

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

“Mejora de la
seguridad y
funcionalidad de UAL
Inventarium”

Curso 2021/2022

Alumno/a:

Alejo Martín Arias Filippo

Director/es:

Juan Francisco Sanjuan Estrada
Alfonso José Bosch Aran



Un aspecto muy importante que hay que considerar dentro del desarrollo web es el de la seguridad de las interacciones que se originan entre el cliente y el servidor. Poder ampliar las funcionalidades que nos ofrece una herramienta representa perfectamente lo que podría ser un entorno de trabajo donde, luego de haber sido entregada la solución al paso del tiempo, el cliente nos pide nuevas funcionalidades. El poder dotar a la herramienta de seguridad en sus transacciones y de una comunicación con el usuario es un aspecto complementario ideal para el desarrollo web realizado en el trabajo fin de grado. Esto favorecería en gran medida a los futuros usuarios que vayan a utilizar la aplicación. Este complemento pretende cubrir campos esenciales en la seguridad del sitio web como puede ser la tokenización de las consultas o la protección del acceso a determinadas rutas de la página. También ampliaré las funcionalidades que nos ofrece la web, implementando un gestor de correos que nos ayudara en todas las interacciones con el usuario. Desarrollaré un sistema de copias de seguridad para la base de datos que permitirá poder recuperar los datos almacenados en UAL Inventarium con la mínima pérdida de información.

Índice general

1	Introducción	5
1.1	Motivaciones	5
1.2	Objetivos	5
1.2.1	La seguridad	6
1.2.2	La escalabilidad	6
1.2.3	La comunicación	6
1.3	Planificación	6
2	Herramientas utilizadas	8
2.1	Hardware	8
2.1.1	Ordenador de sobremesa	8
2.1.2	Ordenador portátil	8
2.2	Software	8
2.2.1	Entorno de desarrollo	8
2.2.2	Redacción del documento	8
2.2.3	Google Cloud	9
2.2.4	Docker Swarm	9
3	Implementación de las funcionalidades	10
3.1	Tokenización	10
3.1.1	Generación del token	10
3.1.2	El middleware	11
3.1.3	¿Cómo se añade el proceso de verificación a cada consulta?	12
3.1.4	AuthGuard	12
3.2	Copias de seguridad	14
3.2.1	MySQL-Backup	15
3.3	Servidor de correo	16
3.3.1	Nodemailer	16
3.3.2	¿Dónde implementar esta nueva funcionalidad?	16
3.4	Balanceo de cargas	19
3.4.1	Docker Swarm	19
3.4.2	¿Cómo podemos implementar Docker Swarm?	19
3.4.3	¿Cómo desplegar nuestro entorno en Docker Swarm?	21
4	Conclusiones y mejoras	23
	Referencias	24

Índice de figuras

3.1	Creando segunda máquina virtual	20
3.2	Entorno de máquinas en Google Cloud	20
3.3	Inicializando el entorno de docker swarm	20
3.4	Añadiendo nodo esclavo a nuestro entorno	21
3.5	Deploy de nuestro entorno Docker en el clúster	21
3.6	Visualización de los servicios corriendo en Docker Swarm	22

1 Introducción

Luego de la construcción de nuestra aplicación el siguiente paso es de cubrir varios aspectos que son claves hoy en día dentro de la producción de componentes software.

Las características de nuestro software son bastante completas gracias a haber respetado modelos de construcción asentados y con un marco teórico y experiencial por detrás, pero, ¿y la seguridad de la aplicación? Cómo se va a realizar el proceso de autenticación de usuarios o por el cual se van a cubrir las solicitudes a la API. O, ¿cómo podemos garantizar que el usuario es miembro de la Universidad? ¿O cómo garantizamos que a los técnicos se enteren de las solicitudes de préstamos que les llegan?

Dentro de este complemento de trabajo fin de grado nos dedicaremos a la resolución de estas problemáticas que surgen al haber finalizado la implementación de nuestra aplicación.

1.1. Motivaciones

Por respetar en pequeña parte la estructura del trabajo fin de grado he decidido poder realizar una pequeña sección donde hablar de los motivos de la realización de este complemento.

El motivo principal es que lo tengo que realizar si o sí, de eso no cabe duda. El segundo radica en las posibilidades que nos brinda el software de las que tanto he hablado en el proyecto.

Una herramienta no acaba con el satisfacer los requisitos que se nos pedían en un principio. Creo que este aspecto nos diferencia bastante de las otras ingenierías. Puede ser quizás a que nuestra forma de implementación es más fácil, quizás. Pero la construcción de herramientas normalmente se expande forma continua y uniforme.

Lo podemos ver con la evolución de las redes sociales de hoy en día. Pongamos como ejemplo, Facebook. Quiero recalcar que en esta sección no estoy tratando de la originalidad de las ideas sino de la ejecución de las mismas.

Facebook al principio de los 2000 no era más que una red social cualquiera. En la que la gente podía interactuar, publicar lo que habían hecho durante el día o habían aprendido y más tarde alrededor del 2008 compartir logros dentro de un sistema de juegos integrados que venían con la plataforma. Ya la implantación de los juegos es “algo”, pero no fue hasta que comercializaron aplicaciones como Whatsapp o Instagram donde en verdad tuvo su éxito.

La tecnología da más tecnología. Es una estufa que no se apaga y las implementaciones y añadidos que se pueden realizar a un proyecto son gigantescas. Del modo y forma debidamente adecuados.

1.2. Objetivos

¿Qué se pretende con la realización de este complemento? Nuestro objetivo principal es la mejora de la aplicación relacionada en tres áreas:

- La seguridad.
- La escalabilidad.
- La comunicación.

1.2.1. La seguridad

La seguridad es algo fundamental que se pide hoy en día en internet. Y algo que suele verse relegado a un segundo plano por bastantes empresarios.

El objetivo de este apartado es centrarnos en dos aspectos: la seguridad en las transacciones con la Interfaz de Programación de Aplicaciones (API), esto lo realizaremos mediante una tokenización de las transacciones que gestionaremos al principio de nuestra aplicación. Lo segundo es la implementación de un sistema de copias de seguridad automatizado que nos ayudará a que en caso de pérdida de información podamos recuperar parte de esta gracias a ficheros almacenados de forma externa que contengan esa información. Estas las iremos generando siguiendo algunas normativas de generación de copias de seguridad en entornos empresariales.

