador de la base de datos.

Para conseguir esto a nivel de servidor se ha creado un servicio nuevo con métodos muy interesantes para el cifrado y el descifrado como se muestra en la Figura 5.2

```
public byte[] AESEncrypt(IntPtr ptr_key, IntPtr ptr_data, int len_data)
{
    using (Aes aesAlg = Aes.Create())
    {
        byte[] data = new byte[len_data];
        byte[] key = new byte[32];
        Marshal.Copy(ptr_key, key, 0, 32);
        Marshal.Copy(ptr_data, data, 0, len_data);
        aesAlg.Mode = CipherMode.CBC;
        aesAlg.Padding = PaddingMode.PKCS7;
        byte[] iv = new byte[16];
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(iv);
        }
        aesAlg.IV = iv;
        aesAlg.Key = key;
        ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);
        byte[] encryptedData = encryptor.TransformFinalBlock(data, 0, data.Length);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(data);
            rng.GetBytes(key);
            rng.GetBytes(aesAlg.Key);
        }
        Console.WriteLine("crypt key");
        Console.WriteLine(encryptedData.Length);
        byte[] fullKey = iv.Concat(encryptedData).ToArray();
        Console.WriteLine(fullKey.Length);

        return fullKey;
    }
}
public byte[] AESDecryptToProtector(byte[] data, IntPtr ptr_key) {
    using (Aes aesAlg = Aes.Create()) {
        byte[] keyd = new byte[32];
        Marshal.Copy(ptr_key, keyd, 0, 32);
        byte[] fullKey = data;
        int ivLength = 16;
        byte[] iv = fullKey.Take(ivLength).ToArray();
        byte[] encryptedKey = fullKey.Skip(ivLength).ToArray();
        aesAlg.IV = iv;
        aesAlg.Key = keyd;
        aesAlg.Mode = CipherMode.CBC;
        aesAlg.Padding = PaddingMode.PKCS7;
        ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);
        byte[] decrypted = decryptor.TransformFinalBlock(encryptedKey, 0, encryptedKey.Length);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(aesAlg.Key);
        }
        return decrypted;
    }
}
public byte[] RSAEncrypt(byte[] public_key, IntPtr ptr_data)
{
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        rsa.KeySize = 2048;
        byte[] data = new byte[32];
        Marshal.Copy(ptr_data, data, 0, 32);
        rsa.ImportSubjectPublicKeyInfo(public_key, out _);
        byte[] claveCifrada = rsa.Encrypt(data, RSAEncryptionPadding.Pkcs1);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(data);
        }
        return claveCifrada;
    }

}
public IntPtr RSADecrypt(byte[] data, byte[] private_key)
{
    using (RSACryptoServiceProvider rsa = new RSACryptoServiceProvider())
    {
        rsa.KeySize = 2048;
        Console.WriteLine("Crypto");
        Console.WriteLine(Convert.ToBase64String(private_key));
        IntPtr ptr = Marshal.AllocHGlobal(32);
        rsa.ImportPkcs8PrivateKey(private_key, out _);
        byte[] key = rsa.Decrypt(data, RSAEncryptionPadding.Pkcs1);
        Marshal.Copy(key, 0, ptr, 32);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(key);
            rng.GetBytes(private_key);
        }
        return ptr;
    }

}
```

Figura 5.2. Métodos de cifrado y descifrado propios de los algoritmos AES-256 y RSA

Las claves que usa la aplicación para cifrar son 32 bytes y se generan aleatoriamente con la clase RSACryptoServiceProvider y el método GetBytes. El algoritmo AES usa un vector de inicialización de 16 bytes. Este se genera aleatoriamente de forma segura con la función GetBytes,

se cifran los datos con él y se concatena al principio de la llave cifrada ya que es necesario para poder descifrar la clave. El método AESDecryptToProtector sabe que el vector de inicialización se encuentra en los primeros 16 bytes, lo extrae y lo usa para descifrar los datos junto con la calve que se le proporcione.

De este modo no se tiene que estar guardando un campo “IV” en la base de datos y queda una estructura más limpia.

