# UNIVERSIDAD DE ALCALÁ

## Escuela Politécnica Superior

### MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

# Trabajo Fin de Máster

## PROGRAMACIÓN REACTIVA CON SPRING BOOT

Óscar Arroyo Nogales

2020/21

# UNIVERSIDAD DE ALCALÁ

## Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA DEL SOFTWARE PARA LA WEB

---

# Trabajo Fin de Máster

## "PROGRAMACIÓN REACTIVA CON SPRING BOOT"

**Autor:** **Óscar Arroyo Nogales**

**Director:** **Salvador Otón Tortosa**

---

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

# Resumen

El objetivo de este Trabajo Final de Máster es conocer la programación reactiva en el marco de trabajo Spring 5 junto con Spring WebFlux y con ello, desarrollar una aplicación con esta tecnología: una aplicación para comerciales autónomos.

La arquitectura de la aplicación está dividida en una base de datos MongoDB NoSQL (no relacional) y dos microservicios, cada uno de ellos con un papel distinto dependiendo de su función: el microservicio *sales-api* hace de servicio RESTFul reactivo para consultar datos y realizar operaciones en la base de datos, y *sales-client*, que actúa como frontend de la aplicación. Es quien llama a *sales-api* dependiendo de las necesidades que tenga en cada momento de interacción con el usuario.

Esta aplicación, que en conjunto recibe el nombre de *Sales* tiene como objetivo servir de herramienta de gestión de facturas, pedidos y clientes para los comerciales autónomos que son representantes de uno o varios proveedores.

*Sales* también contempla un rol de administración que facilita operaciones como la modificación de los datos de un cliente, de un proveedor o asignación de un autónomo como representante de un proveedor. Para contemplar estos roles se ha desarrollado un inicio de sesión o *login* seguro, haciendo uso del marco java Spring Security, que proporcionará seguridad a cada uno de los microservicios que intervienen en la aplicación.

Por último, para el desarrollo de este trabajo se ha utilizado *Kanban* como técnica de gestión visual de flujos continuos de avance y seguimiento de incidencias, implementaciones y sugerencias de diseño.

# Palabras clave

Programación reactiva, Spring, Spring WebFlux, MongoDB, NoSQL, microservicio, RESTFul, Heroku, GitLab, Kanban

# Abstract

The objective of this Master's thesis is to learn about reactive programming in the Spring 5 framework together with Spring WebFlux and, with it, to develop an application with this technology.

The architecture of the application is divided into a MongoDB NoSQL database (non-relational) and two microservices, each of them with a different role depending on its function: the microservice *sales-api* acts as a reactive RESTFul service to query data and perform operations on the database, and *sales-client*, which acts as the frontend of the application. It is the one who calls *sales-api* depending on the needs it has at each moment of interaction with the user.

This application, which is collectively called *Sales*, is intended to serve as an invoice, order and customer management tool for self-employed salespeople who are representatives of one or more suppliers.

*Sales* also includes an administration role that facilitates operations such as modifying the data of a customer, a supplier or assigning a freelancer as a representative of a supplier. To contemplate these roles, a secure *login* has been developed, making use of the java Spring Security framework, which will provide security to each of the microservices involved in the application.

Finally, for the development of this work, *Kanban* has been used as a technique for visual management of continuous progress flows and monitoring of incidents, implementations, and design suggestions.

# Keywords

Reactive programming, Spring, Spring WebFlux, MongoDB, NoSQL, microservice, RESTFul, Heroku, GitLab, Kanban

Agradecimientos

A mi tutor, mi familia y mis amigos.

# ÍNDICE RESUMIDO

# ÍNDICE DETALLADO

# ÍNDICE DE FIGURAS

# INTRODUCCIÓN

## 1.1. Motivación

Hoy en día, el comercial autónomo medio debe depender de la tecnología para llevar al día todos los proveedores para los que trabaja: los clientes de cada uno de ellos, sus pedidos y las facturas emitidas.

A veces el comercial tiene cientos de clientes en cada proveedor, y acordarse de su ubicación, teléfono o incluso el mismo proveedor al que corresponde se complica por momentos.

Llega un punto en el que el papel no es suficiente para manejar la gran cantidad de datos que debe llevar el comercial en cada situación. Además, los comerciales utilizan en gran medida el transporte público y privado para llegar al cliente, por lo que esta información debe ser accesible en un corto período de tiempo para gestionar los datos con sus clientes.

Esta necesidad junto al avance de la tecnología ha propiciado el aumento de aplicaciones web *responsives* para ayudar al usuario en su gestión diaria, evitando el uso del papel y la necesidad de llevar dispositivos de gran tamaño encima.

Sin embargo, estas aplicaciones siempre han ido orientadas a autónomos de pequeñas empresas o *PyMEs:* no se aprecia con facilidad una aplicación destinada al autónomo comercial, ese usuario intermedio que comunica la pyme con su proveedor: así nace *Sales*.

*Sales* busca implementar de forma sencilla y minimalista un sistema de gestión básico para el comercial, accesible desde cualquier dispositivo: desde un ordenador, hasta un dispositivo móvil, gracias al diseño responsivo.

Por estos motivos se desarrolla *Sales* como una aplicación web mediante la programación reactiva; formada por un servicio reactivo RESTful [1] llamado *sales*-api que interactúa con la base de datos no relacional MongoDB [2], y un servicio frontal que alberga el resto de funcionalidad e interfaz de la aplicación llamado *sales-client*.

La aplicación web se desarrolla íntegramente en Java [3] y el *framework* Spring 5 [4]. Para la reactividad, se utiliza el marco Spring WebFlux [5], mientras que, para los aspectos de seguridad, se hace uso de Spring Security [6].

En el punto siguiente se detallará a modo de guía cada uno de los apartados restantes de la memoria.

## 1.2. Estructura de la memoria

En este apartado se describe cada uno de los puntos que forman la memoria del trabajo:

- **Apartado 2 (Estado del Arte):** En este apartado se describen todas las tecnologías que intervienen en todo el desarrollo de cada una de las aplicaciones que forman *Sales*. Desde los marcos de trabajo de Spring Boot, pasando por el marco reactivo Spring WebFlux; hasta la base de datos MongoDB y su despliegue en la nube con el uso de herramientas de gestión online como MongoDB Atlas y de herramientas offline como Robo 3T. Por otra parte, analizaremos las ventajas y desventajas de utilizar la base de datos no relacional de MongoDB.
- **Apartado 3 (Análisis y Diseño):** En esta parte de la memoria se definen todos los requisitos funcionales y no funcionales sobre los que posteriormente se ha desarrollado *Sales*. Por otra parte, se detallará la arquitectura web que se ha seguido, el modelo de datos que compone la base de datos MongoDB y se adjuntará el diagrama de casos de uso, que resume perfectamente cada funcionalidad que puede realizarse en *Sales*.
- **Apartado 4 (Desarrollo):** Durante este capítulo se abordará cada uno de los microservicios para explicar en qué consiste, cómo se ha desarrollado o qué funcionalidades tiene. En el caso de *sales-client* se procederá explicando en base a funcionalidades como *login*, *rol de administración, rol de usuario, conexión con sales-api,* etcétera; mientras que en *sales-api* se explicará a medida que se detalle cada paquete Java que da forma al microservicio.
- **Apartado 5 (Integración y pruebas):** Dado que durante este Trabajo fin de Máster se desplegó cada uno de los microservicios en un entorno real, se explicará brevemente con ayuda de los apéndices que contienen los manuales de montaje, y se detallará la batería de pruebas que se realizó en cada uno de los microservicios con ayuda de personas que probaron los entornos. Tras esto, se dieron de alta una serie de incidencias y mejoras que también se adjuntan en este capítulo y fueron recogidas en el tablero Kanban.
- **Apartado 6 (Resumen y Conclusión):** Una vez terminadas las fases de análisis, diseño, desarrollo e integración y pruebas, finalizaremos el trabajo con un breve resumen y las conclusiones del trabajo realizado. Se expondrán las posibles mejoras para el futuro de los microservicios y por tanto, de *Sales*.
- **Bibliografía:** En esta sección se adjuntarán todas las referencias que se han consultado durante el desarrollo de este trabajo.
- **Apéndices:** Este apartado contiene información útil para terminar de comprender todo el trabajo realizado: manuales de usuario/administrador para las aplicaciones y manual de montaje para desarrolladores.

# ESTADO DEL ARTE

## 2.1 Introducción

En este capítulo denominado **Estado del Arte** se detallará mediante una breve explicación cada una de las tecnologías que han sido utilizadas durante este Trabajo Fin de Máster.

Como primer punto, el objetivo del trabajo era crear una aplicación con programación reactiva y el marco de trabajo Spring WebFlux. Para que el desarrollo aprovechase la programación reactiva era necesario utilizar una base de datos que tuviese soporte a la programación reactiva, con lo que se escogió finalmente utilizar MongoDB.

Para visualizar la información de la base de datos y poder gestionar sus datos, se necesitaba una herramienta de gestión, al igual que existen para las bases de datos relacionales.

Mientras la aplicación *Sales* estuvo en fase de desarrollo sin desplegarse en la nube se utilizó Robo 3T [7] como herramienta de gestión MongoDB.

En cuanto se finalizó la fase de desarrollo y con ello se desplegó la aplicación dando comienzo a la fase de mantenimiento, se utilizó MongoDB Atlas [8] para la gestión MongoDB, debido a que la base de datos en este último caso estaba desplegada en la nube.

Por otra parte, para realizar el despliegue de los microservicios [9] en la nube, había que elegir un proveedor que fuese compatible con las aplicaciones desarrolladas en Java, y finalmente, la opción que se escogió fue Heroku [10].

Finalmente, para la gestión del proyecto se utilizó GitLab [11], mientras que para desarrollar los microservicios se decantó por el IDE IntelliJ [12].

A continuación se explicará brevemente cada una de estas tecnologías y herramientas que intervienen en el desarrollo de *Sales*.

## 2.2 Spring Boot y Spring WebFlux

En este capítulo se explicarán los dos marcos de trabajo con los que se ha llevado a cabo el desarrollo de los dos microservicios que componen este Trabajo Fin de Máster. El primero de ellos está relacionado con la automatización de procesos y configuraciones en el marco Spring, mientras que el segundo está orientado a la programación no bloqueante o reactiva.

### 2.2.1 Spring Boot

Spring Boot [13] y Spring WebFlux son los dos frameworks que se han utilizado durante el desarrollo de los microservicios que forman este Trabajo Fin de Máster.

El primero de ellos nace a partir de la necesidad de ahorrar ciertas configuraciones a la hora de desarrollar aplicaciones.

Cuando se construye una aplicación con Spring Framework [14] se sigue un patrón:

a) Configuración de dependencias mediante gestores de dependencias como Gradle [15] (fichero build.gradle) o Maven [16] (fichero pom.xml): En este Trabajo, se ha elegido Maven.
b) Desarrollo de la aplicación: El desarrollo se ha llevado en el IDE IntelliJ.
c) Despliegue de la aplicación en un servidor mediante la configuración de ficheros (Tomcat [17], Jetty [18], etc.): Como veremos a continuación, se escogió utilizar Netty [19].

Sin embargo, los pasos **a** y **c** pueden evitarse con el uso del marco Spring Boot. Para utilizarlo en el proyecto se tiende a utilizar la herramienta online Spring Initializr [20] , un asistente de la mano de spring que genera un fichero .zip con el proyecto básico y la configuración que se haya solicitado. En la siguiente figura puede apreciarse parte del asistente. Podremos añadir la versión de Spring Boot, así como el lenguaje de programación, el tipo de proyecto (Maven o Gradle para la gestión de dependencias), y las dependencias, de entre las que para este trabajo en particular, habría que añadir Spring WebFlux, pero siempre dependiendo de las necesidades de cada microservicio:

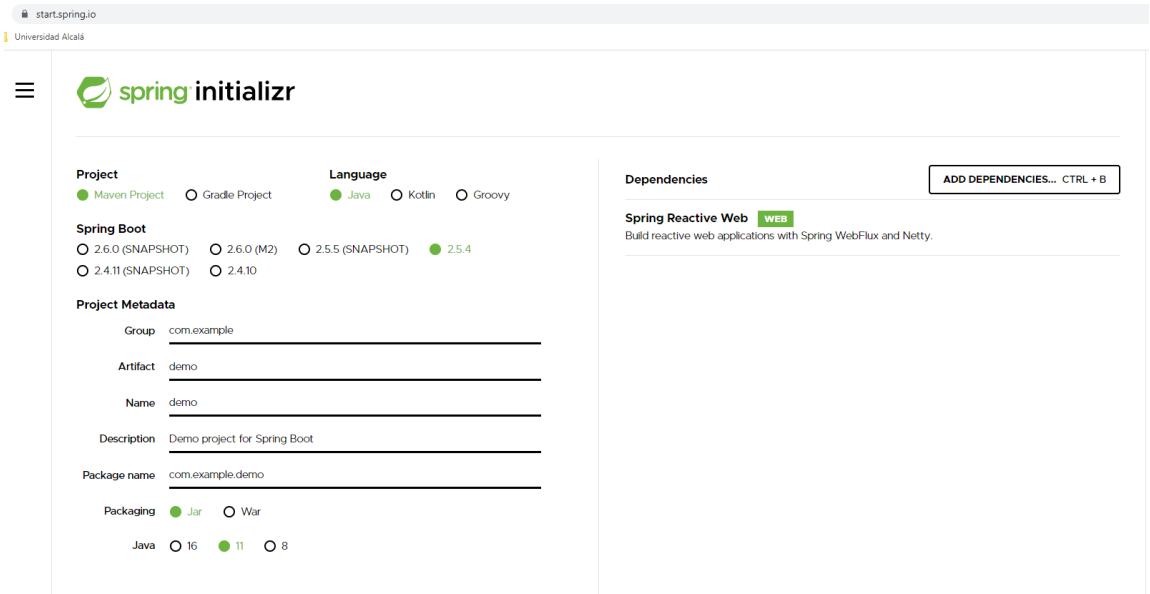


**Figura 1. Spring initializr**

Spring Boot hace uso de los conocidos "starters", que gestionan todas las dependencias necesarias para que la aplicación arranque sin necesidad de que el desarrollador tenga que añadir más configuraciones. A continuación se adjunta una captura del fichero pom.xml, fichero que gestiona Maven para las dependencias:

pom.xml (sales-api)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.5.2</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>es.uah.sales-api</groupId>
    <artifactId>sales-api</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>sales-api</name>
    <description>Sales API</description>
    <properties>
        <java.version>15</java.version>
    </properties>
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-webflux</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-test</artifactId>
            <scope>test</scope>
        </dependency>
```

**Figura 2. Fichero de dependencias pom.xml**

En la figura anterior, además de las versiones de Java utilizadas, se pueden ver cada una de las dependencias que utiliza el proyecto. En particular, *Spring Boot* lleva el **groupId** org.springframework.boot, y a partir de este, pueden incluirse más dependencias del mismo grupo como WebFlux, Boot Starter [21], o MongoDB Reactive [22], todas estas necesarias para el desarrollo de este Trabajo Fin de Máster.

Por otra parte, para desplegar una aplicación en Spring Framework debe definirse el fichero tomcat.xml entre otros para configurar el servidor Tomcat en el entorno de desarrollo. Sin embargo, esto es otro punto a favor de Spring Boot, pues integra Tomcat por defecto.

Con todo esto, se consigue desarrollar una aplicación fácilmente desplegable en el menor tiempo posible, pero para utilizar la programación reactiva [23], debe combinarse con Spring WebFlux.

### 2.2.2 Spring WebFlux

Spring WebFlux nace con la versión del marco Spring 5 como framework no bloqueante, y este punto es realmente el más importante de todo el Trabajo Fin de Máster, pues, ¿qué diferencia hay entre un framework bloqueante y uno no bloqueante [24]? ¿Qué se consigue con WebFlux?

A pesar de que se mostrarán numerosos ejemplos de fragmentos de código de la aplicación para entender cómo funciona el concepto de reactividad, a continuación se explicará de forma sencilla la diferencia principal entre las versiones anteriores de Spring, como la versión cuatro, y la actual, la cinco, utilizando la reactividad que ofrece WebFlux:

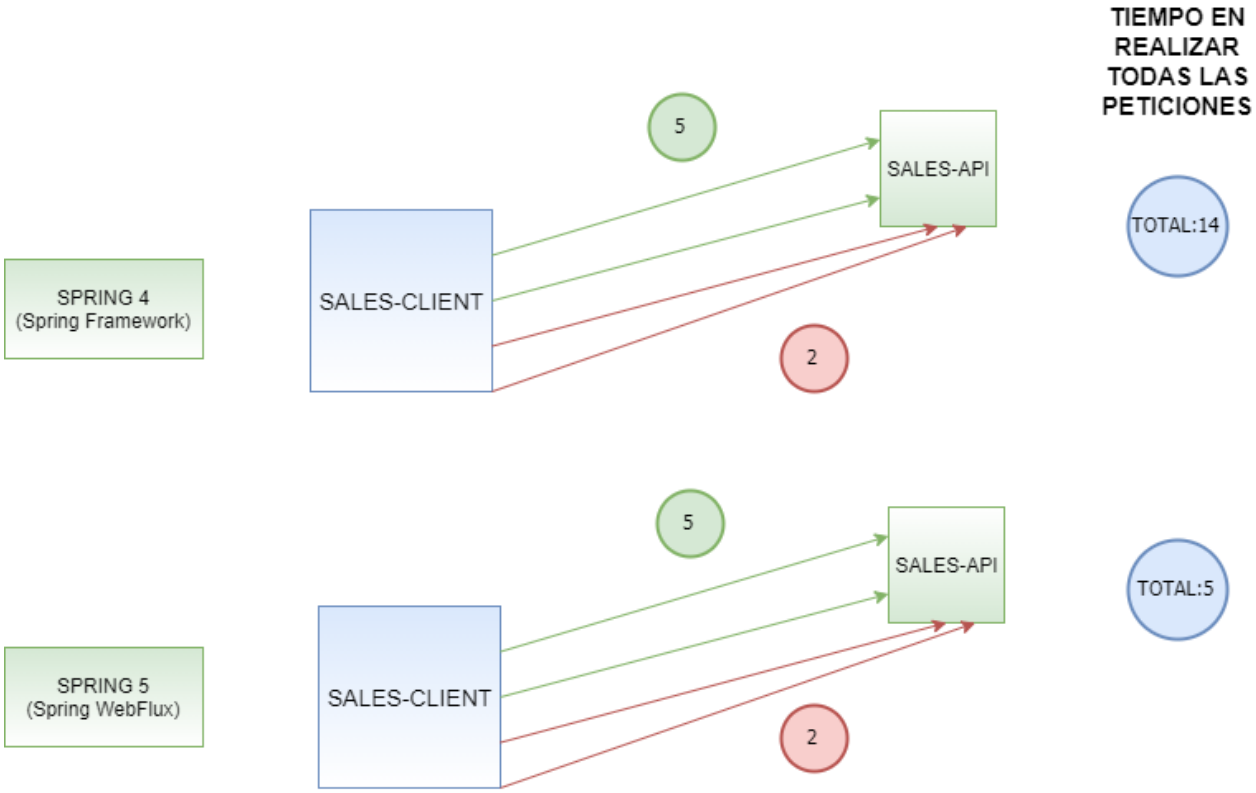



**Figura 3. Spring 4 vs Spring 5 (Reactividad)**

En el primer ejemplo se observa que se trabaja sobre el marco Spring 4. Este framework es bloqueante en el tiempo, lo que significa que cada petición que se realiza desde el microservicio *sales-client* hasta el microservicio *sales-api* en este ejemplo tardan cinco segundos las dos primeras, mientras que las dos siguientes tardan dos. Sin embargo, la petición siguiente no comienza hasta que no termina la anterior. Por este motivo, *sales-client* estaría esperando al resultado de todas las peticiones un total de 14 segundos. Sin embargo, esto cambia cuando se utiliza Spring WebFlux. Si las peticiones fuesen las mismas hacia *sales-api*, ahora tardaría un total de cinco segundos en obtener todos los resultados.

Esto es, porque como se ha mencionado anteriormente, WebFlux trabaja con flujos asíncronos, cada petición que se realiza al api se inicia de forma simultánea junto al resto, reduciendo notablemente el

tiempo total para obtener el mismo resultado. Con esto se consigue manejar la concurrencia [25] utilizando un menor número de recursos.

Aunque Spring WebFlux permita contenedores Tomcat, Jetty o Servlet 3.1+ [26], para este trabajo se ha escogido utilizar un servidor Netty, capaz de gestionar todas y cada una de las peticiones no bloqueantes que le lleguen durante el uso del microservicio. Además de estas alternativas, también puede ejecutarse en contenedores Undertow [27]. A continuación se muestra el log del microservicio *sales-api* donde se aprecia que al arrancarlo, Netty se despliega en el puerto descrito en las propiedades, ya que es el servidor predeterminado del marco WebFlux*:*

```
2021-09-07 19:47:25.376  INFO 14156 --- [localhost:27017] org.mongodb.driver.cluster               : Monitor thread successfully connected to s
2021-09-07 19:47:25.382  INFO 14156 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2021-09-07 19:47:26.593  INFO 14156 --- [  restartedMain] o.s.b.web.embedded.netty.NettyWebServer  : Netty started on port 8080
2021-09-07 19:47:26.605  INFO 14156 --- [  restartedMain] es.uah.salesapi.SalesApiApplication      : Started SalesApiApplication in 4.093 secon
```

**Figura 4. Arranque Netty en *sales-api***

Por último, aunque como ya se ha mencionado, se explicará en mayor detalle en los capítulos siguientes, Spring WebFlux no trabaja con objetos tradicionales, sino con objetos asíncronos.

```
private String direccion;
private Mono<String> direccionMono;
```

**Figura 5. Objeto *Mono***

En la **Figura 5** se muestra la declaración de la variable "dirección" de ambas formas: como objeto del tipo String (**dirección**) y como objeto asíncrono del tipo Mono<String> (**direccionMono**).

Este último objeto pertenece a Reactor [28], la librería escogida por Spring WebFlux para la programación reactiva en Spring 5. Esta proporciona los tipos de dato Mono y Flux [29] (lista de objetos asíncronos) para trabajar con aplicaciones mediante operaciones sin bloqueo admitiendo contrapresión. En la **Figura 6** se ilustra parte del fichero de configuración de dependencias que utiliza Maven, y en concreto muestra la importación de la dependencia de Reactor*:*

```
<dependency>
    <groupId>io.projectreactor</groupId>
    <artifactId>reactor-test</artifactId>
    <scope>test</scope>
</dependency>
```

**Figura 6. Dependencia *Reactor***

Como bien indica la documentación oficial de Spring [30], Reactor no es la única dependencia de Spring WebFlux, sino que el marco de trabajo es compatible con otras librerías reactivas a través de Reactive Streams [31].

Como conclusión de este framework, Spring WebFlux y Spring Boot colaboran para ampliar funcionalidades logrando una aplicación robusta.

## 2.3 Base de datos: MongoDB

Como se ha mencionado en el apartado anterior, MongoDB es una **base de datos basada en documentos**. Esto significa que a diferencia de otras bases de datos relacionales como MySQL [32], MongoDB es no relacional (NoSQL) [33], lo que permite de forma sencilla almacenar y gestionar datos en formato JSON. En la siguiente figura puede comprobarse mediante la herramienta Robo 3T esta disposición de los documentos en formato *JSON*:

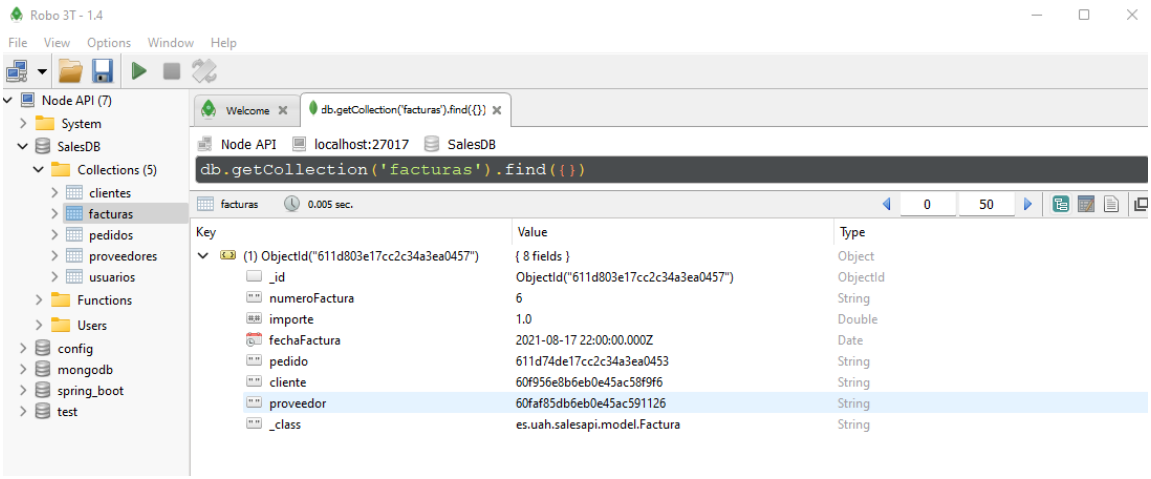



**Figura 7. Ejemplo de documento en MongoDB**

Como se puede apreciar en la figura anterior, en vez de tablas, filas y columnas (como sí ocurre en las bases de datos relacionales), MongoDB utiliza el concepto "colección de documentos". Estas colecciones de documentos (En el ejemplo, la colección **facturas**) constituyen la unidad mínima de información pudiendo almacenar documentos con las propiedades que deseen, o lo que es lo mismo, no es necesario que los documentos de una misma colección sean iguales en relación con su estructura.

Que esta base de datos sea de documentos significa que la estrategia de almacenamiento que se sigue es por **clave-estructura**. Esto quiere decir que cada clave se almacena con una estructura de datos a la que denominamos **documento**. Cada documento puede ser de cualquier complejidad, y en este Trabajo Fin de Máster sigue los esquemas JSON, que coinciden con las clases Java de las distintas aplicaciones que intervienen en *Sales*. En la **Figura 8**, aunque parezca confuso, el elemento (1) no es más que un documento en formato JSON, el cual puede verse en detalle en la siguiente figura:

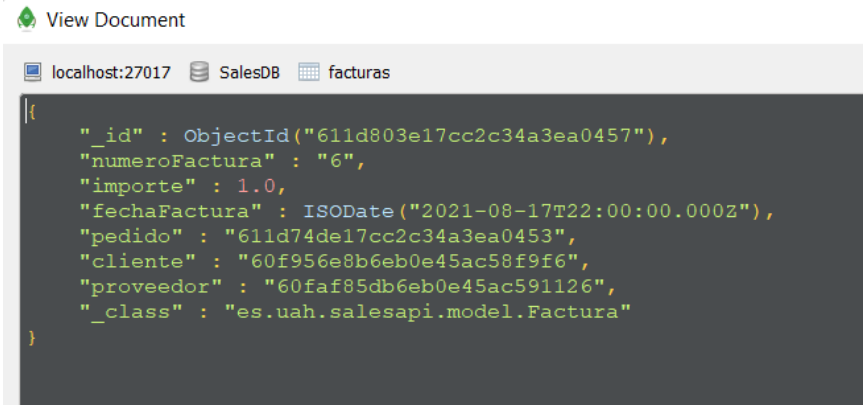


**Figura 8. Visualización de un documento en MongoDB**

Por otra parte, MongoDB guarda una estrecha relación con Java, y facilita en gran medida la incorporación de esta base de datos y su utilización a la aplicación. Mediante la interfaz ReactiveMongoRepository<T, ID> [34] se permite utilizar una colección como repositorio con soporte reactivo. La siguiente figura muestra un fragmento de código donde se utiliza esta interfaz para una colección de documentos, en concreto, la colección denominada "usuarios", que se utiliza en la aplicación para la gestión de los usuarios:

```
public interface IUsuarioJPA extends ReactiveMongoRepository<Usuario, String> {

  Mono<Usuario> findByNombre(String nombre);

  @Query(value = "{'documento.numero' : ?0}")
  Mono<Usuario> findByDocumentoNumero(String documento);
}
```

**Figura 9. IUsuarioJPA**

En este ejemplo, además de poder observar cómo se utiliza la interfaz, se puede ver cómo utilizar lo que se conoce como consultas, a partir de ahora *quer*ies; para por ejemplo como en este caso, poder buscar en un documento a partir de una de sus propiedades, como puede ser el número del documento de identidad.

### 2.3.1 Herramienta de gestión local: Robo 3T

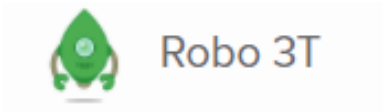



**Figura 10. Robo 3T**

Durante este Trabajo de Fin de Máster se utilizó Robo 3T como herramienta de administración de MongoDB.

Una de sus ventajas es que la plataforma es multiplataforma, permitiendo al desarrollador utilizarla desde cualquier entorno, que como bien indica en su página oficial [7] , consume muy pocos recursos de la máquina.

Robo 3T es Open-source o de código abierto, lo que significa que cuenta con una comunidad muy grande de usuarios que mantienen y mejoran la plataforma, lo que se traduce en que dispone de una documentación muy completa.

Todas las consultas que se realicen desde la herramienta se ejecutan de forma asíncrona o lo que es lo mismo, no bloqueantes en el tiempo.

### 2.3.2 Herramienta de gestión en la nube: MongoDB Atlas

MongoDB Atlas es una base de datos en la nube totalmente gestionada por mongodb.com. A diferencia de otros gestores de bases de datos, este ofrece plan gratuito, el cual vino muy bien para el despliegue de la base de datos en la nube.

Para este Trabajo de Fin de Máster, se desplegó un clúster con tres replica set [35]. Como se observa en la siguiente figura, la interfaz de las colecciones es similar a la interfaz de Robo 3T:

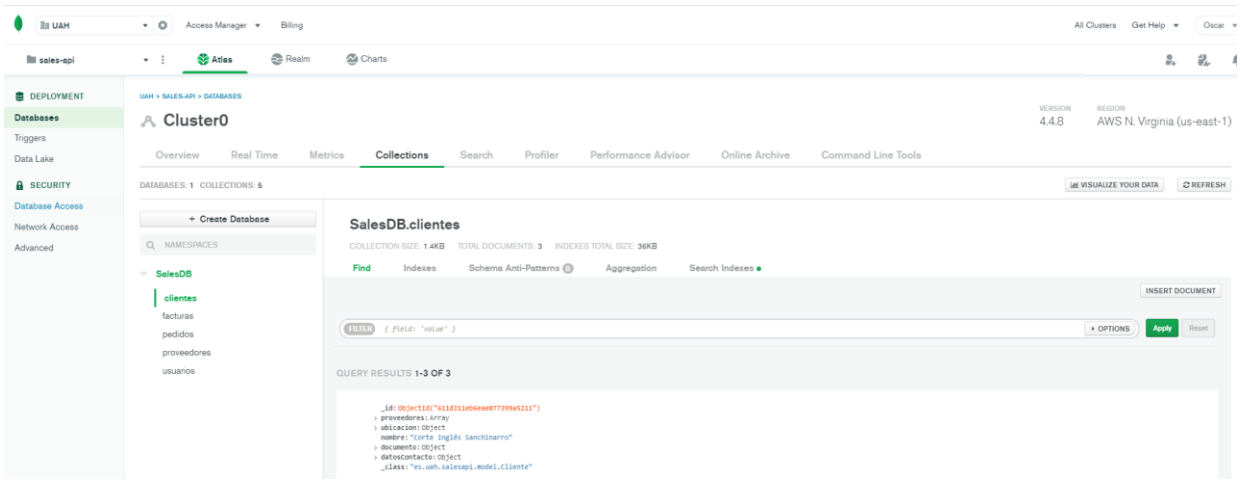



**Figura 11. MongoDB Atlas**

A diferencia de Robo3T, al estar desplegada en la nube, tiene una gran cantidad de opciones de configuración. De entre las que se han utilizado en este trabajo, MongoDB Atlas permite crear accesos de usuarios específicos a la base de datos. Puede especificar el método de autenticación, así como los roles y los recursos a los que puede acceder. La siguiente figura muestra parte del asistente de añadir un usuario de acceso a la base de datos:

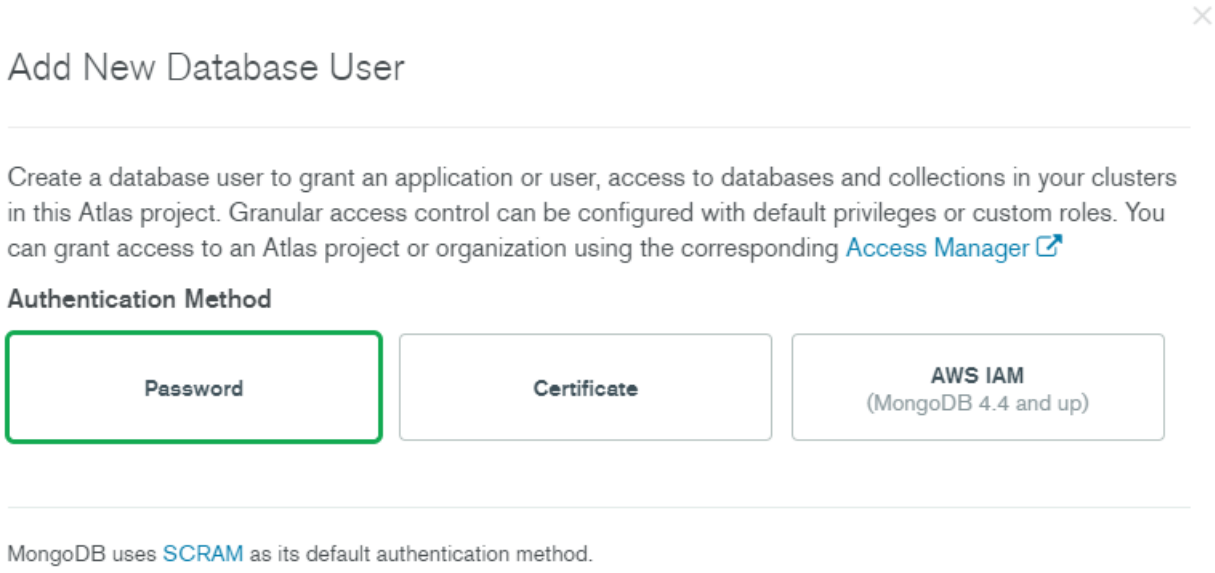



**Figura 12. Añadir nuevo acceso de usuario a MongoDB Atlas**

También destaca la posibilidad de restringir el acceso a la base de datos por dirección *IP*. Esto permite excluir a todos los usuarios cuando se desee realizar pruebas más concretas o permitir a aplicaciones intermedias el acceso a la base de datos introduciendo la dirección estática de esta, la cual suele proporcionarla el proveedor de esta.

### 2.3.3 Ventajas y desventajas

Cuando utilizamos la base de datos MongoDB es evidente que como cualquier otra, esta tiene sus ventajas y desventajas. La siguiente tabla recoge las ventajas y desventajas más relevantes de utilizar bases de datos MongoDB. Estas características se han recogido entre otros, del propio sitio web oficial, mongodb.com:

| Ventajas | Desventajas |
|---|---|
| Permite utilizar un gran volumen de datos estructurados, semiestructurados y no estructurados en constante cambio. | Tiene problemas con las sentencias SQL [36] ya que el sistema de consultas que utiliza MongoDB difiere en numerosos casos. Un ejemplo de consultas que no admite serían las consultas con join. |
| Es una base de datos compatible con las metodologías ágiles [37], ya que no sigue esquemas de bases de datos al permitir al documento almacenar cualquier propiedad. Se utiliza mucho en proyectos BigData [38] debido a que, en la mayoría de los casos, no suele conocerse la estructura de los datos. | No es adecuada para transacciones complejas. En estos casos, se tiende a utilizar bases de datos relacionales. |
| Cuenta con una gran documentación comparada con otras bases de datos NoSQL al ser Open-source. | A pesar de contar con una gran documentación sigue una tecnología muy nueva, con lo que no es tan amplia como la documentación de otras bases de datos, como las relacionales. |
| Programación orientada a objetos, flexible y fácil de utilizar. | |
| La escalabilidad horizontal y vertical es posible al ser una base de datos descentralizada en la nube, a diferencia de las aplicaciones monolíticas. | |
| Alta disponibilidad frente a un gran volumen de acceso. | |

| | |
|---|---|
| El uso de MongoDB es gratuito, a diferencia de otras bases de datos que necesitan la adquisición de una licencia. | |

**Figura 13. Ventajas y desventajas de utilizar MongoDB**

## 2.4 Heroku

Heroku es una *PAAS* propiedad de SalesForce.com, y el concepto *PAAS* significa *Platform as a Service:* es una plataforma como servicio en la nube que permite desplegar proyectos, soportando múltiples lenguajes de programación, entre los que se encuentra Java, que es el que se utiliza en este Trabajo Fin de Máster.

**Figura 14. Heroku**

Entre las ventajas de utilizar esta plataforma destaca la facilidad de uso mediante su cliente, así como los planes de los que dispone, entre los que está el gratuito, para casos como desarrollos casuales o pruebas antes de un despliegue en producción.

A la hora de desplegar aplicaciones, es frecuente el uso de variables de entorno. Esto son variables que utiliza la aplicación y están alojadas directamente en Heroku, desde donde se pueden modificar según la necesidad.

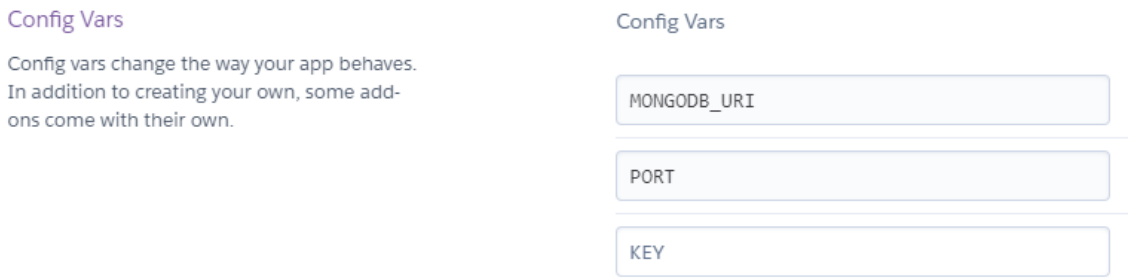



**Figura 15. Configurar variables en Heroku**

En la **Figura 15** se observa la declaración de la variable **MONGODB_URI**, utilizada por el microservicio *sales-api* para conectar con la base de datos MongoDB, mientras la variable **PORT** se utiliza para desplegar la aplicación en el puerto indicado.

Como se ha explicado anteriormente, estas variables se utilizan a su vez en la aplicación en Java para su correcto funcionamiento. En la **Figura 16,** mostrada a continuación, se muestra dónde se utiliza la propiedad **MONGODB_URI**. Al ser una variable de entorno, se recoge en el fichero application.properties:


**Figura 16. Variable de entorno en aplicación Java**

Mediante la propiedad propia del marco de trabajo Spring **spring.data.mongodb.uri** se establece la dirección de la base de datos ubicada en la nube, o donde indique la variable declarada en este caso en Heroku

# ANÁLISIS Y DISEÑO

## 3.1 Especificación de requisitos

A lo largo de este punto se describirán los requisitos funcionales y no funcionales de la aplicación que fueron recopilados durante la fase de análisis. Éstos describen la funcionalidad que ofrecerá el sistema y posteriormente se completará con un diagrama de casos de uso y varios ejemplos de casos de uso.

### 3.1.1 Requisitos funcionales

**R1 – Roles de Usuario**

- El usuario puede tener un rol de comercial autónomo (usuario estándar) o rol de administrador.
- Un usuario estándar no puede modificar su rol, mientras que el administrador sí.

**R2 – Acceso**

- El usuario puede iniciar sesión mediante su documento (DNI, NIE o NIF) y contraseña.
- Se mostrará un mensaje genérico en caso de error durante el proceso.

**R3 – Cerrar sesión**

- El usuario podrá cerrar sesión mediante el enlace correspondiente.

**R4 – Registro**

- Un usuario puede registrarse en la aplicación mediante un formulario de datos personales, datos de contacto y datos de registro, como la contraseña. Por defecto, se completará el registro con el rol de usuario estándar.
- El administrador puede registrar a cualquier usuario una vez inicie sesión en la plataforma. Además, podrá asignarle cualquier rol, ya sea estándar o administrador.

**R5 – Configuración**

- Un usuario podrá modificar sus datos básicos desde el menú "Configuración" de la plataforma.
- A diferencia del usuario estándar, el administrador podrá modificar su rol en este punto.

**R6 – Seleccionar proveedor**

- Cuando el usuario inicie sesión, visualizará una lista de los proveedores de los que es representante.
- Si el usuario no es representante de ningún proveedor, se mostrará un mensaje de advertencia.
- El administrador podrá ver todos los proveedores.

**R7 – Clientes**

- Si el usuario selecciona el proveedor se mostrará un listado de los clientes cuya lista de proveedores contenga el proveedor seleccionado en el paso anterior.

- Si el proveedor no tiene ningún cliente registrado, se mostrará un mensaje de advertencia.
- El usuario podrá ver los datos básicos de cada cliente: documento, teléfono, email, etc. Además, aparecerá un enlace a los pedidos que tenga el cliente.
- Si el cliente tiene definida su ubicación (latitud, longitud), el usuario podrá visualizarla con la ayuda de un mapa.

**R7.1 – Registro**

- El usuario podrá registrar un cliente a partir de su documento.
- Si el cliente está registrado en la plataforma para el documento especificado, traerá la información de este y formará parte de la lista de clientes del proveedor actual.
- Si el cliente tenía datos anteriormente en el proveedor, los traerá una vez se registre de nuevo.
- Si el cliente no está registrado en la plataforma para el documento especificado, se mostrará un formulario que podrá completar el usuario con los datos básicos de la empresa.

**R7.2 – Baja**

- El administrador podrá dar de baja cualquier cliente de un proveedor. Este cliente dejará de formar parte del proveedor actual, pero no perderá ninguna información: pedidos para el proveedor, facturas, etc.

**R7.3 – Modificación**

- El administrador podrá modificar los datos básicos de cualquier cliente, sus proveedores y su ubicación, pero no puede cambiar su documento.

**R8 – Pedidos**

- Si el usuario selecciona los pedidos de un cliente, se mostrará un listado de los pedidos cuyos emisores contengan al usuario actual y que a su vez, sean del cliente seleccionado.
- Si el cliente no tiene ningún pedido registrado, se mostrará un mensaje de advertencia.
- Un pedido puede estar en tres estados diferentes: *Emitido*, *Servido* y *Terminado*.
- El usuario podrá ver los datos básicos de cada pedido: número de pedido (alfanumérico), su estado, la fecha de creación, el IVA aplicado, etc. Además, aparecerá un enlace a las facturas que tenga el pedido. Estas facturas sólo serán accesibles cuando el pedido se encuentre en estado *Terminado*.

**R8.1 – Creación**

- El usuario podrá crear un pedido completando un formulario con los datos correspondientes.
- A la hora de crear el pedido, el usuario será el principal emisor. Sin embargo, puede haber x emisores que podrán ver el pedido. Un emisor es un usuario representante del mismo proveedor actual.

**R8.2 – Eliminación**

- El usuario podrá eliminar cualquier pedido accesible para él. A su vez, eliminará las facturas que contenga.

**R8.3 – Modificación**

- El usuario podrá modificar todos los datos básicos de cualquier pedido que sea accesible para él.

**R9 – Facturas**

- Si el usuario selecciona las facturas de un pedido, se mostrará un listado de sus facturas.