1.2.2. La escalabilidad

En el proyecto hablé sobre Docker y todas las utilidades que nos podía brindar. En este caso hablaremos de Docker Swarm, un sistema que implementa Docker para poder realizar un balanceo de carga en nuestra aplicación y que esta no se vea sobrecargada. Ideal por si ocurren caídas en algunas de nuestras máquinas o queremos rebajar un poco los recursos que consumen. Explicaré cómo implementaremos Docker Swarm en nuestro entorno y cómo estos tipos de herramientas pueden ser de gran utilidad para aspectos como la escalabilidad.

1.2.3. La comunicación

La interacción que una aplicación tiene con su usuario es un aspecto básico que tiene que ser cubierto.

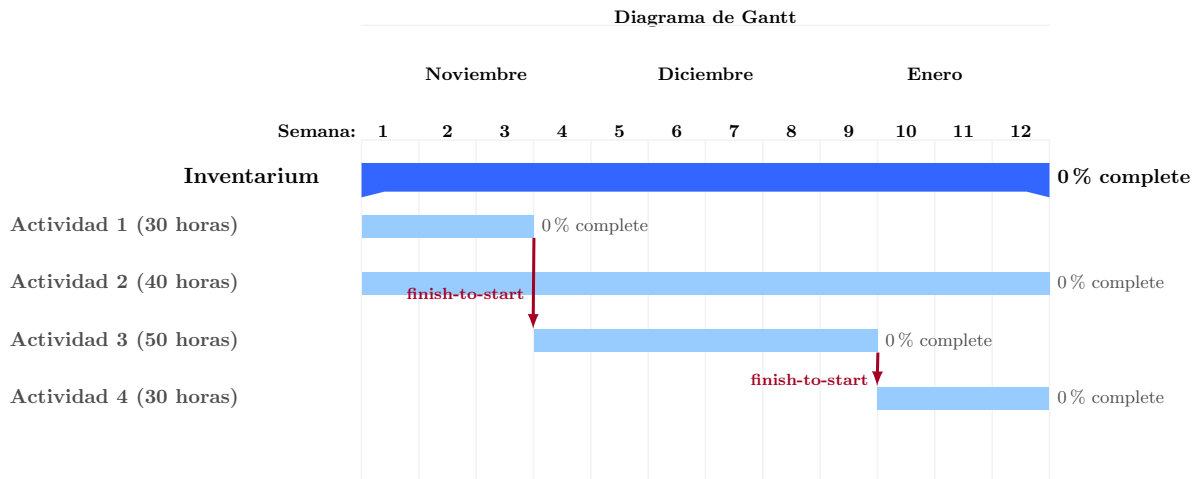
Esta comunicación hoy en día se puede realizar de diferentes formas, por ejemplo, el pitido que hace un microondas cuando termina su ejecución. Ese ya es un método de comunicación que está teniendo con el usuario.

Al principio del auge de internet y de los sistemas informáticos la comunicación era unidireccional, es decir, iba del usuario a la máquina, y esta ya mostraba lo que el usuario le pedía. Nuestra aplicación funciona al igual que hacían las páginas webs hace 20 años. El usuario si necesita algo o quiere consultar si le han concedido una solicitud tiene que buscarlo.

Este aspecto lo cubriremos gracias a la implementación de un servidor de gestión de correo dentro de nuestra API. También decidiremos los modos y momentos en los que la aplicación se comunicará con el usuario.

1.3. Planificación

La planificación se dividirá en cuatro grandes grupos, muy parecidos a los del proyecto: Reuniones con el cliente, planificación y elaboración del desarrollo del proyecto, construcción de la aplicación y testeo y comprobación de la aplicación y comprobación de errores. Son los mismos puntos que en el proyecto nada más que no trataremos el punto de “Preparación del entorno de trabajo”.



- **Actividad 1** (30 horas) Reuniones con el cliente
 - Reunión para tratar sobre la comunicación a cubrir con el usuario
 - Reunión para tratar sobre la seguridad y la escalabilidad
- **Actividad 2** (40 horas) Planificación y elaboración del proyecto
- **Actividad 3** (50 horas) Construcción de la aplicación
 - Tokenización de las solicitudes a la base de datos
 - Configuración de generador automático de copias de seguridad
 - Implementación del sistema de envío de correos y gestión de la comunicación
 - Implementación de Docker Swarm
- **Actividad 4** (30 horas) Testeo, comprobación de la aplicación y comprobación de errores

La metodología del trabajo a realizar será en cascada. Es decir, iremos cubriendo cada uno de los apartados y hasta no terminar con el superior no podremos pasar al siguiente. Este modelo se realizará para las tres implementaciones que haremos.

Las primeras fases se centrarán en el análisis y planificación de la solución que se pretenda implementar. Las fases intermedias consistirán en esta implantación de la aplicación y las finales se centrarán en la realización de pruebas para la comprobación de que todas las funcionalidades que se pensaban implementar en un momento cumplen a la perfección su función y lo hacen de forma correcta.

2 Herramientas utilizadas

En este capítulo se hablará sobre las herramientas utilizadas durante el desarrollo del proyecto. Algunas de ellas ya se han presentado en el anterior proyecto pero no serán demasiadas para no volver tan extenso el documento.

En la sección de hardware, eso sí, disponemos de los mismos dispositivos que antes así que no va a haber variación.

2.1. Hardware

Dentro del apartado de hardware disponemos de dos ordenadores. Su procesador no conlleva relevancia en el desarrollo de la aplicación debido a la utilización de servicios en la nube.

2.1.1. Ordenador de sobremesa

La cual contiene como procesador un Xeon E5-2620 V3. 16GB de RAM DDR3. Una tarjeta gráfica RTX 570 de 4GB DDR5. Tiene 256GB de memoria SSD y 1TB de memoria HDD.

2.1.2. Ordenador portátil

Es un MacBook Air M1 de 2020 con 8GB de RAM y 256GB de almacenamiento.

2.2. Software

2.2.1. Entorno de desarrollo

El entorno de desarrollo y desde donde se hará casi absolutamente todo será Visual Studio Code. Un editor de código desarrollado por Microsoft que soporta varias distribuciones de sistemas operativos, entre ellas: Windows, Mac Os y Ubuntu.

Una de las características de esta herramienta que la hacen la predilecta de varios desarrolladores es el gran soporte que tiene por parte de la comunidad. Tiene un mercado de plugins bastante grande que apoya la creación continua de código para todos los desarrolladores.

Dispone de integraciones con Git, resaltado en errores de sintaxis, finalización de código y hasta conexión remota a otros entornos de trabajo mediante SSH.