5.2 Rendimiento y Optimización

En el servidor se manejan muchas claves ya que para acceder a un recurso se tiene que pasar por un proceso largo de cifrado descifrado. Para esto esta la clase storage. Como se ha visto anteriormente la calve privada descifrada del usuario se guarda en el servicio UserStorage, diccionario_storage (ver figura B). Con todo esto se pueden realizar más optimizaciones para acortar aun más esta cadena de cifrado/descifrado.

Para hacer esto se puede implementar un diccionario donde la clave es el id de carpeta y el valor la calve de esa carpeta descifrada. Cada usuario tiene su propio diccionario personal y este se guarda en un diccionario general donde la calve es el id de usuario y el valor este diccionario personal. Así cuando el usuario accede a un recurso ya tiene disponible la clave de la carpeta y no tiene que pasar por todo el proceso de cifrado descifrado. En la Figura 5.3 se muestra la implementación de esto y sus métodos.

```
namespace password_manager.Services
{
    public class UserStorage
    {
        private readonly ConcurrentDictionary<int, byte[]> _storageKey;
        private readonly ConcurrentDictionary<int, ConcurrentDictionary<int, byte[]>> _folderA;

        public UserStorage()
        {
            _storageKey = new ConcurrentDictionary<int, byte[]>();
            _folderA = new ConcurrentDictionary<int,ConcurrentDictionary<int, byte[]>>();
        }

        public bool ExistUserKey(int id_user)
        {
            return _storageKey.ContainsKey(id_user);
        }

        public void AddUserKey(int id_user, byte[] key)...

        public void RemoveUserKey(int id_user)...
        public byte[] GetUserKey(int id_user) ...

        public bool ExistsFolderAKey(int id_user, int id_folderA)
        {
            if (!ExistUserKey(id_user))
            {
                return false;
            }
            var dict = _folderA[id_user];
            return dict.ContainsKey(id_folderA);
        }
        public byte[] GetFolderAKey(int id_user, int id_folderA)
        {
            var dict = _folderA[id_user];
            return dict[id_folderA];
        }
        public void SetFolderAKey(int id_user, int id_folderA, byte[] key)
        {
            var dict = _folderA[id_user];
            dict.AddOrUpdate(id_folderA, key, (keyExistente, nuevoValor) => key);

        }
        public void RemoveFolderAKey(int id_user, int id_folderA)
        {
            var dict = _folderA[id_user];
            dict.TryRemove(id_folderA, out _);

        }
    }
```

Figura 5.3. Contenido del archivo UserStorage.cs

Este diccionario debe tener las claves para poder usarlo, pero cuando un usuario hace se autentifica el diccionario personal está vacío y sólo tiene guardada su clave privada en _storageKey. Sería un error rellenar el diccionario al hacer login porque no se sabe si un usuario va a consultar todas las carpetas que tiene, además de que tardaría mucho en realizar todo el proceso.

La solución más efectiva es ir rellenándolo a medida que se van usando las carpetas. Cada método de los servicios hace una comprobación al inicio: mirar a ver si el id de la carpeta está en el diccionario personal. Si está, se utiliza. Si no está, se descifra y se guarda. Después la ejecución continúa con su curso.

Así, se rellena poco a poco a medida que el usuario usa el gestor de contraseñas y de esta forma no tiene un impacto notable en el rendimiento de la aplicación. A continuación, se muestra (Figura 5.4) el código hace esta tarea en el método ShareAccount del servicio AccountService.

```
if (!_storage.ExistsFolderAKey(id_user, request.id_folder_A))
{
    var accesControl_local = await _dbContext.AccesControlAs.FirstOrDefaultAsync(ac => ac.IdFolderA == request.id_folder_A && ac.PermA == "o");
    IntPtr ptr_shared_local = _crypto.RSADecrypt(accesControl_local.SharedKeyA, _protector.Unprotect(_storage.GetUserKey(id_user)));
    var folder = await _dbContext.FolderAccounts.FirstOrDefaultAsync(f => f.IdFolderA == request.id_folder_A);
    var folder_key_protected = _protector.Protect(_crypto.AESDecryptToProtector(folder.PasswordA, ptr_shared_local));
    _crypto.ClearPTR(ptr_shared_local);
    _storage.SetFolderAKey(id_user, request.id_folder_A, folder_key_protected);
}
```