- Si el pedido no tiene ninguna factura registrada, se mostrará un mensaje de advertencia.
- El usuario podrá ver los datos básicos de cada factura: número de factura (alfanumérico), la fecha de creación, el importe con o sin IVA, etc.
- Al final del listado de las facturas aparecerá un aviso con el total del importe con y sin IVA.

**R9.1 – Creación**

- El usuario podrá crear una factura completando un formulario con los datos correspondientes.
- A la hora de crear la factura, esta tomará el IVA del pedido al que pertenece.

**R9.2 – Eliminación**

- El usuario podrá eliminar cualquier factura accesible para él.

**R9.3 – Modificación**

- El usuario podrá modificar todos los datos básicos de cualquier factura que sea accesible para él.

**R10 – Proveedores**

- El administrador podrá acceder a la lista de todos los proveedores de la plataforma.
- Si no existe ningún proveedor, se mostrará un mensaje de advertencia.
- El administrador podrá ver los datos básicos de cada proveedor: documento, teléfono, email, etc.

**R10.1 – Registro**

- El administrador podrá registrar un proveedor a partir de sus datos básicos.
- Si el proveedor está registrado en la plataforma para el documento especificado en el formulario, impedirá el registro y lanzará una advertencia.
- El administrador podrá asignar cualquier usuario de la plataforma como representante del proveedor.

**R10.2 – Eliminación**

- El administrador podrá eliminar cualquier proveedor. Este dejará de formar parte de la plataforma. Por motivos de seguridad, no se perderá la información de los pedidos, facturas y clientes asociados, pero los usuarios representantes dejarán de tener acceso.

**R10.3 – Modificación**

- El administrador podrá modificar los datos básicos de cualquier proveedor, pero no puede cambiar su documento.

### 3.1.2 Requisitos no funcionales

**R1 – Seguridad**

- Se utilizará Spring Security para la seguridad del inicio de sesión. A su vez, las contraseñas generadas irán encriptadas en la base de datos con BCryptPasswordEncoder [39].
- Las comunicaciones externas entre los microservicios tienen que estar protegidas. Es recomendable implementar Basic Authorization [40] con un usuario y contraseñas concretos.

**R2 – Eficiencia**

- Se aprovechará el marco de trabajo Spring WebFlux para programar de forma reactiva, permitiendo que peticiones entre microservicios sean no bloqueantes en el tiempo.
- Los tiempos de respuesta deben ser adecuados, tomando un máximo de cinco segundos.

**R3 – Usabilidad**

- La aplicación debe contar con manuales de usuario y administrador redactados adecuadamente.
- Se debe mostrar mensajes de error cuando estos sucedan durante cualquier proceso de la aplicación.
- La aplicación debe tener una interfaz intuitiva, minimalista, amigable, y de diseño responsivo.
- La aplicación debe contar con internacionalización, como mínimo, de dos idiomas: español e inglés.

**R4 – Producto**

- La aplicación debe desarrollarse en Spring Boot y Spring WebFlux.
- La seguridad de la aplicación debe desarrollarse con Spring Security.
- La base de datos de la aplicación debe ser NoSQL, utilizando preferiblemente MongoDB.
- La arquitectura de la aplicación estará formada por dos microservicios: un microservicio actuará como API REST, y un microservicio cliente.

## 3.2 Arquitectura y diseño de la aplicación

En este apartado se explicará la arquitectura y diseño del conjunto de las aplicaciones que forman *Sales* junto con un diagrama, se detallará el modelo de datos que sigue la aplicación junto a la base de datos NoSQL de MongoDB, y por último, se adjuntará un diagrama de clases donde podrá observarse toda la funcionalidad explicada en el apartado de requisitos funcionales.

### 3.2.1 Arquitectura

A continuación se adjunta el diagrama de la arquitectura simplificada del conjunto de aplicaciones que intervienen en *Sales*. Este diagrama corresponde ya a la fase de mantenimiento, ya que todos los microservicios y base de datos están desplegados en entornos de la nube:

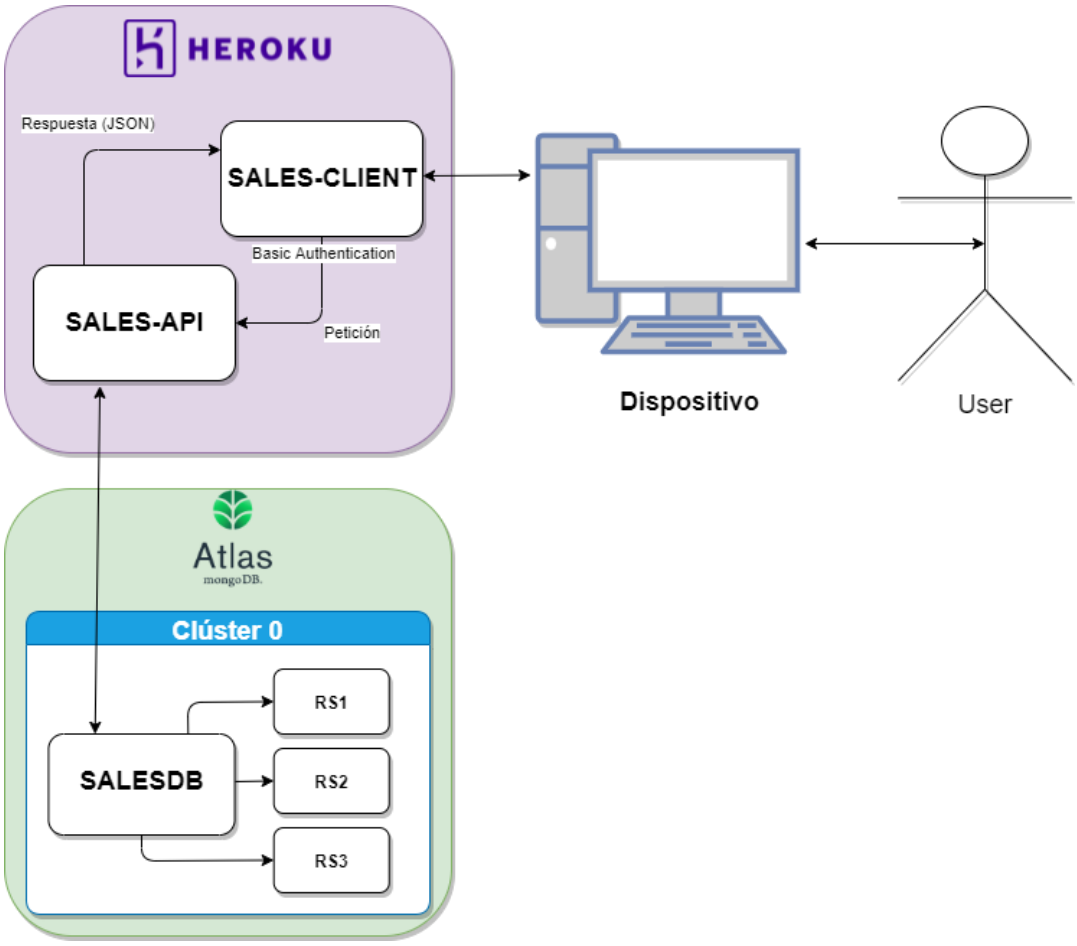



**Figura 17. Arquitectura de *Sales***

Todo comienza con dos entornos bien diferenciados: **Heroku y MongoDB Atlas**. El primero de ellos es quien contiene los dos microservicios desarrollados en Java y Spring: *sales-client* y *sales-api*.

En primer lugar, un usuario entra en la aplicación a través de un dispositivo cualquiera, ya que la plataforma es responsive. Una vez accede, se pide una solicitud al microservicio *sales-client*, quien muestra por pantalla al usuario las distintas interfaces.

En el momento en que *sales-client* necesita de información alojada en la base de datos, es cuando realiza una petición a *sales-api* utilizando **BA o Basic Authentication**, una estrategia de seguridad mediante acceso entre peticiones HTTP, y se explicará en capítulos siguientes.

Una vez la petición es recibida por *sales-api*, esta interactúa con la base de datos **SALESDB,** alojada en un clúster denominado **clúster 0 en MongoDB Atlas,** que a su vez replica la base de datos en tres *replicas set*. Un replica set no es más que una instancia replicada a partir de la base SALESDB, de tal manera que se establecen distintos **backups o copias** de la base de datos principal de forma automatizada.

Cuando SALESDB quiere devolver la información, responde a *sales-api* con ella, y *sales-api* devuelve la información en el cuerpo de la respuesta a *sales-client*.

Por último, *sales-client* termina manejando esta información que necesitaba, bien para mostrarla por pantalla al usuario en su dispositivo, o para realizar alguna funcionalidad.

### 3.2.2 Modelo de datos

El modelo que se ha utilizado para este Trabajo Final de Máster es un modelo no relacional. Aun así, eso no impide crear un esquema que muestre cada una de las entidades que forman la aplicación junto con una breve explicación de las conexiones que existen entre ellas. Uno de los softwares que se suelen utilizar se conoce como **Dataedo [62].** Este programa se utiliza para entre otras cosas, crear diagramas. Como la base de datos es no relacional, las relaciones típicas que se suelen establecer, como por ejemplo, el uso de identificadores de unas colecciones en otras se debe hacer manualmente en la aplicación. De esta manera, el resultado del diagrama que representa el modelo de datos completo de *Sales* es el siguiente:

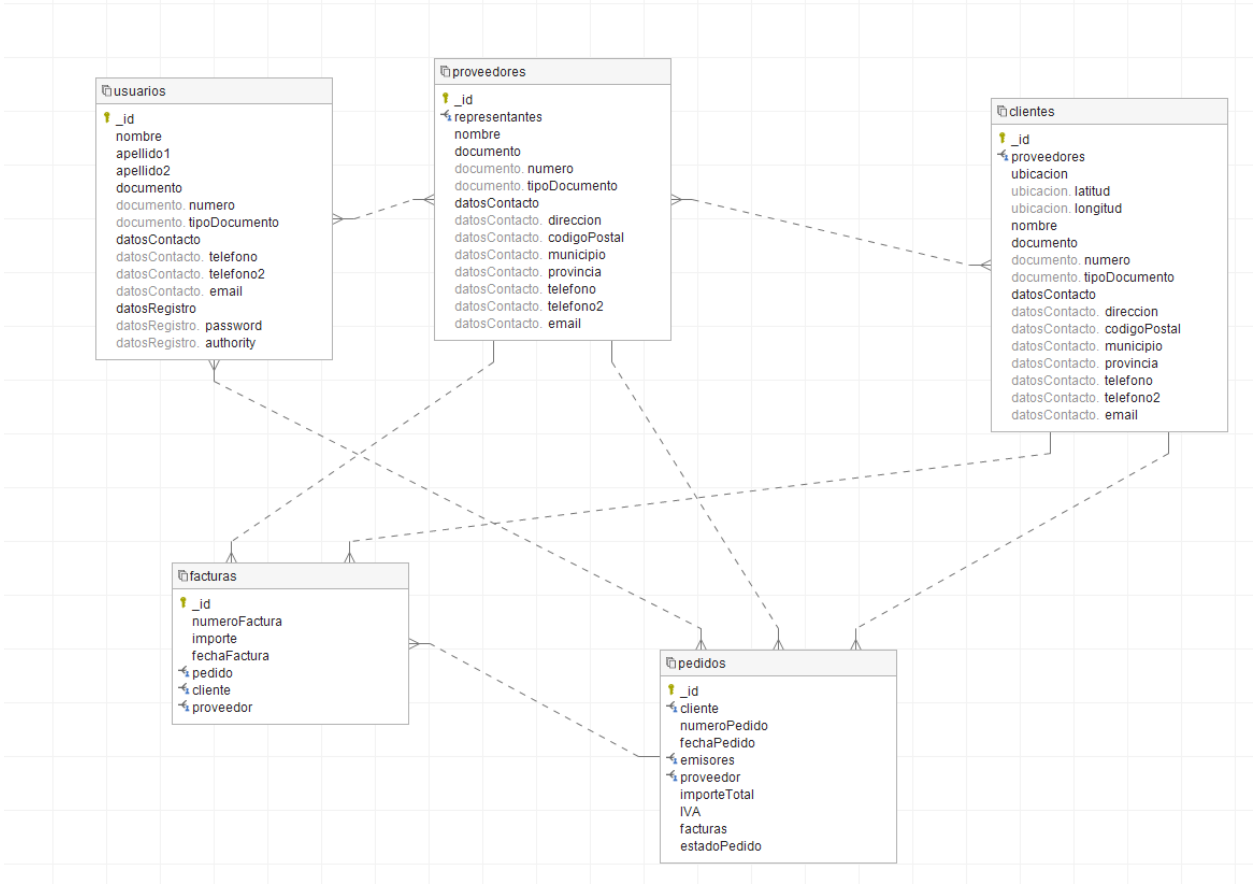



**Figura 18. Diagrama de clases**

Como se puede observar, existen varias relaciones entre las colecciones representadas. A continuación se explica el propósito de cada una de ellas y por qué está relacionada con una u otra:

- **Usuarios:** Representa los usuarios de una aplicación. Tiene datos básicos como nombre, apellidos y **documento**, donde almacena su número y tipo (NIF,CIF,DNI). También tiene una estructura **datosContacto y datosRegistro**, donde se almacenan como bien indica el nombre, datos de contacto y propios del registro respectivamente. Tiene dos relaciones, una con proveedores, ya que cada uno de los representantes que puede tener un proveedor es del tipo Usuario. Y por otra parte, cada pedido puede tener un número determinado de emisores, también del tipo Usuario.

- **Proveedores:** Representa a las empresas que a su vez tienen toda la información de sus clientes, pedidos y facturas. Entre sus propiedades, también tiene una estructura **documento** y **datosContacto**, aunque como se puede apreciar, el tipo de objeto es diferente, pues admite otras propiedades como provincia, municipio o código postal. Las relaciones que mantiene son con los **Usuarios** por lo explicado anteriormente, y con **Clientes**, **Pedidos** y **Facturas**, ya que un proveedor puede tener todo ello. Luego es el usuario como autónomo quien gestiona los datos de cada uno de estos proveedores.

- **Clientes:** Cada proveedor puede tener un cliente. Estos clientes en cuanto a propiedades o datos tienen los mismos que los proveedores, con la única diferencia en que tienen una estructura llamada **ubicación**, utilizada para almacenar la latitud y longitud de su ubicación para la integración con **MapBox**, explicada en capítulos siguientes. La relación que mantiene esta colección es con **Proveedores** por el punto anterior, y con **Pedidos y Facturas,** ya que cada cliente tiene una serie de pedidos y estos a su vez una serie de facturas.

- **Pedidos:** Cada pedido puede tener varias facturas. La información que tiene cada pedido va desde su número, fecha, hasta el IVA aplicado o el estado de este. Las relaciones principales que tiene son con **Clientes** y con **Facturas**. Además, como cada pedido puede tener varios emisores, mantiene una relación con los **Usuarios.** Por último, como todos los pedidos parten de un cliente y a su vez, este parte de un proveedor mantiene una relación con **Proveedores**.

- **Facturas:** Las facturas son la colección más simple en cuanto a información, como puede observarse en la figura del diagrama. Pero en cuanto a relaciones, de forma escalonada como hemos visto en colecciones anteriores, mantiene con **Pedidos**, **Clientes y Proveedores.**

### 3.2.3 Análisis de los casos de uso

Por último se adjunta el diagrama de casos de uso que detalla a la perfección la funcionalidad de la aplicación que se ha descrito anteriormente mediante el análisis de requisitos funcionales y no funcionales.

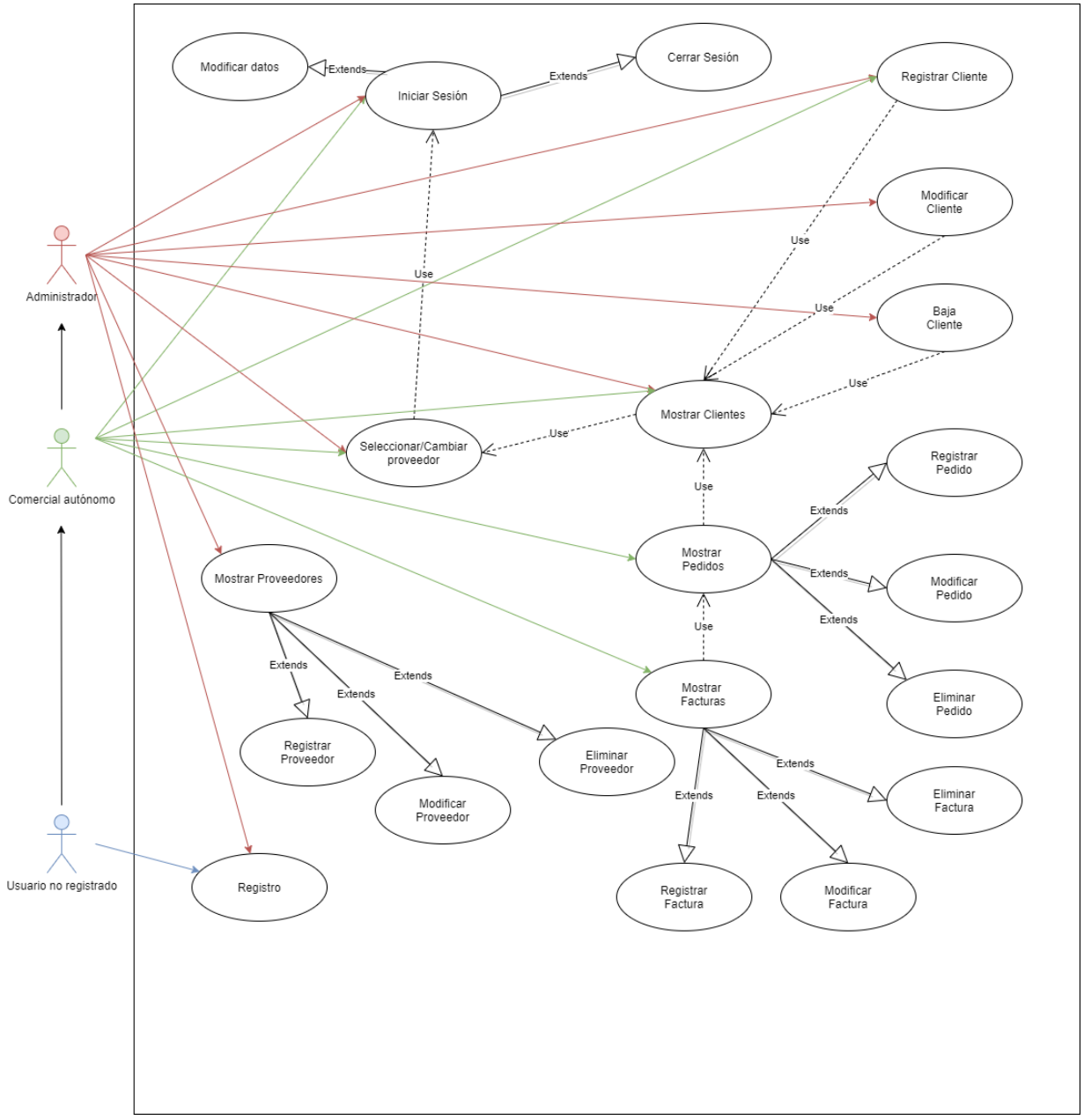



**Figura 19. Diagrama de casos de uso**

Este diagrama resume de forma simplificada toda la funcionalidad posible que recoge *Sales*. Desde lo que puede hacer un usuario que no se ha registrado, hasta lo que puede hacer un comercial autónomo registrado o un administrador. Este tema se explicará de forma visual con guías en **el Apéndice A**

# DESARROLLO

## 4.1 Microservicio *sales-api*

Como se ha comentado en capítulos anteriores, *sales-api* es el microservicio de *Sales* encargado de interactuar con la base de datos MongoDB y proveer de toda la información necesaria al microservicio *sales-client*. Este es, como se apreciará en los siguientes puntos, un microservicio con una estructura bastante extensa, por lo que se explicará cada uno de los paquetes principales Java que componen el microservicio

### 4.1.1 Estructura

En este apartado se realizará un recorrido por cada uno de los paquetes que forman el microservicio y se adjuntará una breve explicación de los aspectos más importantes que intervienen en cada uno de ellos:

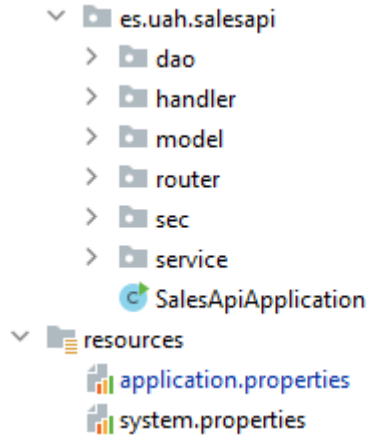



**Figura 20. Estructura de *sales-api***

Cada una de las carpetas corresponde a un paquete Java con distintos propósitos:

#### 4.1.1.1 DAO - JPA

El paquete **dao** contiene las clases e interfaces encargadas de implementar el acceso a la base de datos MongoDB mediante el uso del patrón **DAO** o Data Access Object [41]. También hace uso del Java Persistence API [42] o **JPA** como alternativa para la persistencia sin necesidad de crear consultas, como puede observarse en la siguiente figura:

```
ication.properties  ClienteHandler.java  IPedidoJPA.java  IPedidoDao.java  IFacturaJPA.java  IClienteJP
package es.uah.salesapi.dao.cliente.pedido;

import es.uah.salesapi.model.Pedido;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import reactor.core.publisher.Flux;

import java.util.List;

public interface IPedidoJPA extends ReactiveMongoRepository<Pedido, String> {

  Flux<Pedido> findByProveedorAndClienteIn(String idProveedor, List<String> clientes);
}
```

**Figura 21. JPA de Pedido**

Esta interfaz JPA es la encargada de permitir las consultas sobre MongoDB de la colección de documentos "Pedidos" de la aplicación, representada por Objetos del tipo Pedido. Al igual que esta interfaz, existen otras que representan al resto de colecciones: facturas, clientes, proveedores y usuarios:

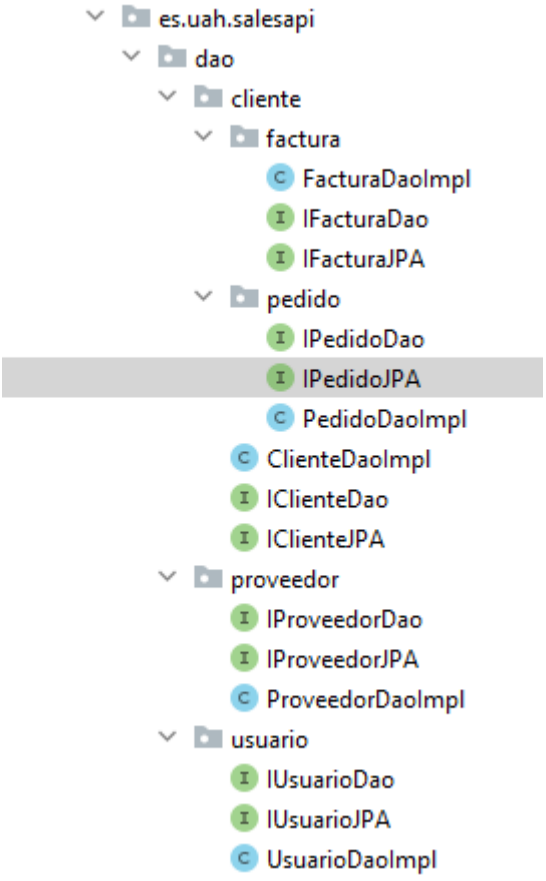



**Figura 22. Estructura del paquete dao**

Cada interfaz JPA hereda como se comentó en el capítulo **Estado del Arte,** de ReactiveMongoRepository<T, ID>, donde *T* es la clase que representa a un documento de la colección

mientras que *ID* es el tipo de la variable identificador del documento, siendo en este caso una variable del tipo String, que se explicará posteriormente en el **paquete model**.

A modo de explicación, el método **findByProveedorAndClienteIn** sigue la nomenclatura JPA. Buscará en la colección de pedidos y propiedades "proveedor" y "cliente", que corresponden al modelo Pedido, como puede apreciarse en la figura:

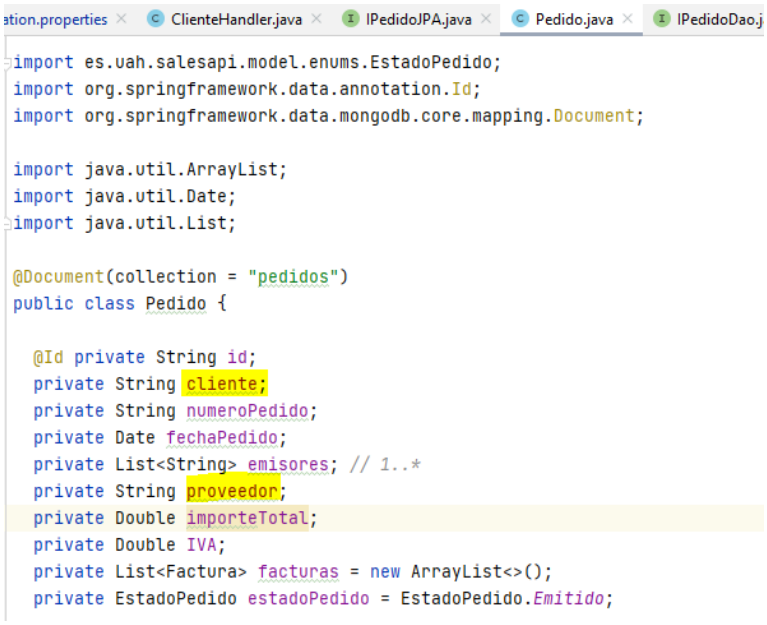



**Figura 23. Fragmento de Pedido.java**

Buscará todos los pedidos cuya propiedad "proveedor" sea la introducida por parámetro y a su vez, que la propiedad "cliente" coincida con alguna de las proporcionadas por la lista de String.

Podrían realizarse funciones aún más simples como, siguiendo la figura anterior, buscar por el número de pedido sería "**findByNumeroPedido**".

También existen métodos que proporciona JPA de forma estándar, como pueden ser: **findAll**, **findById**, **save**, **remove**, y se utilizan mucho para operaciones básicas como crear, eliminar y buscar elementos por su identificador.

Por este motivo se ha implementado cada DAO inyectando la dependencia de su JPA utilizando **@Autowired** [43] para acceder desde los métodos más básicos hasta los más complejos que permite. A estas implementaciones se les añade como cabecera de la clase la anotación **@Repository** [44] para advertir que la clase se utiliza principalmente para la implementación de las operaciones encargadas de interactuar con la base de datos.

En la siguiente figura se muestra cada uno de los detalles mencionados. Corresponde a la clase PedidoDaoImpl.java, encargada de implementar el patrón DAO de la colección de pedidos:

```java
PedidoDaoImpl.java

@Repository
public class PedidoDaoImpl implements IPedidoDao {

  @Autowired
  private IPedidoJPA pedidoJPA;

  @Override
  public Flux<Pedido> findAll() { return pedidoJPA.findAll(); }

  @Override
  public Mono<Pedido> findById(String id) { return pedidoJPA.findById(id); }

  @Override
  public Mono<Pedido> save(Pedido pedido) { return pedidoJPA.save(pedido); }

  @Override
  public Mono<Pedido> update(Pedido pedido) { return save(pedido); }

  @Override
  public Mono<Void> remove(String id) { return pedidoJPA.deleteById(id); }

  @Override
  public Flux<Pedido> findByProveedorAndClienteIn(String idProveedor, List<String> clientes) {
    return pedidoJPA.findByProveedorAndClienteIn(idProveedor, clientes);
  }
}
```

**Figura 24. Implementación del DAO de Pedido**

Al tratarse del framework Spring WebFlux, podemos apreciar cada uno de los tipos que devuelven los métodos que implementa mediante la interfaz IPedidoDAO: flujos de datos asíncronos Flux y Mono. Al igual que esta clase, existen ficheros JPA y DAO para cada una de las clases que representan cada una de las colecciones posibles en la aplicación.

#### 4.1.1.2 Handler y Router

Los paquetes **Handler y Router** contienen las clases livianas que implementan el ApiRest con ayuda Spring WebFlux, utilizando lo que se conoce como Router Functions. La interfaz RouterFunctions [45] tiene su origen en la dependencia de WebFlux **springframework.web.reactive** y posibilita el enrutamiento con programación reactiva. En la **Figura 25** aparece la estructura del paquete Router del microservicio. Como se puede observar, se ha creado una clase "RouterConfig" por cada colección, de tal manera que cada una de ellas maneja todas las operaciones posibles de cada colección:

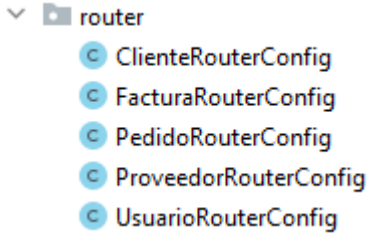



**Figura 25. Paquete Router**

Para cambiar la perspectiva, pasaremos de ver clases relacionadas con los pedidos a visualizar por ejemplo, la clase "RouterConfig" de los usuarios. Este es el aspecto que tiene la clase **UsuarioRouterConfig**:

UsuarioRouterConfig.java

```
package es.uah.salesapi.router;

import es.uah.salesapi.handler.UsuarioHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RequestPredicates;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;

@Configuration
public class UsuarioRouterConfig {

    @Bean
    public RouterFunction<ServerResponse> userRoutes(UsuarioHandler handler) {

        return RouterFunctions.route(RequestPredicates.GET( pattern: "/usuarios"), handler::findAll)
                .andRoute(RequestPredicates.GET( pattern: "/usuarios/{id}"), handler::findById)
                .andRoute(RequestPredicates.GET( pattern: "/usuarios/documento/{documento}"), handler::findByDocumentoNumero)
                .andRoute(RequestPredicates.GET( pattern: "/usuarios/nombre/{nombre}"), handler::findByNombre)
                .andRoute(RequestPredicates.PUT( pattern: "/usuarios"), handler::update)
                .andRoute(RequestPredicates.POST( pattern: "/usuarios"), handler::save);
    }
}
```

**Figura 26. Clase UsuarioRouterConfig**

Esta clase contempla todo el enrutamiento posible de la colección "Usuarios". Contiene todas las rutas a las que llegarán las peticiones relacionadas con los usuarios que posteriormente el componente Handler tendrá que manejar. Para crear una clase de este tipo se empleó el etiquetado **@Configuration,** mediante la cual se indica al marco Spring inicializar todas las configuraciones de la clase; y se etiquetó el **@Bean [46]** en el método **userRoutes.** De esta forma, cuando el microservicio arranque todas las configuraciones, llegará a esta clase, donde establecerá de inicio todas las rutas posibles de los usuarios o la colección que toque.

Mediante el método **RouterFunctions.route** se crea cada una de las rutas posibles. Este método espera un objeto del tipo **RequestPredicate [47],** mediante el cual se indica el tipo de petición (GET, PUT, POST, DELETE …) y su patrón o dirección url.

Como segundo parámetro de entrada se espera un objeto del tipo **HandlerFunction<T extends ServerResponse> [48].** En la **Figura 26** se utiliza la clase **UsuarioHandler**, llamando a diferentes métodos de esta dependiendo del enrutado.

Supongamos que este microservicio está desplegado en la siguiente dirección:

https://tfm-sales-api.herokuapp.com/ .Observando el "RouterConfig" expuesto, ¿cómo podría obtenerse la lista de usuarios? La solución sería https://tfm-sales-api.herokuapp.com/usuarios . Lanzando esta petición en el navegador, aparecería lo siguiente:

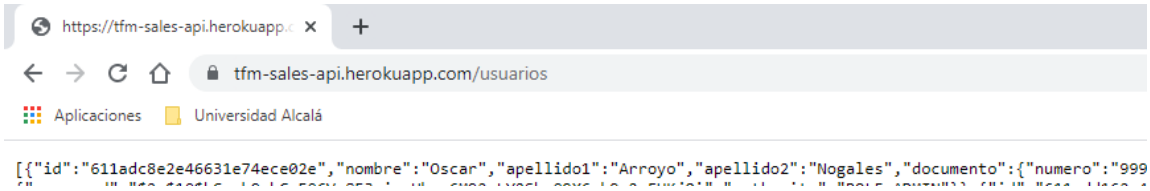



**Figura 27. Ejemplo de petición al api desde el navegador**

A pesar de que se ha recortado la imagen porque aparecía una lista completa de otros usuarios que utilizaron la aplicación, se aprecia como la petición GET descrita en el "RouterConfig" devuelve una lista de objetos en formato JSON. Esta respuesta es emitida por la clase **UsuarioHandler,** concretamente el método **findAll().** En la siguiente figura aparecerá el fragmento de código con este método entre otros:

UsuarioHandler.java

```java
package es.uah.salesapi.handler;

import es.uah.salesapi.model.Usuario;
import es.uah.salesapi.service.usuario.IUsuarioService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.client.WebClientResponseException;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Mono;

import org.springframework.web.reactive.function.BodyInserters;

import java.net.URI;

@Component
public class UsuarioHandler {

  @Autowired private IUsuarioService usuarioService;

  public Mono<ServerResponse> findAll(ServerRequest req) {
    return ServerResponse.ok()
        .contentType(MediaType.APPLICATION_JSON)
        .body(usuarioService.findAll(), Usuario.class);
  }

  public Mono<ServerResponse> findById(ServerRequest req) {
    String id = req.pathVariable( name: "id");
    return usuarioService
        .findById(id)
        .flatMap(
            u ->
                ServerResponse.ok()
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(BodyInserters.fromValue(u))
                    .switchIfEmpty(ServerResponse.notFound().build()));
  }
```

**Figura 28. Fragmento de código de UsuarioHandler.java**

En la figura se observan los métodos **findAll()** y **findById()**, y ambos devuelven un objeto del tipo Mono<ServerResponse>, que como ha sido explicado en anteriores apartados, es un objeto utilizado por el marco Spring WebFlux. FindAll empieza por devolver un OK como respuesta, mientras que mediante el método **contentType** se especifica que la respuesta va a ir en formato JSON. Por último, el método **body** es algo más complejo de entender. Esta es su definición:

```
<T, P extends Publisher<T>> Mono<ServerResponse> body(P var1, Class<T> var2);
```

**Figura 29. Definición del método body de ServerResponse**

Esta definición varía según los tipos que estemos utilizando. Para este caso, la definición sería la siguiente:


**Figura 30. Definición del método body de ServerResponse 2**

Ahora el objeto **P var1** corresponde a un objeto Flux de Usuario, es decir, una lista de flujos de datos del tipo usuario, mientras que la **T** define la clase del objeto que estamos utilizando.

Aunque no se haya explicado aún el contenido del paquete **service**, **usuarioService** corresponde al servicio de la aplicación que maneja todo lo relacionado con los usuarios, y en este caso, está obteniendo todos ellos y devolviéndolos en la respuesta de la petición.

Para terminar con estos dos paquetes de enrutamiento se explicará también a modo de ejemplo el método **save**, un método más complejo que se aprecia en la siguiente figura:

```
public Mono<ServerResponse> save(ServerRequest req) {
  Mono<Usuario> usuario = req.bodyToMono(Usuario.class);
  return usuario
      .flatMap(p -> usuarioService.save(p))
      .flatMap(
          p ->
              ServerResponse.created(URI.create("/usuarios/".concat(p.getId())))
                  .contentType(MediaType.APPLICATION_JSON)
                  .bodyValue(p))
      .onErrorResume(
          error -> {
            WebClientResponseException errorResponse = (WebClientResponseException) error;
            if (errorResponse.getStatusCode() == HttpStatus.BAD_REQUEST) {
              return ServerResponse.badRequest()
                  .contentType(MediaType.APPLICATION_JSON)
                  .bodyValue(errorResponse.getResponseBodyAsString());
            }
            return Mono.error(errorResponse);
          });
}
```

**Figura 31. Definición del método save de UsuarioHandler**

Este método es llamado cuando se realiza una llamada POST sobre el patrón /usuarios. En el cuerpo de la petición se le envía el usuario a guardar y mediante el método **bodyToMono** este usuario pasa a ser una variable reactiva del tipo Mono<Usuario>.

Una vez obtenido, se devuelve este objeto no sin antes realizar una serie de operaciones típicas en el marco de Spring WebFlux. **FlatMap** sirve para emitir el objeto, de forma que esto ayuda a realizar diferentes operaciones, ya que el objeto **p** en la operación *lambda* ya no es Mono, sino únicamente Usuario. En primer lugar se llama al servicio de usuarios para guardar el objeto. El método de dicho servicio devuelve un Mono<Usuario> por lo que para seguir utilizándolo, se realiza un segundo **FlatMap**, donde **p** vuelve a ser un objeto del tipo Usuario.

En este segundo **FlatMap** se crea la respuesta de la petición. Se devuelve la ruta /usuarios/id, donde id es el identificador del usuario que se ha guardado. Mediante el método **contentType**, al igual que en el caso anterior, se le define el tipo del contenido de la respuesta (JSON), mientras que mediante el método **bodyValue** se especifica el objeto que irá en el cuerpo, en este caso, el usuario creado.

En Spring WebFlux es muy habitual utilizar los métodos que manejan excepciones y errores para controlar casos inesperados de la aplicación.

En este método se utiliza **onErrorResume**, donde puede definirse qué devolver de forma personalizada en caso de que durante la ejecución de las operaciones de usuario se haya lanzado algún error.

En primer lugar se recoge la variable **error** y se realiza un casting al tipo WebClientResponseException. Gracias a este casting, podemos ver el código del estado de la respuesta. Si devuelve **HttpStatus.BAD_REQUEST**, el método devuelve un ServerResponse con el código BadRequest y como error, el propio mensaje de error de la respuesta, obtenido a partir del método **getResponseBodyAsString.** Si el error es de código distinto al mencionado anteriormente, el método retorna un Mono.error, que correspondería a un error genérico, con el objeto error.

Como conclusión,  se observa como la aplicación de la programación reactiva es posible en el enrutamiento, y puede ser desde un método muy sencillo hasta todo lo complejo que se desee. Es , como siempre, trabajo del desarrollador decidir cómo de robusta quiere que sea su aplicación.

#### 4.1.1.3 Model

Este paquete es el que tiene todas las clases que representan las colecciones de MongoDB. Como esta base de datos es NoSQL, el tipo de los objetos puede complicarse todo lo que se quiera, ya que las estructuras entrarán en formato JSON a la base de datos sin mucha complicación. La apariencia de este paquete es la siguiente:

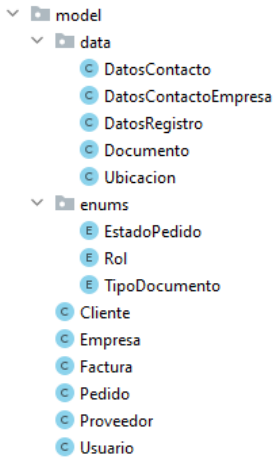


**Figura 32. Paquete model**

El paquete **model** se divide en otros paquetes dependiendo de su utilidad: **data** para estructuras de datos más simples y reducidas que pueden reutilizarse en las clases principales; **enum** para enumeraciones, y la raíz para las clases principales, que como se ha explicado en puntos anteriores, definen cada una de las colecciones de la base de datos MongoDB. A continuación se explicará mediante un pequeño análisis dos de las clases más complejas de *sales-api*, **Cliente y Proveedor**:

Cliente.java | Proveedor.java | Empresa.java

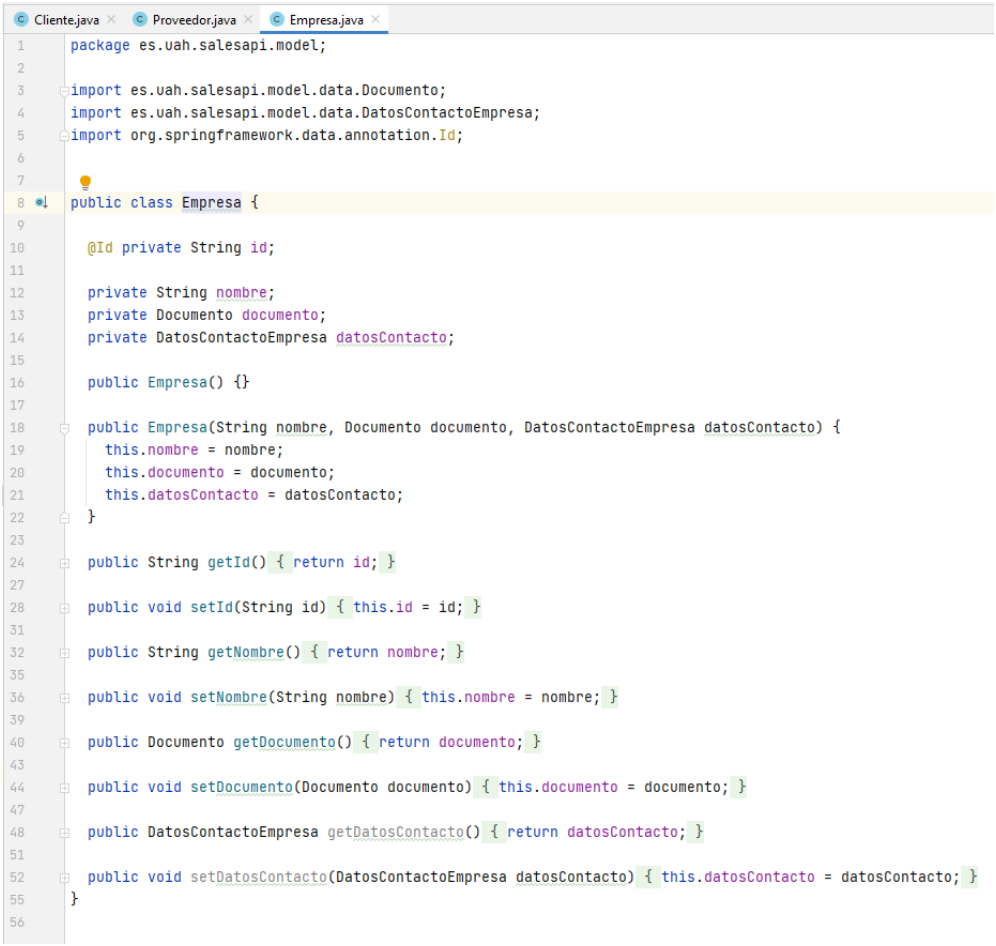
```java
package es.uah.salesapi.model;

import es.uah.salesapi.model.data.DatosContactoEmpresa;
import es.uah.salesapi.model.data.Documento;
import es.uah.salesapi.model.data.Ubicacion;
import org.springframework.data.mongodb.core.mapping.Document;

import java.util.ArrayList;
import java.util.List;

@Document(collection = "clientes")
public class Cliente extends Empresa {

  public List<String> proveedores = new ArrayList<>();

  public Ubicacion ubicacion;

  public Cliente() {}

  public Cliente(
      String nombre,
      Documento documento,
      DatosContactoEmpresa datosContacto,
      List<String> proveedores) {
    super(nombre, documento, datosContacto);
    this.proveedores = proveedores;
  }

  public List<String> getProveedores() { return proveedores; }

  public void setProveedores(List<String> proveedores) { this.proveedores = proveedores; }

  public Ubicacion getUbicacion() { return ubicacion; }

  public void setUbicacion(Ubicacion ubicacion) { this.ubicacion = ubicacion; }
}

```

**Figura 33. Cliente.Java**

En primer lugar, en el marco Spring se utiliza la anotación **@Document** sobre las clases que sean colección de MongoDB. A través del atributo "collection" se determina el nombre exacto de la colección en la base de datos.

La clase **Cliente.java** realmente no es compleja, ya que únicamente tiene dos parámetros propios: la ubicación y una lista de String que contendrá los identificadores de los proveedores que tiene el cliente.

Pero la verdadera particularidad de **Cliente.java** la herencia de una clase más genérica llamada **Empresa.java**, que le da una serie de propiedades que posteriormente observaremos que utiliza también la clase **Proveedor.java**



```java
package es.uah.salesapi.model;

import es.uah.salesapi.model.data.Documento;
import es.uah.salesapi.model.data.DatosContactoEmpresa;
import org.springframework.data.annotation.Id;


public class Empresa {

  @Id private String id;

  private String nombre;
  private Documento documento;
  private DatosContactoEmpresa datosContacto;

  public Empresa() {}

  public Empresa(String nombre, Documento documento, DatosContactoEmpresa datosContacto) {
    this.nombre = nombre;
    this.documento = documento;
    this.datosContacto = datosContacto;
  }

  public String getId() { return id; }

  public void setId(String id) { this.id = id; }

  public String getNombre() { return nombre; }

  public void setNombre(String nombre) { this.nombre = nombre; }

  public Documento getDocumento() { return documento; }

  public void setDocumento(Documento documento) { this.documento = documento; }

  public DatosContactoEmpresa getDatosContacto() { return datosContacto; }

  public void setDatosContacto(DatosContactoEmpresa datosContacto) { this.datosContacto = datosContacto; }
}
```

**Figura 34. Empresa.Java**

Esta clase en concreto implementa la anotación **@Id**, pero no tiene ninguna anotación **@Document**. De esta forma, la clase **Cliente.java** tendrá un campo id con la correspondiente etiqueta **@Id.** Este punto es crucial, ya que mediante este anotado se define la variable en la que se almacenará el identificador de un documento del tipo Cliente o Proveedor.

Empresa contiene las propiedades que tienen en común sus colecciones hijas: un nombre, el documento de identidad, cuya estructura se observa en la **Figura 35**; y los datos de contacto que utilizan el tipo de objeto **DatosContactoEmpresa**, ilustrado en la **Figura 36**.

Cliente.java | Proveedor.java | Empresa.java | Documento.java

```java
package es.uah.salesapi.model.data;

import ...

public class Documento {

  @Indexed(unique=true)
  private String numero;
  private TipoDocumento tipoDocumento;

  public Documento() {}

  public Documento(String numero, TipoDocumento tipoDocumento) {
    this.numero = numero;
    this.tipoDocumento = tipoDocumento;
  }

  public String getNumero() { return numero; }

  public void setNumero(String numero) { this.numero = numero; }

  public TipoDocumento getTipoDocumento() { return tipoDocumento; }

  public void setTipoDocumento(TipoDocumento tipo) { this.tipoDocumento = tipo; }
}
```

**Figura 35. Documento.Java**

Cliente.java | Proveedor.java | Empresa.java | Documento.java | DatosContacto.java | DatosContactoEmpresa.java

```java
package es.uah.salesapi.model.data;

public class DatosContactoEmpresa extends DatosContacto {

  private String direccion;
  private String codigoPostal;
  private String municipio;
  private String provincia;

  public DatosContactoEmpresa() {}

  public DatosContactoEmpresa(
      String telefono,
      String telefono2,
      String email,
      String direccion,
      String codigoPostal,
      String municipio,
      String provincia) {
    super(telefono, telefono2, email);
    this.direccion = direccion;
    this.codigoPostal = codigoPostal;
    this.municipio = municipio;
    this.provincia = provincia;
  }

  public DatosContactoEmpresa(String telefono, String telefono2, String email) { super(telefono, telefono2, email); }

```

**Figura 36. DatosContactoEmpresa.Java**

En conjunto, componen una estructura compleja que se almacenará por cada registro en la base de datos de MongoDB. En la **Figura 37** se puede observar desde la herramienta Robo 3T documentos creados que pertenecen a la colección de clientes:

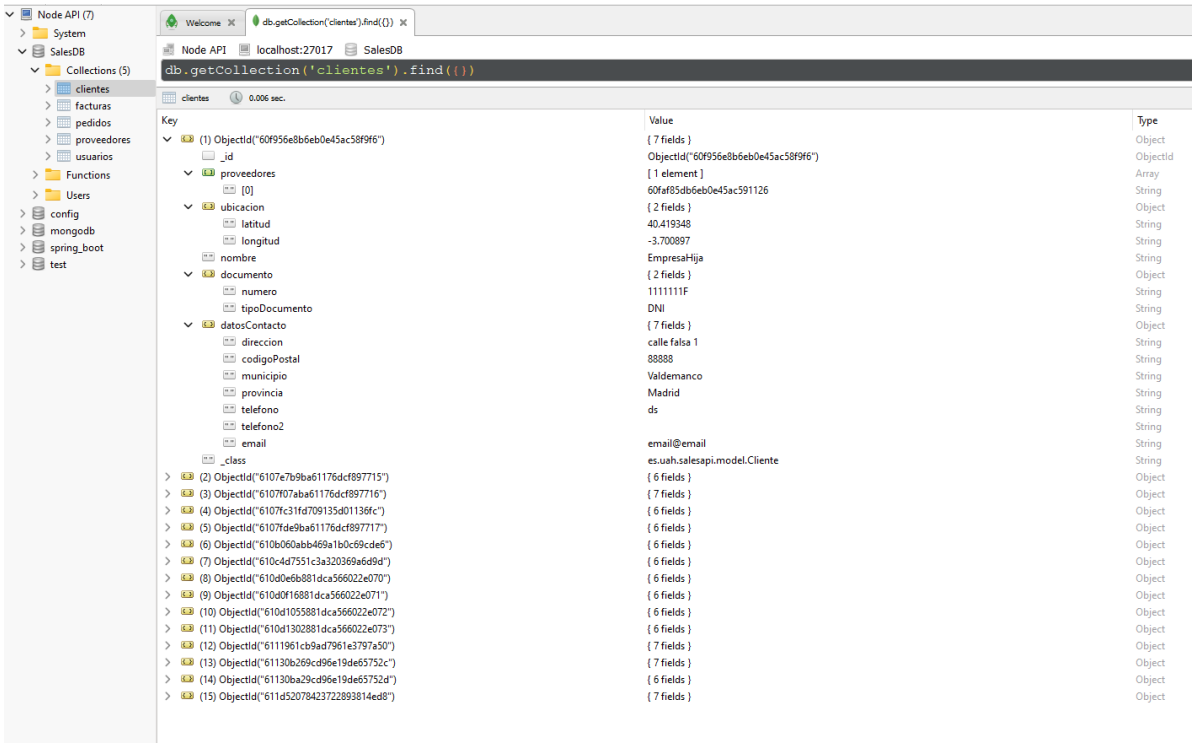



**Figura 37. Colección de clientes en Robo 3T**

Desde el primero de los elementos desplegado se visualiza cada una de las propiedades que se han ilustrado en figuras anteriores y que corresponden a cada clase que actúa como documento de cada colección de MongoDB. A continuación se muestra de forma detallada este primer documento en formato JSON:

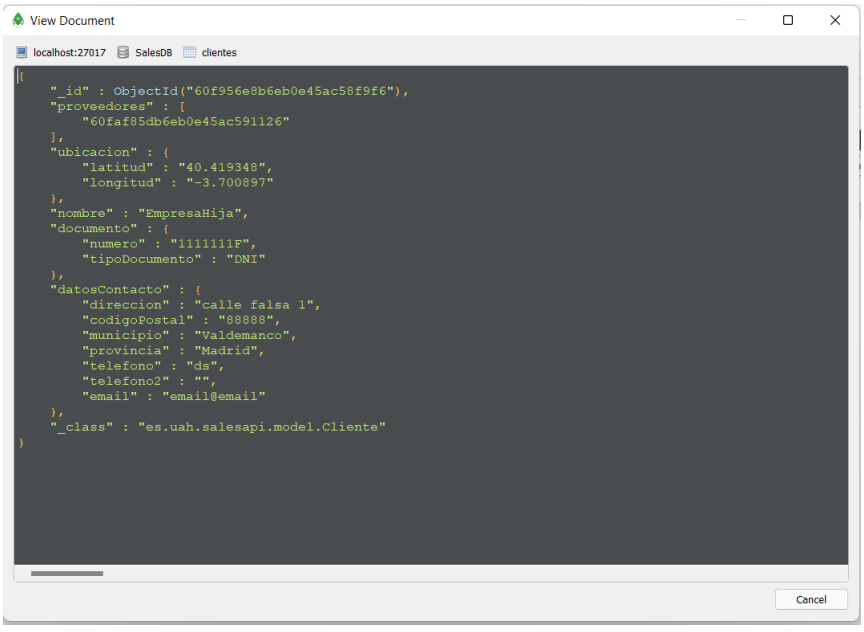


**Figura 38. Documento cliente detallado en Robo 3T**

#### 4.1.1.4 Service

El último paquete relacionado con la gestión de los datos es **service**. Este implementa un servicio por colección, resultando en los siguientes: **FacturaServiceImpl, PedidoServiceImpl, ProvedorServiceImpl** y **UsuarioServiceImpl.** Cada uno de ellos implementa la interfaz que define los métodos del servicio. Respectivamente, son las interfaces **IFacturaService, IPedidoService, IClienteService, IProveedorService** y **IUsuarioService**.

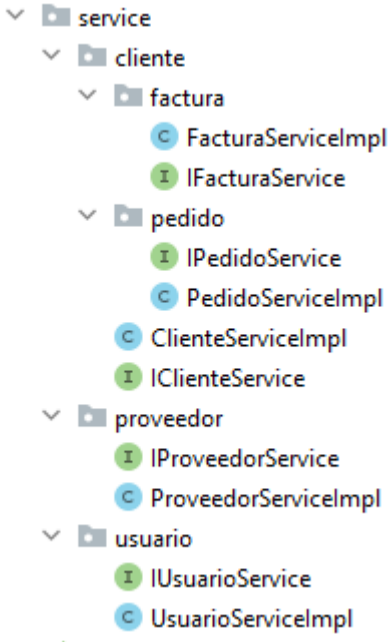



**Figura 39. Paquete service**

En la **Figura 40** se observa cómo la interfaz IProveedorService implementa los métodos que anteriormente aparecían en los paquetes **DAO – JPA:**

IProveedorService.java