Otra enorme ventaja que presenta es el consumo de memoria que tiene, bastante pequeño. Es un programa para ordenadores de todos los tamaños y precios, un software gratuito y una herramienta increíblemente potente al alcance de todos.

2.2.2. Redacción del documento

Estas líneas están siendo escritas ahora mismo desde LaTeX. LaTeX es un sistema de composición de textos que está formado mayoritariamente por órdenes construidas a partir de comandos TeX. En un principio no estaba seguro de qué herramienta utilizar, ya que la posición de varios profesores respecto a esta herramienta era bastante férrea pero Word siempre había ido agarrado a mi mano desde comienzos del instituto.

Luego de pasar de Word a LaTeX y de LaTeX a Word bastantes veces no fue hasta que mi profesora Rosa, en una de mis visitas matinales a su despacho me dijo: Yo hice mi TFG en

LaTeX.

No me lo podía creer y al comprobar la fecha de publicación de este programa de procesado de textos me sorprendí al ver que su lanzamiento oficial fue en 1980. “Si el programa ha durado tanto es que algo de importante tendrá” pensé. Y aquí me hallo redactando este documento con un programa que facilita el control de versiones de Git de una manera asombrosa. Facilita también los procesos de documentación y disposición de las diferentes subsecciones. Y, lo que más me gusta sin lugar a dudas, que puedo realizar una separación de cada capítulo por documentos separados y es que a mí, el tener las cosas modularizadas, me puede.

2.2.3. Google Cloud

Se volverá a usar Google Cloud pero esta vez no para el deploy de la página web, sino que será para la generación de un entorno red donde se desplegará una máquina virtual más. Esto se hará con el objetivo de generar una unión entre ellas y poder utilizar una de las herramientas que incorpora Docker.

2.2.4. Docker Swarm

Docker Swarm permite tener varios contenedores Docker interconectados entre sí en diferentes máquinas virtuales o físicas.

Es decir, Docker Swarm permite la gestión de un clúster de servidores Docker. Además aporta herramientas para poder gestionar estos como si se tratase de la gestión de un simple contenedor.

3 Implementación de las funcionalidades

3.1. Tokenización

La utilización de tokens durante el desarrollo de APIs ha incrementado a lo largo de los últimos años. Estos tokens permiten aislar de esa capa de seguridad de implementación extra a la que antes se estaba sometido por hacer aplicaciones que gestionasen sus propias consultas con la base de datos.

A medida que se iba realizando la construcción de la aplicación ya se tenía en mente esta implementación de seguridad.

En un principio se almacenaron los datos del usuario y la contraseña de este en caché. Cada vez que se mandaba una consulta se utilizaban los datos en caché del usuario para analizarla y proceder a un sistema de autenticación.

Este proceso de autenticación también iba ligado a una “API” podría llegar a considerarse aunque nunca llegó a ser tan potente ni bien planificada como la actual.

El proceso de generación de tokens desde un entorno del cliente resulta un proceso algo tedioso y que nunca hay que realizar. Por suerte, para el entorno del servidor de la API había algunas herramientas que iban a resultar de gran ayuda.

3.1.1. Generación del token

La generación de tokens siempre ha sido un proceso dudoso para el usuario intermedio, ya que, en un principio, se suele pensar que el almacenamiento de este se realiza sobre la base de datos en vez de en el propio servidor. Por suerte existe un plugin ideal que se encuentra en el Node Package Manager para esta tarea: `jwt-simple`.

Gracias a este plugin se puede realizar el codificado y decodificado de un plugin generado cada vez que un usuario iniciara sesión.

Dentro de las fases del codificado de este token hay que pasarle como parámetros unas fechas que será la duración que tendrán de validez estos “simbólicos” (traducción de token al español). El proceso de codificado del token se realizará dentro del inicio de sesión. Quedando de la siguiente forma:

```
const createToken = (usuario) => {
  let payload = {
    userId: usuario.idUsuario,
    createdAt: moment().unix(),
    expiresAt: moment().add(1, 'day').unix()
  }
  return jwt.encode(payload, process.env.TOKEN_KEY);
};
```

Esta función se ejecuta cuando se ha podido corroborar que el usuario ha iniciado sesión correctamente y sin problemas.

Lo que se codifica dentro de la variable `TOKEN_KEY` ubicada en el archivo `.env` es un payload que cogerá como parámetros: el id del usuario, el momento en el que se ha logueado y el momento donde expirará su sesión.

Teniendo ya el token generado hay que, de alguna forma, hacer que cada vez que el usuario quiera hacer una consulta lo haga desde su token.

¿Por qué se almacena la id del usuario en el payload?

La id del usuario se almacena porque cada vez que se quiere acceder a una consulta siendo clientes se podrá meter en cada una de las solicitudes ese nuevo campo. Esta acción se realizará siempre por lo que se dispondrá de los datos del usuario en cada uno de los casos de uso.

Por ejemplo, si un usuario intenta acceder a una consulta la cual solo se le permite a los técnicos, gracias a poder identificar el id del usuario en base a su token podrá denegarse dicha consulta y que no se permita.

3.1.2. El middleware

El middleware, como concepto, representa a todo software que se sitúa entre el software y las aplicaciones que corren sobre él. Este funciona como una capa de traducción que posibilita la comunicación y la administración de datos en aplicaciones distribuidas. En este caso la función no trabajará en si como un “middleware” pero si que será una capa intermedia que se irá ejecutando a medida que el usuario interactúe con la API aportando la comprobación de la validez de su token y su id de usuario.

Dentro del middleware se tendrá una única función que correrá en cada momento que el usuario acceda a uno de los endpoints.

La implementación sería la siguiente:

```
const jwt = require("jwt-simple");
const moment = require("moment");

const checkToken = (req, res, next) => {
  if (!req.headers['user_token'])
    return res.json({
      error: "You must include the header"
    });

  const token = req.headers['user_token'];
  let payload = null
  try {
    payload = jwt.decode(token, process.env.TOKEN_KEY);
  } catch (err) {
    return res.json({
      error: "Invalid token"
    });
  }

  if (moment().unix() > payload.expiresAt) {
    return res.json({ error: "Expired token" });
  };

  req.userId = payload.userId;

  next();
};
```

Dentro de esta función se realizan 3 comprobaciones.

Comprobación de los headers

El token irá dentro del encabezado de las solicitudes que se hagan a la API. En concreto con la key “user_token”. Si no se incluye este encabezado no se podrá continuar con la comprobación por lo que se devuelve como respuesta al cliente *You must include the header*.