Figura 5.4. Inicio del método ShareAccount del archivo AccountService.cs

5.3 Cifrado de memoria

Como se puede observar el servidor guarda mucha información privada del usuario mientras está en ejecución sobre todo en UserStorage y un atacante malicioso que consiguiera acceso al servidor o el propio Administrador podría hacer un volcado de memoria en busca de las claves de cifrado de los usuarios. Por eso es necesario cifrar los datos que están en memoria.

Para eso se usa el servicio Data Protection. [16] Es una característica proporcionada por la plataforma para ayudar a los desarrolladores a proteger y cifrar datos confidenciales que se almacenan o transmiten dentro de una aplicación. Esta característica está diseñada para facilitar la tarea de asegurar datos sensibles, como contraseñas, tokens de acceso, información personal u otra información confidencial que una aplicación pueda necesitar procesar.

El servicio Data Protection se basa en la API de protección de datos de .NET, que permite cifrar y descifrar datos de forma segura utilizando claves de protección específicas de la aplicación y del sistema operativo. La idea principal detrás de este servicio es que los desarrolladores no necesiten preocuparse por la implementación de algoritmos criptográficos complejos o la gestión de claves de cifrado, ya que el servicio se encarga de todo eso.

Para usarlo primero se configura el servicio como añadiendo la siguiente línea en el archivo Program.cs: `builder.Services.AddDataProtection().SetApplicationName("MyApp");`

A continuación, se inyecta el servicio y se inicializa en el constructor del servicio que se va a usar. En la figura 5.5, se muestra el servicio AccountService.

```
public class AccountService
{
    private readonly mydbContext _dbContext;
    private readonly UserStorage _storage;
    private readonly IConfiguration _config;
    private readonly IDataProtector _protector;
    private readonly CryptoService _crypto;

    public AccountService(mydbContext dbcontext, IConfiguration config, IDataProtectionProvider dataProtectionProvider, UserStorage storage, CryptoService cs)
    {
        _dbContext = dbcontext;
        _storage = storage;
        _config = config;
        _protector = dataProtectionProvider.CreateProtector(_config.GetValue<string>("JwtConfig:ProtectorName"));
        _crypto = cs;
    }
```

Figura 5.5. Constructor de la clase AccountService.

De esta forma usando _protector.Protect(clave) se cifra y devuelve un payload que puede ser guardado en memoria de forma segura. Usando _protector.Unprotect(payload) se obtiene de nuevo la clave.

Este no soluciona todo ya que las claves se guardan en los tipos de datos que proporciona la plataforma de .NET ,principalmente array de bytes(byte[]). Esto es un problema ya después de utilizar una variable convencional de tipo byte[] o String, hay que asegurarse de que esta no va a quedar en memoria.

Esto es especialmente difícil por culpa del recolector de basura. En la administración de memoria es responsabilidad del Recolector de Basura (Garbage Collector, GC). [17] El Recolector de Basura es un componente esencial del tiempo de ejecución de .NET que se encarga de liberar automáticamente la memoria que ya no está en uso por objetos en el programa.

Cuando un programa .NET crea objetos en tiempo de ejecución, el Recolector de Basura realiza un seguimiento de dichos objetos y se asegura de que los que ya no son accesibles desde el código se marquen como basura. Luego, el recolector de basura identifica y recupera la memoria ocupada por estos objetos no utilizados, lo que libera recursos y ayuda a prevenir pérdidas de memoria y problemas de rendimiento.

.NET utiliza un enfoque de recolección de basura de generaciones, que divide los objetos en diferentes generaciones según su tiempo de vida. Los objetos recién creados se colocan en la generación más joven (gen 0) y, si sobreviven a la recolección de basura, pueden moverse a generaciones más antiguas. La recolección de basura se realiza de manera periódica o según la necesidad del sistema operativo y el tiempo de ejecución de .NET.

