# 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 motivmo 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 comercialiaron 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? Nuestros 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

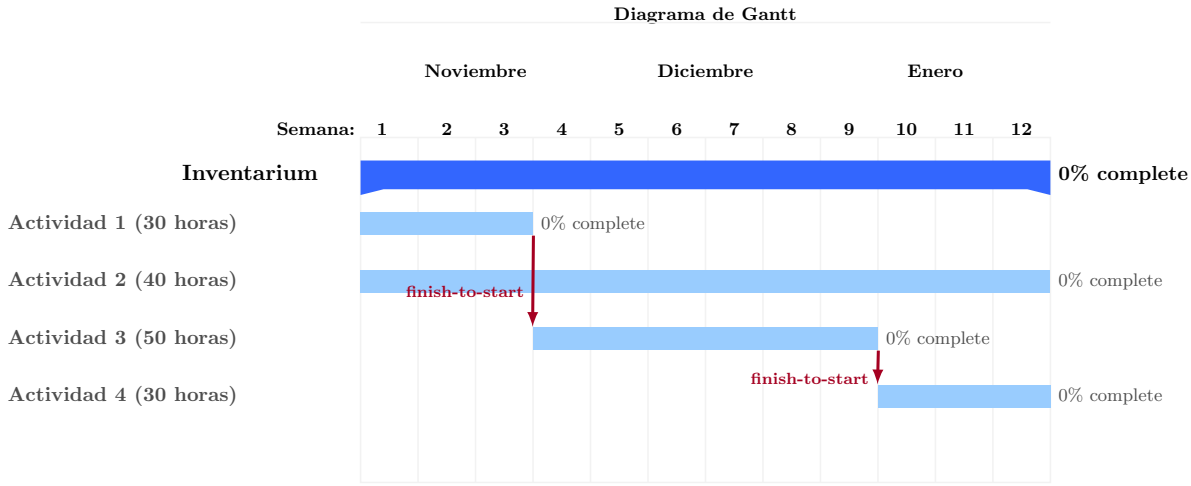
En el proyecto hablé sobre Docker y todas las utilidades que nos podía brindar. En este caso hablaremos de Docker Swarm, un sistema que implementa Docker para poder realizar un balanceo de carga en nuestra aplicación y que esta no se vea sobrecargada. Ideal por si ocurren caídas en algunas de nuestras 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 generado 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 hablaré sobre las herramientas utilizadas durante el desarrollo del proyecto. Algunas de ellas ya las habré 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 Torre de PC

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 Un 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

Nuestro entorno de desarrollo y desde donde haremos casi absolutamente todo será desde Visual Studio Code. Este es 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 descompuestas, me puede.

### 2.2.3 Google Cloud

Volveremos a usar Google Cloud pero esta vez no para el deploy de nuestra página web. Ya que ya lo hemos hecho. Sino que será para la generación de un entorno red donde desplegaremos una o dos máquinas virtuales más. Esto lo haremos con el objetivo de generar una unión entre ellas y poder utilizar una de las herramientas que nos incorpora Docker.

### 2.2.4 Docker Swarm

Docker Swarm nos permite tener varios contenedores Docker interconectados entre sí en diferentes máquinas virtuales o físicas.
Es decir, Docker Swarm nos permite la gestión de un clúster de servidores Docker. Además nos 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 te antes te veías sometido por hacer aplicaciones que gestionasen sus propias consultas con la base de datos.
A medida que iba realizando la construcción de la aplicación pensaba en esta implementación de seguridad.
En un principio lo que hice fue almacenar 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" podríamos llegar a considerarlo 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 nuestra API había algunas herramientas que nos podrían ayudar.

### 3.1.1 Generación del token

La generación de tokens siempre es un proceso que me ha plasmado dudas, ya que en un principio pensaba que el almacenamiento de este realizaba sobre la base de datos en vez de en el propio servidor. Por suerte me encontré con el plugin ideal el cual se encuentra en el Node Package Manager, jwt-simple.
Gracias a este plugin podría 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 necesitamos pasarle como parámetros unas fechas que será la duración que tendrán de validez estos "simbólicos" (traducción de token al español).
El proceso de codificado de nuestro token se realizará dentro de nuestro 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 hemos podido corroborar que el usuario ha iniciado sesión correctamente y sin problemas.
Lo que codificamos dentro de nuestra 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 nuestro token generado tenemos que de alguna forma hacer que cada vez que quiera hacer una consulta el usuario pueda hacerlo con su token.

**¿Por qué almacenamos la id del usuario en nuestro payload?**

La id del usuario se almacena porque cada vez que queramos acceder a una consulta siendo clientes podremos meter en nuestra soicitud ese nuevo campo. Esta acción la realizaremos siempre por lo que nos aseguramos disponer de los datos del usuario en cada uno de nuestros 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 podremos denegar 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 nuestro caso nuestra 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 nuestra API aportándonos la comprobación de la validez de su token y su id de usuario.
Dentro del middleware tendremos una única función que correrá en cada momento que el usuario acceda a uno de nuestros 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 realizamos 3 comprobaciones.

#### Comprobación de los headers

Nuestro token irá dentro del encabezado de las solicitudes que hagamos a la API. En concreto con la key "user_token". Si no se incluye este encabezado no podríamos 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 de nuestro header procedemos a llamar a nuestro 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, obtendremos como respuesta en nuestro sitio web el siguiente mensaje: *Invalid token.*

#### Verificación de la validez

Después del segundo paso podemos haber obtenido un payload pero tenemos que comprobar que este no haya expirado. Es decir, accedemos a su variables "expiresAt". En el caso de que lo haya hecho devolerá un error como respuesta con el siguiente mensaje: *Expired token.*

Incorporamos el user id a nuestra solicitud y con esto ya tendríamos implementado nuestro middleware.