Decodificación del payload

Luego de la comprobación de que haya “algo” al menos dentro del header se procede a llamar al plugin de jwt y preguntarle si tiene un payload almacenado con la clave.

En el caso de que no pueda identificarlo, es decir, que no sea válido, se obtendrá como respuesta en el sitio web el siguiente mensaje: *Invalid token*.

Verificación de la validez

Después del segundo paso se puede haber obtenido un payload pero hay que comprobar que este no haya expirado. Es decir, hay que acceder a su variable “expiresAt”. En el caso de que lo haya hecho devolverá un error como respuesta con el siguiente mensaje: *Expired token*.

Se incorporará el user id a la solicitud y con esto ya se tendría implementado el middleware.

3.1.3. ¿Cómo se añade el proceso de verificación a cada consulta?

Gracias a haber hecho el código de forma estructurada y haber distinguido en servicios y rutas con solo añadir una línea al campo de rutas hará que se pueda añadir este paso de verificación intermedio.

Primero se importará el fichero “middleware” y luego se llamará en el código con la siguiente función:

```
router.use(middleware.checkToken);
```

¿Cómo se inicia sesión si no se tiene un token?

La respuesta es muy sencilla y va en relación a la lógica del archivo de rutas. Al entrar una petición a uno de los endpoints lo que ocurre es que lo hace en forma de barrido en los ficheros. Es decir que hasta que no llegue a la línea de código de antes no se habrá hecho la comprobación.

Por lo tanto, la solución es que dentro del fichero donde se realice el login y las demás acciones del usuario, se coloque esta primero en la parte de arriba, en medio la comprobación y abajo el resto de consultas que también se puedan hacer.

3.1.4. AuthGuard

AuthGuard es un “comprobador de usuarios”. Es decir, comprueba que estés logeado cogiendo tu token y mandándolo a la API, si esta no devuelve respuesta o lo hace con un *null* el AuthGuard comunicaría que el usuario no está logeado.

Angular brinda la posibilidad de poder realizar una autenticación dentro del sistema de rutas de la aplicación. Es decir, dependiendo de a qué sitios se quiera acceder de la aplicación se pueden implementar unas medidas de seguridad u otras.

Esto se consigue mediante la adición del parámetro **canActivate** y **canActivateChild**:

```
path: '', component: MainComponent,  
canActivate: [AuthGuardService],  
canActivateChild: [AuthGuardService],
```

```
children: [  
  { path: 'dashboard', component: DashboardComponent },  
  {  
    path: 'add-object', component: AddObjectComponent,  
    children: [  
      .  
      .  
      .  
    ]  
  }  
]
```

`canActivate` y `canActivateChild` son dos interfaces que se pueden implementar en un servicio de Angular. Estas dos interfaces van a implementar dos funciones de respectivo nombre y devolverán un booleano.

Este booleano, verdadero o falso, le dirá al sitio web si se puede entrar a la ruta que se esté solicitando (en caso de ser verdadero) o si no se puede (en caso de ser falso).

El problema que han supuesto estas interfaces es que la implementación tenía que realizarse de forma asíncrona, asincronía de consultas que se han utilizado en la API Rest pero que no se habían utilizado en el sitio web.

Gracias a este retraso que ha llevado poder asegurar las rutas de la aplicación se investigaron los Resolvers, los cuales son unos servicios que se pueden implementar en documentos de rutas cuando se necesite recabar una información antes de inicializar el componente de Angular. El problema era que la inicialización del componente de Angular no conllevaba la inicialización del autenticador de ruta, servicio que se ejecutaba antes.

Después se investigaron los *Observables* y las *Promises*.

Observable

- Es una implementación para eventos que conlleven una carga continua a lo largo del tiempo. Por ejemplo la reproducción de un vídeo.
- Tiene atributos que permiten realizar nuevamente solicitudes que han devuelto un error o de las que se han perdido información.
- Pueden solicitar datos en varias pipelines al mismo tiempo.

Promise

- Una promise puede manejar solamente un elemento.
- Devuelve solo un único valor.
- Puede manejar también errores aunque sin un control exhaustivo como un Observable.

Al observarse las diferencias entre estos dos elementos se decidió utilizar un elemento de tipo “Promise”.

El segundo contatiempo ocurrido fue la creencia de que en la interfaz de `canActivate` y `canActivateChild` únicamente se podía devolver un valor booleano. Pero también pueden devolver una Promise del tipo booleana por lo que de esa forma el problema de la asincronía ya estaba resuelto.

El AuthGuard implementaría la siguiente función:

```
checkLoginPromise(): Promise<boolean> {  
  const promise = new Promise<boolean>((resolve, reject) => {  
    this.loginS.getUser()  
      .toPromise()  
      .then((res: any) => {
```

```
        // Success
        this.usuario = res;
        resolve(true);
    },
    err => {
        // Error
        this.logout();
        reject(false);
    }
  );
});
return promise;
}
```

Se puede comprobar que los dos valores que puede devolver la promesa son verdadero o falso. En caso de que sea verdadero el usuario quedará cargado en el sitio web para poder gestionar algunos estilos que requieren sus datos dentro de ella.

Y por último están los dos métodos `canActivate` y `canActivateChild`:

```
canActivateChild(): Promise<boolean> {
  return this.checkLoginPromise();
}

canActivate(): Promise<boolean> {
  return this.checkLoginPromise();
}
```

3.2. Copias de seguridad

La generación de una copia de seguridad automatizada puede parecer un aspecto secundario en muchos desarrollos. Una copia de seguridad sin planificación puede realizarse cada dos meses, cada semana, cada año pero lo que es seguro es que no se hace de una forma precisa, ordenada y organizada.

Uno de los principios básicos de UAL Inventarium es la del almacenaje de información por lo que es de vital importancia la seguridad que presenta ante eventos indeseados.

Los requisitos de la implantación del sistema de copias de seguridad debe ir en relación a los principios del proyecto, tales como modularidad o escalabilidad. Por eso este sistema de copias también tenía que ser creado desde un contenedor de Docker.

Todos los sistemas de copias de seguridad giraban en torno a funcionalidades que presentaba MariaDB. El más referenciado era la siguiente línea de código:

```
docker exec nombre_contenedor /usr/bin/mysqldump -u root --password=root \\
nombre_de_la_base_de_datos > direccion_copia_de_seguridad.sql
```

En función a esta idea se buscó la forma gracias a Crontab, un programador de tareas que se puede instalar en sistemas Linux, de poder automatizar un script de generación de copias de seguridad y enlazarlo a un fichero externo al contenedor que el usuario pudiera estar controlando en todo momento.