Esto es un reto ya que el programador ya no controla la memoria. Se podrían usar objetos e implementar la interfaz IDisposable. Después se podría llamar al método dispose() y rezar para que el recolector de basura lo elimine ya que según la documentación el método dispose() no borra el objeto, sólo lo marca para que el recolector de basura sepa que tienen que eliminarlo.

Tampoco sirve llamar al recolector de basura manualmente con el método GC.Collect() ya que no se le puede especificar un objeto en concreto y el GC hará un escaneo de toda la memoria otra vez en busca de objetos marcados para su eliminación y tendría un impacto enorme en el gestor de contraseñas ya que este está todo el rato cifrando y descifrando.

Modernamente se pueden usar cláusulas using e introducir el código critico allí. El problema es que, según la documentación, el using llama a disposble() cuando acaba de ejecutar el código contenido en la cláusula. En verdad hace lo mismo solo que el programador no tiene que llamar manualmente al método.

La última opción que queda es programar de forma que justo después de usar una variable se sobrescriba su contenido con contenido aleatorio. Se hace con el método presentado anteriormente GetBytes(variable a sobrescribir). Esto tampoco es una solución ya que el GC es

dueño de toda la memoria y según la documentación puede incluso hacer copias de variables y objetos y trasladarlos por el heap a placer con el fin de optimizar la memoria.

Esto es un gran problema ya que si se usa la variable y justo después el GC decide copiar su contenido a otra dirección de memoria y se sobrescriben los datos, el programador puede pensar que se ha borrado pero el objeto sensible aún está en memoria esperando para ser borrado. Además, en la documentación no se proporciona información detallada a nivel de código sobre el planificador del GC y no queda otra que fiarse de él.

La solución es no usar los tipos de datos que ofrece .NET. Usar un tipo de dato al que GC no tiene acceso y son los punteros a memoria no administrada. Eso es una variable especial que contiene una dirección de memoria no administrada. En .NET, esto se logra utilizando el tipo de dato IntPtr, que se utiliza para representar un puntero a memoria no administrada.

Para trabajar con memoria no administrada, se puede utilizar el espacio de nombres System.Runtime.InteropServices, que contiene clases y atributos para trabajar con punteros a memoria no administrada y realizar llamadas a funciones no administradas desde el código administrado.

En el código de la figura 5.6 se muestran los métodos de CryptoService que no se enseñaron en la figura 5.2. Ahora se entiende bien el motivo por el cual se han creado esta clase especial con métodos propios (aparte de que el código queda mucho más legible).

```
public class CryptoService
{

    public byte[] AESEncrypt(IntPtr ptr_key, IntPtr ptr_data, int len_data)...
    public byte[] AESDecryptToProtector(byte[] data, IntPtr ptr_key) ...
    public byte[] RSAEncrypt(byte[] public_key, IntPtr ptr_data)...
    public IntPtr RSADecrypt(byte[] data, byte[] private_key)...
    public IntPtr CreateKeyPTR()
    {   IntPtr ptr = Marshal.AllocHGlobal(32);
        byte[] key = new byte[32];
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(key);
            Marshal.Copy(key, 0, ptr, 32);
            rng.GetBytes(key);
        }
        return ptr;
    }
    public void ClearPTR(IntPtr ptr)
    {
        byte[] key = new byte[32];
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(key);
            Marshal.Copy(key, 0, ptr, 32);
        }
        Marshal.FreeHGlobal(ptr);

    }
    public void ClearPTR(IntPtr ptr, int len)
    {
        byte[] key = new byte[len];
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(key);
            Marshal.Copy(key, 0, ptr, len);
        }
        Marshal.FreeHGlobal(ptr);

    }
    public byte[] retBytes(IntPtr ptr, int len)
    {
        byte[] key = new byte[len];
        Marshal.Copy(ptr, key, 0, len);
        return key;
    }
    public byte[] retBytes(IntPtr ptr)
    {
        byte[] key = new byte[32];
        Marshal.Copy(ptr, key, 0, 32);
        return key;
    }
    public (IntPtr, int) retPtr(byte[] secret) {
        int len = secret.Length;
        IntPtr ptr = Marshal.AllocHGlobal(len);
        Marshal.Copy(secret, 0, ptr, len);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(secret);
        }
        return (ptr, len);
    }
    public IntPtr retKeyPtr(byte[] secret)
    {
        IntPtr ptr = Marshal.AllocHGlobal(32);
        Marshal.Copy(secret, 0, ptr, 32);
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(secret);
        }
        return ptr;
```