### 3.1.3 ¿Cómo añadimos el proceso de verificación a cada consulta?

Gracias a haber hecho nuestro código de forma estructurada y haber distinguido en servicios y rutas con solo añadir una línea a nuestro campo de rutas hará que podamos añadir este paso de verificación intermedio.
Primero importaremos nuestro fichero "middleware" y luego lo llamaremos en el código con la siguiente función:

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

**¿Cómo iniciamos sesión si no tenemos un token?**

La respuesta es muy sencilla y va en relación a la lógica de nuestro archivo de rutas. Al entrar una petición a uno de nuestros endpoints lo que ocurre es que lo hace en forma de barrido en nuestros 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 este primero en la parte de arriba, en medio nuestra comprobación y abajo el resto de consultas que también podemos hacer.

### 3.1.4 AuthGuard

Aunque esta sea la parte final de la primera sección en verdad es el último apartado que estoy redactando de este capítulo, ya que ha sido el más difícil de implementar por la adquisición de nuevos conceptos que antes no conocía.
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 nos comunicaría que el usuario no está logeado.
Angular nos brinda la posibilidad de poder realizar una autenticación dentro del sistema de rutas de la aplicación. Es decir, dependiendo de a qué sitios quiera acceder de la aplicación puedo implementar unas medidas de seguridad u otras.
Esto lo consigo 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 podemos implementar en un servicio de Angular. Estas dos interfaces nos van a implementar dos funciones de respectivo nombre y nos devolverán un booleano.
Este booleano, verdadero o falso, le dirá al sitio web si entro a la ruta que esté solicitando (en caso de ser verdadero) o si no lo hago (en caso de ser falso).
El problema que me han supuesto estas interfaces es que la implementación tenía que realizarla de forma asíncrona, asincronía de consultas que he utilizado en la API Rest pero que no había utilizado en el sitio web.
Gracias a este retraso que me ha llevado poder asegurar las rutas de la aplicación he conocido los Resolvers, los cuales son unos servicios que podemos implementar en documento de rutas cuando necesitemos 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. Por lo que estuve dándole vueltas bastantes horas hasta que me di cuenta del fallo de concepto que había cometido.
Después descubrí 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.

Cuando aprendí las diferencias entre estos dos elementos supe que tenía que utilizar una promise.
El segundo contatiempo y el más largo a mi parecer es que creía que la interfaz de canActivate y canActivateChild únicamente podían 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.
Nuestro 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;
}
```

Podemos comprobar que los dos valores que nos 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 tendríamos 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 las principios básicos de nuestra aplicación es la del almacenaje de información por lo que es de vital importancia la seguridad que presenta esta presente ante eventos indeseados. Todo cabe decir, que desde que empecé a programar, que fue a los 14 años, una sola vez se me ha vuelto un archivo corrupto. La única similitud, a mi parecer, que comparte la informática con la política.
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 busqué la forma gracias a Crontab, un programador de tareas que podemos 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 encontré 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 la he considerado fiable.
Esta imagen nos permite poder varias acciones sobre nuestra 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

A medida que avanzaba en la lectura de toda la documentación que tenían en Docker Hub pude comprobar que toda la personalización que presentaba esta aplicación era gigante. Aparte de tener una perfecta compatibilidad con MariaDB, nuestra base de datos.
Lo siguiente que teníamos que hacer era añadir la configuración de esta máquina a nuestro archivo *dockercompose.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: ualinventarium
```

Le añadimos como nombre de referencia a nuestro contenedor "backup". Referenciamos la imagen anteriormente citada en Docker Hub, le añadimos el parámetro "restart: always" que nos ayudará a que la máquina se encienda en caso de que se apague.
En la sección "volumes" lo que hacemos es enlazar nuestro directorio *backup* ubicado en la misma carpeta que nuestro fichero *dockercompose.yml* con la carpeta interna del contenedor *db* que será donde se localizarán las copias de seguridad. Los siguientes parámetros los hemos tratado ya en la configuración de la base de datos pero hay unos nuevos que la imagen nos da la posibilidad de usar:

- **DB_DUMP_TARGET**: Aquí escribiremos la ubicación donde se almacenarán las copias de seguridad de nuestra 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. He estimado que la base de datos en aproximadamente unos 5 años podría alcanzar un tamaño, estimando bastantes filas de la base de datos, de 100MB. Por lo que una política de copias de seguridad diaria o

semanal puede ser ideal. Dado el poco espacio que estimo que van a ocupar recomendaría 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 el 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 inicies el script. Por lo que definir una hora inicial ayuda a poder controlar esa temporización.

- **DB_SERVER**: Aquí definiremos el nombre del servidor de la base de datos de la cual vamos a realizar el copiado. Este no es el nombre que tiene definido el contenedor, sino el que le definimos al inicio de la sección con la referenciación de Docker Compose.

- **DB_NAMES**: Esta variable nos permite añadir una o varias bases de datos para poder realizar el copiado de los datos. En el caso de no lo definamos el sistema realizará una copia entera de la base de datos.

Con esto ya tendríamos nuestro sistema automático de copias de seguridad en marcha.

## 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 conceción 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.

# References

Biot, M. A. (1962). Mechanics of deformation and acoustic propagation in porous media. *Journal of applied physics*, *33*(4), 1482–1498.

Heinz, M., Carsten, & Hoffmann, J. (2014, March). *The listings package, march 2014.* http://texdoc.net/texmf-dist/doc/latex/listings/listings.pdf. Retrieved 12/12/2014, from `http://texdoc.net/texmf-dist/doc/latex/listings/listings.pdf`