Pero, durante la búsqueda se encontró una imagen en Docker Hub que resolvía estos requisitos a la perfección.

3.2.1. MySQL-Backup

MySQLBackup es una imagen que se encuentra subida a Docker Hub que tiene más de 10 millones de descargas por lo que puede ser considerada fiable.

Esta imagen permite programar varias acciones sobre la base de datos:

- Guardar y restaurar copias de seguridad.
- Guardar a un sistema de ficheros local o a un servidor SMB copias de seguridad.
- Seleccionar la/s base/s de datos a copiar.
- Configurar la temporización del generado de copias de seguridad.

Lo siguiente que tenía que hacerse era añadir la configuración de esta máquina al archivo *docker-compose.yml*.

```
backup:
  image: databack/mysql-backup
  restart: always
  volumes:
    - ./backup:/db
  environment:
    DB_DUMP_TARGET: /db
    DB_USER: root
    DB_PASS: secretpass
    DB_DUMP_FREQ: 1440
    DB_DUMP_BEGIN: 2330
    DB_SERVER: db
    DB_NAMES: ualinventory
```

Se le añadió como nombre de referencia al contenedor “backup”. Se referencia la imagen anteriormente citada en Docker Hub, se le añade como parámetro “restart: always” que ayudará a que la máquina se encienda en caso de que se detenga.

En la sección “volumes” lo que se hace es enlazar el directorio *backup* ubicado en la misma carpeta que el fichero *dockercompose.yml* con la carpeta interna del contenedor *db* que será donde se localizarán las copias de seguridad. Los siguientes parámetros se han tratado ya en la configuración de la base de datos pero hay unos nuevos que la imagen da la posibilidad de usar:

- **DB_DUMP_TARGET:** Aquí se escribirá la ubicación donde se almacenarán las copias de seguridad de la base de datos.
- **DB_DUMP_FREQ:** Esta será la variable que regulará la velocidad con la que se crearán las copias de seguridad. Son valores enteros en minutos. Se ha estimado que la base de datos en aproximadamente unos 5 años podría alcanzar un tamaño de 100MB. Por lo que una política de copias de seguridad diaria o semanal puede ser ideal. Dado el poco espacio estimado que va a ocupar se recomienda que la frecuencia de ejecución sea diaria y al llegar al mes se reduzcan las copias pasadas a una copia por semana.
- **DB_DUMP_BEGIN:** Define la hora cuando empezará a efectuarse la copia de seguridad. Por ejemplo, en el caso de que se quieran realizar las copias de seguridad cada día a las 12 del mediodía una frecuencia de 24 horas en cada copia puede depender del momento en el que se inicie el script. Por lo que definir una hora inicial ayuda a poder controlar esa temporización.

- **DB.SERVER:** Aquí se definirá el nombre del servidor de la base de datos de la cual se va a realizar el copiado. Este no es el nombre que tiene definido el contenedor, sino el que se define al inicio de la sección con la referenciación de Docker Compose.
- **DB.NAMES:** Esta variable permite añadir una o varias bases de datos para poder realizar el copiado de los datos. En el caso de que no se defina, el sistema realizará una copia entera de la base de datos.

Con esto ya se tendría implementado el sistema automático de copias de seguridad.

3.3. Servidor de correo

A la hora de desarrollar el servicio de correo tenía que pensar en dos objetivos, primero, ¿cómo implementarlo? y segundo ¿dónde implementarlo?

Lo que tenía claro es que iba a necesitar una dirección de Gmail con la dirección de la aplicación para poder utilizar los servicios de correos que nos aporta Google. En parte la mensajería que ocurriera en la aplicación se podría supervisar desde una entidad superior como es la bandeja de entrada de Gmail. Donde podríamos visualizar todos los envíos realizados e intentar cualquier tipo de incidencia que pudiera ocurrir.

La dirección de Gmail con la que se ha vinculado la API es:

```
ualinventarium@gmail.com
```

También es la dirección con la que está registrado el administrador del sistema dentro de la aplicación.

3.3.1. Nodemailer

Las posibilidades que nos brinda Node JS y el enorme abanico de implementaciones que ha hecho la comunidad con él son increíbles. En un principio el servidor web se realizó en PHP y fue una tarea bastante tuortosa.

Los pasos que tuvimos que hacer para configurar el plugin de Nodemailer fue, primero de todo, instalarlo:

```
npm install nodemailer
```

La implantación del servidor de correo se ha hecho desde la API. Esto es debido a que los complementos de los casos de uso de la aplicación tienen que ir junto a los casos de uso. Por ello los haremos en sintonía con las solicitudes que vayan llegando a la Interfaz de Programación de Aplicaciones.

3.3.2. ¿Dónde implementar esta nueva funcionalidad?

El proceso de añadir nueva funcionalidad a una aplicación puede resultar tedioso. Tienes que tocar casos de uso, modificar secciones de código... En este caso fue bastante cómodo. La API lo facilitó todo en gran medida. ¿Por qué? Por la estructura de carpetas y de desarrollo que hemos hecho en la aplicación, aparte de que la mayoría de los componentes presentan una modularidad bastante alta.

Casos de uso donde se decidió realizar una notificación al usuario:

- **Confirmación de registro:** La confirmación de registro supone dos cosas: la primera es que aumentamos el grado de seguridad frente a posibles registros falsos. Aparte de que ahorramos trabajo a los encargados que tendrían que ir revisando una por una las

solicitudes. La segunda es que podemos garantizar que las personas que se registren posean un email institucional ya que si el dominio del correo no es *inlumine.ual.es* o *ual.es* la aplicación no permite el registro. Eso en sintonía a la confirmación hacen la pareja perfecta.

¿Cómo podemos realizar la confirmación del registro?

La confirmación del registro no supone darse de alta en la aplicación pero sí que supone garantizar tu identidad dentro de la aplicación. Hacía falta generar un estilo de token para poder garantizar que no confirmara su registro cualquier tipo de usuario.

Para poder realizar esto en el momento de registro del usuario, en la confirmación de creación del mismo se manda una solicitud de envío de correo. En esa solicitud se adjunta un enlace que será el que utilice el usuario para darse de alta. Dentro del enlace tenemos, el id del usuario, el número de teléfono y un hash generado a partir del número de teléfono.

Cuando el usuario accede al link para confirmar su registro se mandan estos parámetros nuevamente al servidor. Primero se comprueba que el usuario no esté baneado o haya sido de alta ya. Luego comprobamos si el número de teléfono del usuario es el del usuario (gracias al id) y posteriormente si el hash coincide con el del teléfono.