Figura 5.6. Fichero CryptoService.cs

Estos métodos son útiles para trabajar con punteros en los servicios propios. Porque crean claves, devuelven punteros borrando memoria y así se puede encadenar estas funciones con los métodos de Data Protector para que el flujo de datos no pase nunca por variables con tipos de datos convencionales (véase figura 4.13).

Un último detalle y es que desde el Frontend puede llegar un DTO con información sensible. En ese caso lo que se hace es inmediatamente en el controlador pasarlo a un puntero(si se va a usar dentro de 3 líneas de código) y sobrescribir la variable del DTO o protegerlo con _protector.Protect() y sobrescribir la variable del DTO cómo se muestra en la Figura 5.7.

```
var tuple_result = _crypto.retPtr(Encoding.UTF8.GetBytes(request.password_entry));
IntPtr ptr_password_entry = tuple_result.Item1;
int len_password_entry = tuple_result.Item2;
request.password_entry = "################";
```

Figura 5.7. Pasar la información sensible del DTO a puntero.

5.4 Cifrado de claves de configuración

Después de toda la parte de cifrado aún queda un detalle por cifrar: las variables de configuración.

La primera es la clave con la que Data Protector inicia el servicio de protección. Si un atacante se hiciera con esta clave podría replicar el funcionamiento del servicio y descifrar los datos del memoria. Por eso se decide crear una clave aleatoria al iniciar el servidor para que cada vez que se reinicie esta cambie y ni siquiera el dueño del servidor pueda acceder a ella . Esto se muestra en la Figura 5.8.

```
byte[] randomBytes = new byte[32];
using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
{
    rng.GetBytes(randomBytes);
}
builder.Services.AddDataProtection().SetApplicationName(Convert.ToBase64String(randomBytes));
```

Figura 5.8. Configuración segura del servicio Data Protector

La segunda es la clave con la que el servicio de tokens cifra los tokens JWT(JSON Web Tokens) ya que estos no se pueden extraer del appsettings.json. El dueño del servidor incluso podría firmarse un token con los datos que quiera y acceder en nombre de cualquier usuario. Para solucionar esto primero se crea una clase según el patrón de diseño Singleton como se muestra en la Figura 5.9

```
namespace password_manager.Services
{
    public sealed class SealedVariables
    {
        private static readonly SealedVariables _instance = new SealedVariables();
        private byte[] _jwtKey;

        private SealedVariables()
        {
        }

        public static SealedVariables Instance
        {
            get { return _instance; }
        }

        public void InitializeJWT(byte[] arrBytes)
        {
            if (_jwtKey == null)
            {
                _jwtKey = arrBytes;
            }
        }

        public byte[] GetImmutableArray()
        {
            return _jwtKey.ToArray();
        }
```

Figura 5.9. Fichero SealedVariables.cs

La implementación del patrón Singleton aquí asegura que sólo exista una instancia de SealedVariables durante toda la ejecución del programa, lo que puede ser útil cuando se desea compartir una misma instancia de una clase entre diferentes partes del código sin crear duplicados innecesarios. El patrón Singleton es útil porque se necesita mantener un estado compartido entre diferentes componentes de una aplicación y no se desea que este estado sea instanciado o compartido a través de múltiples instancias de la clase. Es un contenedor para la clave privada de firma de tokens.