- **Alta del usuario:** Cuando el usuario es dado de alta por un técnico este recibe una notificación.
- **Concesión de préstamo:** Resulta un proceso tedioso el ir revisando la web cada día por si han aceptado tu solicitud de préstamo. Por lo que se facilita todo bastante si te llega una notificación de correo con la concesión de este.
- **Recordatorio de entrega:** Cuando hay un préstamo activo y este ha expirado damos la posibilidad a un técnico de que mande un recordatorio por correo a un usuario para que devuelva el objeto.

Generación del mailer.js

Dentro de la API he generado un nuevo directorio llamado mails y dentro de este un fichero llamado *mailer.js*. El objetivo de este archivo es el de poder importar nuestro nodemailer e inicializar nuestra clase “transportadora”, esto último lo hacemos de la siguiente forma:

```
var transporter = nodemailer.createTransport({
  service: 'gmail',
  auth: {
    user: 'ualinventarium@gmail.com',
    pass: 'pass'
  }
});
```

El campo “pass” es un token que podemos generar desde Gmail. Para ello tenemos que tener el segundo factor de autenticación habilitado. Al hacerlo, debajo de este nos aparece una sección de autenticación para aplicaciones donde nos permitirá generar una clave que será la que ingresemos en el campo “pass”.

Luego de haber generado este token procedemos a la creación de la función que se encargará de mandar los mails de confirmación de registro:

```
async function registro_creado(direccion, url, nombre) {
  var mailOptions = {
```

```
    from: 'UAL Inventarium',
    to: direccion,
    subject: 'Confirmación de registro',
    html: 'Estimado ' + nombre + ' :<br>Para poder confirmar...
  };

  transporter.sendMail(mailOptions, function (error, info) {
    if (error) {
      console.log(error);
    } else {
      console.log('Email register create sent: ' + info.response);
    }
  });
}
```

Le pasaremos como parámetros la dirección de correo donde mandaremos el correo. La url que utilizará el usuario para confirmar su registro y el nombre del usuario.

Generamos nuestro objeto `mailOptions` con los siguientes parámetros:

- **from:** Que es el campo del remitente, en este caso al usuario le llegará *Correo recibido de UAL Inventarium*.
- **to:** La dirección del destinatario. En este caso la del usuario que se registra.
- **subject:** El asunto del correo.
- **html:** En este caso también me daba la posibilidad de poder ingresar un campo de texto “body”. El problema que presenta body es que la capa de personalización es bastante baja. También es verdad que la capa de personalización que ofrecen los sistemas de correos es muy básica y antigua. Los correos que normalmente envían las empresas, estos que vienen bastante bien decorados que parece que han sido realizados con un modelado punto por punto en realidad son hechos en base a tablas con sus filas y columnas.

Luego de la asignación de campos a `mailOptions` mandamos los parámetros con “sendEmail”. Aquí intentamos capturar cualquier posible error que pueda surgir en el envío del correo.

Estas funciones las llamaremos desde nuestros archivos del directorio service. Siguiendo con el ejemplo del caso anterior, el de la confirmación del usuario, quedaría de la siguiente forma:

```
if (result.affectedRows) {
  message = 'usuario created successfully';
  await transporter.registro_creado(usuario.correoElectronico,
    "http://" + process.env.HOST + "/register-confirmed/" + result.insertId
    + '/' + usuario.telefono + '/' + bcrypt.hashSync(usuario.telefono, 3),
    usuario.nombre);
}
```

Este condicional comprueba que hayamos creado con éxito el usuario. Por consiguiente, se le envía un correo para que confirme su registro. Llamando a nuestra clase `transporter` anteriormente implementada.

Con todo esto, ya tendríamos nuestro servidor de correo implementado.

3.4. Balanceo de cargas

Esta sección creo que merecía estar en la parte final del complemento ya que es una mirada hacia un futuro. Por lo que después de él no hay nada.

La escalabilidad de las aplicaciones web ha conseguido mucha representatividad en los últimos tiempos. Podemos ver sitios web gigantescos y acceder a ellos desde todas las partes del mundo. Tenemos ejemplos como Google, Amazon, Facebook. Empresas que sus sitios webs lo son todo y que sin ellos no hubieran cogido la relevancia que han tenido durante todo este tiempo.

La escalabilidad la considero un concepto de relevancia debido a que los sistemas tienden a crecer y a ampliarse, siendo optimistas claros. No sabemos si en cuarenta años los docentes y estudiantes se habrán quintuplicado en el Departamento de Informática y en sintonía el uso de la aplicación.

Si todo este argumento no convence creo que es mejor recurrir al viejo y clásico: “Mejor prevenir que curar”.

3.4.1. Docker Swarm

Docker Swarm es una funcionalidad que nos brinda Docker que nos permite gestionar los recursos de nuestros contenedores agrupándolos en un cluster.

Esta característica nos ayuda a poder distribuir la carga de trabajo de una aplicación en la red de forma rápida y ahorrando tiempo y recursos de nuestros contenedores. Toda esta gestión se realizaría de forma centralizada.

La arquitectura de Swarm es maestro-esclavo. Es decir, cada clúster está formado al menos por un nodo maestro y tantos nodos esclavos como queramos. Mientras que el maestro se encarga de gestionar el clúster y delegar tareas el esclavo se encarga de ejecutar las unidades de trabajo. Como ejemplo pongamos la COVID-19.

La COVID supuso un aumento de las infraestructuras red en tiempo récord. Tanto supuso este incremento del uso de internet que había varios momentos al día que este dejaba de funcionar durante un tiempo.

Ahora pasemos el ejemplo de infraestructuras red de una página web como la del Aula Virtual de la Universidad de Almería. Esta página implementó un servicio externo para poder realizar videoconferencias que utilizaban bastantes universidades españolas.

En un principio todo iba perfecto, ya que se preparó el sistema para una carga grande pero entonces llegó la época de exámenes y la historia se cuenta sola.

Una herramienta magnífica que nos ayuda a soportar cargas altas o simplemente distribuir recursos para que unas máquinas no se vean sobrecargadas es Docker Swarm.

3.4.2. ¿Cómo podemos implementar Docker Swarm?

Primero de todo necesitamos poder crear otra máquina virtual en nuestra cuenta de Google Cloud. La primera máquina la ubicamos en Estados Unidos pero esta vez la ubicaremos en Finlandia. La configuración la podremos ver en la figura 3.1.

Las dos máquinas virtuales que tendríamos en nuestro entorno Cloud serían las que podemos ver en la figura 3.2.

Procederemos a configurar y a instalarle Docker.

Cuando ya lo tengamos instalado iremos a nuestra máquina principal que queremos que actúe como líder, ejecutaremos el siguiente comando que vemos en la figura 3.3.

```
sudo docker swarm init
```

Esto nos generará un token para poder añadir un nodo a nuestro clúster de contenedores. Para poder ingresar a él escribiremos el comando que nos indica Docker tal como se muestra en la figura 3.4. Con esto ya tendríamos nuestro swarm creado. Podemos añadir tantas máquinas como queramos a ella.

Nombre *
inventarium-2

Etiquetas ?
+ AGREGAR ETIQUETAS

Región *
europe-north1 (Finlandia)

Zona *
europe-north1-a

La región es permanente

La zona es permanente

Configuración de la máquina

Familia de máquinas

USO GENERAL OPTIMIZADA PARA PROCESAMIENTO

Tipos de máquinas para cargas de trabajo comunes, optimizados en función del costo y la flexibilidad

Serie
E2

Selección de la plataforma de CPU según la disponibilidad

Tipo de máquina
e2-small (2 CPU virtuales, 2 GB de memoria)

vCPU
1 núcleo compartido

Memory
2 GB

PLATAFORMA DE CPU Y GPU

Figura 3.1: Creando segunda máquina virtual

<input type="checkbox"/>	Estado	Nombre ↑	Zona	Recomendaciones	En uso por	IP interna	IP externa	Conectar	
<input type="checkbox"/>	✓	inventarium	us-west4-b			alejomaf-inventarium (10.182.0.2) (nic0)	35.219.152.107	SSH	⋮
<input type="checkbox"/>	✓	inventarium-2	europe-north1-a			10.166.0.2 (nic0)	35.228.116.13	SSH	⋮

Figura 3.2: Entorno de máquinas en Google Cloud

```

alejomaf@inventarium: ~/Inventarium
alejomaf@inventarium:~/Inventarium$ sudo docker swarm init
Swarm initialized: current node (jsuarnj2quoikklyp6gu169f7) is now a manager.

To add a worker to this swarm, run the following command:

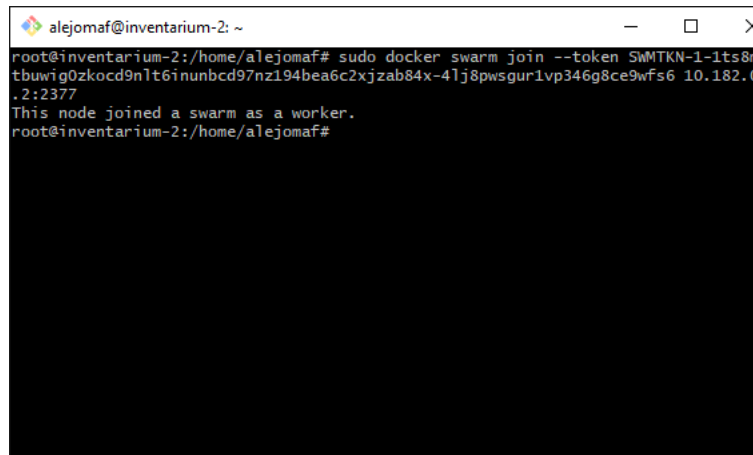
    docker swarm join --token SWMTKN-1-1ts8mtbuwig0zkocd9nlt6inunbcd97nz194bea6c
2xjzab84x-4lj8pwsqur1vp346g8ce9wfs6 10.182.0.2:2377

To add a manager to this swarm, run 'docker swarm join-token manager' and follow
the instructions.

alejomaf@inventarium:~/Inventarium$

```

Figura 3.3: Inicializando el entorno de docker swarm

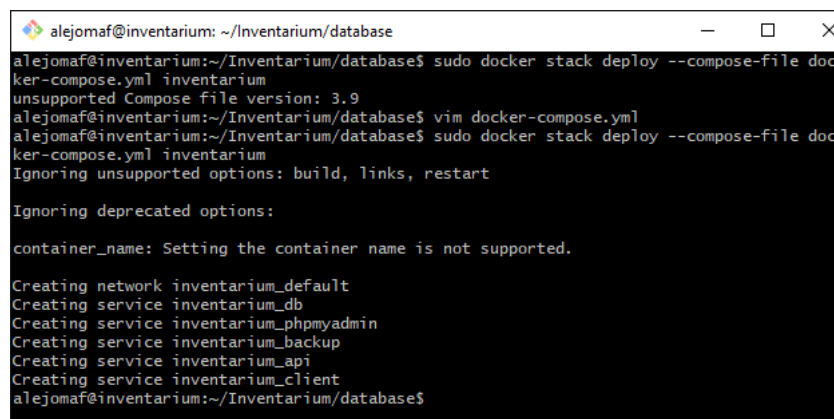


```

alejomaf@inventarium-2: ~
root@inventarium-2:/home/alejomaf# sudo docker swarm join --token SwMTKN-1-1ts8m
tbuwig0zkocd9nlt6inunbcd97nz194bea6c2xjzab84x-4lj8pwsгур1vp346g8ce9wfs6 10.182.0
.2:2377
This node joined a swarm as a worker.
root@inventarium-2:/home/alejomaf#

```

Figura 3.4: Añadiendo nodo esclavo a nuestro entorno



```

alejomaf@inventarium: ~/Inventarium/database
alejomaf@inventarium:~/Inventarium/database$ sudo docker stack deploy --compose-file doc
ker-compose.yml inventarium
unsupported Compose file version: 3.9
alejomaf@inventarium:~/Inventarium/database$ vim docker-compose.yml
alejomaf@inventarium:~/Inventarium/database$ sudo docker stack deploy --compose-file doc
ker-compose.yml inventarium
Ignoring unsupported options: build, links, restart

Ignoring deprecated options:

container_name: Setting the container name is not supported.

Creating network inventarium_default
Creating service inventarium_db
Creating service inventarium_phpmyadmin
Creating service inventarium_backup
Creating service inventarium_api
Creating service inventarium_client
alejomaf@inventarium:~/Inventarium/database$

```

Figura 3.5: Deploy de nuestro entorno Docker en el clúster

3.4.3. ¿Cómo desplegar nuestro entorno en Docker Swarm?