Luego en la configuración del token se crea una clave aleatoria se protege con Data Protector y se guarda en SealedVariables instanciado la clase como muestra la Fgura 5.10.

```
builder.Services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        byte[] keyBytes = new byte[16];
        byte[] protectedKeyBytes = new byte[16];
        using (var serviceProvider = builder.Services.BuildServiceProvider())
        {
            var dataProtectionProvider = serviceProvider.GetService<IDataProtectionProvider>();
            var protector = dataProtectionProvider.CreateProtector(builder.Configuration["JwtConfig:ProtectorName"]);
            using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
            {
                rng.GetBytes(keyBytes);
            }
            protectedKeyBytes = protector.Protect(keyBytes);
            SealedVariables.Instance.InitializeJWT(protectedKeyBytes);
        }
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,

            ValidIssuer = builder.Configuration["JwtConfig:Issuer"],
            ValidAudience = builder.Configuration["JwtConfig:Audience"],
            IssuerSigningKey = new SymmetricSecurityKey(keyBytes),
        };
        using (RandomNumberGenerator rng = RandomNumberGenerator.Create())
        {
            rng.GetBytes(keyBytes);
        }

    });
```

Figura 5.10. Configuración de tokens JWT segura.

Por último, cuando se firma un token se tiene que conocer la clave de firma. Esta se extrae primero de SealedVariables, se descifra con _protector.Unprotect y se usa para firmar como se muestra en la Figura 5.11.

```
var tokenHandler = new JwtSecurityTokenHandler();
var tokenDescriptor = new SecurityTokenDescriptor
{

    Subject = new ClaimsIdentity(new Claim[]
    {
        new Claim(ClaimTypes.NameIdentifier, user.IdUser.ToString()),
        new Claim(ClaimTypes.Email, user.Email)
    }),
    Audience = _config.GetValue<string>("JwtConfig:Audience"),
    Issuer = _config.GetValue<string>("JwtConfig:Issuer"),
    Expires = DateTime.UtcNow.AddMinutes(_config.GetValue<double>("JwtConfig:AccessTokenExpirationInMinutes")),
    SigningCredentials = new SigningCredentials(new SymmetricSecurityKey(_protector.Unprotect(SealedVariables.Instance.GetImmutableArray())), SecurityAlgorithms.HmacSha256Signature)
};
var token = tokenHandler.CreateToken(tokenDescriptor);

// Retorna el token de autorización en la respuesta
return Ok(new
{
    access_token = tokenHandler.WriteToken(token)
});
```

Figura 5.11. Firma de tokens JWT segura. Parte del fichero LoginController.cs

6. Bibliografía

[1] Michael Carr and Siamak Shahandashti Revisiting Security Vulnerabilities in Commercial Password Managers arXiv:2003.01985

[2] Dashlane whitepaper https://www.dashlane.com/download/whitepaper-en.pdf

[3]lastpass whitepapaper https://support.lastpass.com/s/document-item?language=en_US&bundleId=lastpass&topicId=LastPass/lastpass_technical_whitepaper.html&_LANG=enus

[4] Nordpass whitepapaper https://nordpass.com/nordpass-business-whitepaper.pdf

[5] LastPass Hacked https://threatpost.com/lastpass-network-breached-calls-for-master-password-reset/113324/

[6] https://restfulapi.net/

[7] https://www.w3schools.com/xml/xml_soap.asp

[8]https://graphql.org/learn/

[9] https://grpc.io/docs/

[10] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[10] https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API

[11] https://insights.stackoverflow.com/survey/2021#most-popular-technologies-webframe-prof

[12] https://learn.microsoft.com/es-es/dotnet/core/whats-new/dotnet-6

[13] https://auth0.com/docs/secure/tokens/json-web-tokens

[14] https://survey.stackoverflow.co/2022#technology

[15] Withdrawn NIST Technical Series Publication https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.197.pdf

[16] HTTPS://LEARN.MICROSOFT.COM/EN-US/ASPNET/CORE/SECURITY/DATA-PROTECTION/INTRODUCTION?VIEW=ASPNETCORE-7.0

[17] HTTPS://LEARN.MICROSOFT.COM/EN-US/DOTNET/CORE/RUNTIME-CONFIG/GARBAGE-COLLECTOR