Para poder realizar el despliegue ejecutaremos el siguiente comando dentro de nuestro nodo maestro. Para ello necesitaremos disponer del *docker-compose.yml* que hemos generado en el proyecto. Desde el directorio de la aplicación donde se encuentra ubicado el fichero ejecutamos:

```
sudo docker stack deploy --compose-file docker-compose.yml inventarium
```

Inventarium es el nombre que tendrá nuestro deploy y del que luego haremos referencia. Podemos ver el proceso de creación en la figura 3.5. A partir de este punto surgió una problemática y es que al analizar las máquinas que estaban ejecutándose dentro de nuestro swarm observé que había una que no llegaba a crearse.

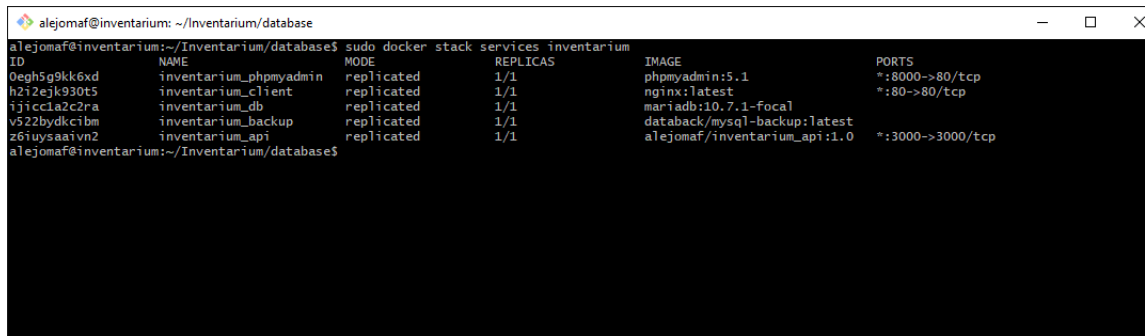
Esta máquina era la de nuestro servidor personalizado con Node. El contenedor con el cual tuvimos que realizar una configuración personalizada con un Dockerfile para que su configuración fuera correcta.

Lo que ocurría era que su máquina al no estar publicada en DockerHub esta no se podía descargar desde ningún lado de nuestro nodo esclavo por lo que teníamos que publicar nuestra imagen.

Primero de todo tenemos que iniciar sesión en Docker con:

```
sudo docker login
```

Luego construiremos nuestra imagen con el Dockerfile, para poder publicarla en DockerHub el nombre de la imagen tiene que seguir la estructura *nombre_usuario/nombre_imagen:version* ejecutando el siguiente comando en la terminal:



```
alejomaf@inventarium: ~/Inventarium/database$ sudo docker stack services inventarium
ID                NAME                MODE                REPLICAS            IMAGE                PORTS
0egh5g9kk6xd     inventarium_phpmyadmin replicated           1/1                 phpmyadmin:5.1      *:8000->80/tcp
h21zejk930t5     inventarium_client  replicated           1/1                 nginx:latest        *:80->80/tcp
ijiecia2c2ra     inventarium_db      replicated           1/1                 mariadb:10.7.1-focal
v522bydkcibm     inventarium_backup  replicated           1/1                 databack/mysql-backup:latest
z6iuy5aainv2     inventarium_api     replicated           1/1                 alejomaf/inventarium_api:1.0  *:3000->3000/tcp
alejomaf@inventarium:~/Inventarium/database$
```

Figura 3.6: Visualización de los servicios corriendo en Docker Swarm

```
sudo docker build -t alejomaf/inventarium_api:1.0 .
```

Al generar nuestra imagen lo que nos quedaría es subirla a DockerHub y eso lo conseguimos ejecutando:

```
sudo docker push alejomaf/inventarium_api:1.0
```

Volveremos a ejecutar nuestro deploy y obtendremos el maravilloso resultado de la figura 3.6.

4 Conclusiones y mejoras

Creo que gran parte de la implementación del complemento de trabajo fin de grado la han facilitado herramientas y soluciones fantásticas que ofrece la inmensa comunidad de internet.

Dentro de la implementación del sistema de copias de seguridad el poder haber encontrado aquella imagen en Docker Hub ayudó en enorme medida a la creación de la solución.

El resto de implementaciones aunque en verdad se basan en todas las modificaciones correspondientes que les pude haber realizado, estas no serían nada sin aquellos frameworks o plugins que me han facilitado la tarea.

Se me han ocurrido bastantes mejoras a medida que iba redactando el documento pero las dos principales y que veo con mayor consistencia son:

Mejora de la gestión de las copias de seguridad

Podría mejorarse la gestión de las copias de seguridad sin que se tenga la necesidad de que tenga que intervenir un técnico en ellas. En la redacción de la sección comenté que sería conveniente ir realizando una copia diaria de la base de datos y que a final de mes podrían descartarse la mayoría de las copias de seguridad y dejar una por semana.

Todo esto iba fundamentado al espacio que ocuparían tales ficheros.

Una mejora sería automatizar este proceso para que no conlleve la revisión mensual de un técnico.

Mejora de las interacciones con el gestor de correo

La siguiente mejora es un poco más compleja. Pensé en una personalización del usuario “técnico” para que este recibiera correos cada vez que se solicitaba un objeto.

El problema ocurría que, primero de todo no podía enviar un correo a todos los técnicos cada vez que se solicitara un objeto debido a que esta cantidad puede llegar a ser bastante grande y, segundo, tendría que modificar el dominio para añadir un nuevo atributo a un grupo de objeto llamado “coordinador”. Que implicaría tener que remodelar bastantes aspectos de la interfaz de usuario para poder utilizar aquella funcionalidad.

A partir de este grupo de implementaciones creo que la página web que se nos presenta dispone de una configuración bastante profesional pero, sobre todo, y más importante: segura, fiable y cómoda de utilizar.

Referencias

Chenkie, R. (24 de julio de 2017). *Angular authentication: Using route guards*. Descargado de https://medium.com/@ryanchenkie_40935/angular-authentication-using-route-guards-bf7a4ca13ae3

Mysql backup. (s.f.). Descargado de <https://hub.docker.com/r/databack/mysql-backup>

Olubisi, I. (15 de junio de 2021). *How to build an authentication api with jwt token in node.js*. Descargado de <https://www.section.io/engineering-education/how-to-build-authentication-api-with-jwt-token-in-nodejs/>

Rachel, T. B. (25 de enero de 2021). *How to use nodemailer to send emails from your node.js server*. Descargado de <https://www.freecodecamp.org/news/use-nodemailer-to-send-emails-from-your-node-js-server/>