```
1   package es.uah.salesapi.service.proveedor;
2
3   import es.uah.salesapi.model.Proveedor;
4   import reactor.core.publisher.Flux;
5   import reactor.core.publisher.Mono;
6
7   public interface IProveedorService {
8
9
10      Flux<Proveedor> findAll();
11
12      Mono<Proveedor> findById(String id);
13
14      Mono<Proveedor> save(Proveedor proveedor);
15
16      Mono<Proveedor> update(Proveedor proveedor);
17
18      Mono<Void> remove(String id);
19  }
```

**Figura 40. IProveedorService.java**

Todas las interfaces mencionadas implementan métodos similares que permiten interactuar con la base de datos para buscar, crear, modificar y eliminar registros.

La clase **ProveedorServiceImpl.java** que implementa la interfaz de la **Figura 40** es muy similar a las clases **DaoImpl** del punto anterior **4.1.1.1,** con la diferencia que en vez de utilizar la inyección de dependencia del JPA, se inyecta mediante **@Autowired** la interfaz Dao de la clase, en este caso, IProveedorDAO, para utilizar los métodos de este en cada método del servicio. La **Figura 41** muestra esta clase:

ProveedorServiceImpl.java

```
1  package es.uah.salesapi.service.proveedor;
2
3  import es.uah.salesapi.dao.proveedor.IProveedorDao;
4  import es.uah.salesapi.model.Proveedor;
5  import org.springframework.beans.factory.annotation.Autowired;
6  import org.springframework.stereotype.Service;
7  import reactor.core.publisher.Flux;
8  import reactor.core.publisher.Mono;
9
10 @Service
11 public class ProveedorServiceImpl implements IProveedorService {
12
13   @Autowired private IProveedorDao proveedorDao;
14
15   @Override
16   public Flux<Proveedor> findAll() { return proveedorDao.findAll(); }
19
20   @Override
21   public Mono<Proveedor> findById(String id) { return proveedorDao.findById(id); }
24
25   @Override
26   public Mono<Proveedor> save(Proveedor cliente) { return proveedorDao.save(cliente); }
29
30   @Override
31   public Mono<Proveedor> update(Proveedor cliente) { return save(cliente); }
34
35   @Override
36   public Mono<Void> remove(String id) { return proveedorDao.remove(id); }
39 }
40
```

**Figura 41. ProveedorServiceImpl.java**

Otra de las diferencias con las clases que implementan el DAO es que, al tratarse de una clase servicio, se anota con **@Service**. De esta forma, el marco de trabajo Spring sabe cómo tratar la clase para que actúe como servicio. Como se ha visto en figuras anteriores, estas clases servicios son las que utiliza el paquete **Handler** para gestionar los datos necesarios en cada petición que se realiza al microservicio *sales-api*.

### 4.1.1.5 Sec

El último paquete está relacionado con la seguridad de este microservicio, y recibe el nombre **sec.** Está compuesto únicamente por una clase llamada **SecurityConfig.java**, que implementa el marco Spring Security para Spring WebFlux. Spring Security proporciona configuraciones de seguridad tales como

autenticación o autorización entre otras. En este Trabajo Fin de Máster se ha incorporado Spring Security mediante el fichero de gestión de dependencias pom.xml que maneja Maven:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    <version>2.3.3.RELEASE</version>
</dependency>
```

**Figura 42. Dependencia de Spring Security**

Esta dependencia trae compatibilidad con el marco reactivo WebFlux, y para su utilización es necesario anotar la clase con **@EnableWebFluxSecurity [49].** De esta forma, se añade el soporte a Spring Security WebFlux. Como esta clase está directamente relacionada con toda la seguridad que se ha desarrollado para este microservicio, en el siguiente punto **4.1.2** se entrará en detalle en la clase y en su puesta en práctica.

### 4.1.2 Seguridad con Basic Authentication

Basic Authentication o BA, en el contexto de una petición, es una configuración de acceso básica a un servicio basada en la autenticación mediante usuario y una contraseña. Estas credenciales son proporcionadas y configuradas en el microservicio *sales-api* para que todas y cada una de sus peticiones necesiten las credenciales para acceder al mismo.

De esta manera, si como en ejemplos anteriores, un usuario quisiese realizar una consulta de todos los usuarios, realizaría la siguiente llamada:

https://tfm-sales-api.herokuapp.com/usuarios

Sin embargo, cuando accede a la petición, por ejemplo, mediante el navegador, aparece la siguiente ventana emergente:

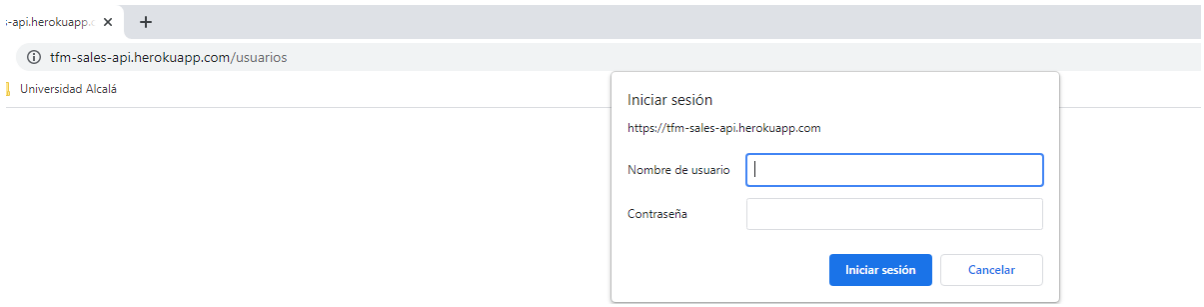



**Figura 43. BA en navegador**

El navegador solicita las credenciales de acceso que fueron establecidas en el microservicio como método de autenticación y acceso para acceder a cada una de sus peticiones configuradas en las clases de enrutamiento.

Toda esta configuración de seguridad es posible gracias a Spring Security y al paquete **sec** explicado en el apartado anterior. Es la clase **SecurityConfig** quien se ocupa de todo ello, y en la siguiente figura se observa el código que compone la clase, destacando la etiqueta **@EnableWebFluxSecurity**:

SecurityConfig.java

```
7  import org.springframework.security.core.userdetails.MapReactiveUserDetailsService;
8  import org.springframework.security.core.userdetails.User;
9  import org.springframework.security.core.userdetails.UserDetails;
10 import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
11 import org.springframework.security.crypto.password.PasswordEncoder;
12 import org.springframework.security.web.server.SecurityWebFilterChain;
13
14 @EnableWebFluxSecurity
15 public class SecurityConfig {
16
17   @Bean
18   public PasswordEncoder encoder() { return new BCryptPasswordEncoder(); }
21
22   @Value("${authentication.username}")
23   public String username;
24
25   @Value("${authentication.password}")
26   public String password;
27
28   @Bean
29 @ public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
30     return http
31             .csrf().disable() ServerHttpSecurity
32             .authorizeExchange() ServerHttpSecurity.AuthorizeExchangeSpec
33             .pathMatchers( ...antPatterns: "/").permitAll()
34             .anyExchange().authenticated()
35             .and() ServerHttpSecurity
36             .httpBasic() ServerHttpSecurity.HttpBasicSpec
37             .and() ServerHttpSecurity
38             .formLogin().disable()
39             .build();
40   }
41
42   @Bean
43   public MapReactiveUserDetailsService userDetailsService() {
44     UserDetails user = User.builder()
45             .username(username)
46             .password(encoder().encode(password))
47             .roles("USER")
48             .build();
49     return new MapReactiveUserDetailsService(user);
50   }
51 }
```

**Figura 44. SecurityConfig.java**

Uno de los puntos más importantes es el **@Bean PasswordEncoder**. Esta interfaz de servicio se utiliza para codificar contraseñas. La implementación que suele seguirse es precisamente la que se ha realizado en este trabajo y puede observarse en la línea 18: **BCryptPasswordEncoder [39]**. Esta implementación utiliza la función de hash de **BCrypt**. De esta manera, si se utiliza esta implementación junto a las credenciales del BA, la seguridad aumentará y si se combina con un protocolo seguro como HTTPS, se hace muy complicado vulnerar la seguridad de este microservicio.

El usuario y contraseña de acceso al servicio se configuran desde el fichero application.properties**,** concretamente mediante las variables **authentication.username** y **authentication.password.** Una de las posibilidades que tiene el marco Spring es precisamente la que se observa en las líneas 22 y 25 de la

figura anterior. Mediante la anotación **@Value [50]** se importan ambas variables a esta clase para posteriormente utilizarlas en el método **userDetailsService**, encargado de implementar la configuración del usuario de servicio. Por defecto tendrá el rol "USER" y configurado el usuario y contraseña anteriormente mencionados. Este método devuelve un **MapReactiveUserDetailsService**, una clase propia del marco Spring Security, pero basado en programación reactiva de Spring WebFlux. Mediante la anotación **@Bean,** ese método se iniciará en el momento de arranque de la aplicación.

Por otra parte, está el método **securityWebFilterChain**, un método bastante estandarizado que recoge las normas de conexión al servicio. Como se puede apreciar en la figura anterior, tiene configuraciones como las siguientes:

- authorizeExchange()
- pathMatchers("/").permitAll()
- anyExchange().authenticated()

Estas dos configuraciones en particular permiten el acceso a cualquier dirección del microservicio **siempre que esté autenticado.**

- formLogin().disable()

Esta configuración deshabilita el formulario de login que por defecto implementa la dependencia de Spring Security.

Con esta configuración tan sencilla, se consigue un microservicio robusto al que únicamente podrán realizar solicitudes aquellos servicios, sitios web, usuarios, etcétera; que tengan ambas credenciales de acceso. En este Trabajo de Máster y como es lógico, el único que tendrá ambas credenciales de acceso será el microservicio *sales-client*.

## 4.2 Microservicio sales-client

Como se ha comentado en capítulos anteriores, *sales-client* es el microservicio de *Sales* encargado de interactuar con el microservicio *sales-api* y proveer de toda la información necesaria al usuario mediante una interfaz web o *frontend*. Al igual que el microservicio, este cuenta con una estructura y una lógica de negocio bastante extensa, por lo que se explicará cada uno de los paquetes principales Java que componen el microservicio de forma breve y se detallará aquello más importante que no se haya visto en apartados anteriores.

Una de las características de este microservicio es que hereda en parte las mismas clases del paquete **model** para su interacción con el usuario, ya que necesita estos objetos para utilizarlos en la lógica de negocio y en el frontal. La diferencia es que el microservicio consulta toda información al microservicio de *sales-api* y recoge estos datos en las clases mencionadas. Estas no tienen por qué ser iguales, y las propiedades que coincidan con las de los objetos del *api* son las que se traerá de vuelta al microservicio.

Primero se hablará de la estructura de forma similar a *sales-api*, pero después de este punto, se mostrarán algunas de las interfaces de la aplicación y se explicará parte de la funcionalidad desarrollada.

Es importante mencionar que debido a la extensión de todo el desarrollo, es imposible abarcar en detalle todas y cada una de las funcionalidades de la aplicación *Sales*. Por este motivo, pueden consultarse en el **Apéndice A. Manuales de la aplicación** las guías de usuario para interactuar con *Sales*, así como las guías de administrador para interactuar con *Sales* con dicho rol. Por otra parte, en el mismo apéndice podrá consultarse una guía orientada al montaje de la aplicación en su conjunto.

### 4.2.1 Estructura y funcionalidad

A diferencia del apartado Estructura del microservicio anterior, en este punto se abordará la aplicación *sales-client* desde el punto de vista de la funcionalidad del proyecto con una breve descripción y capturas de pantalla.

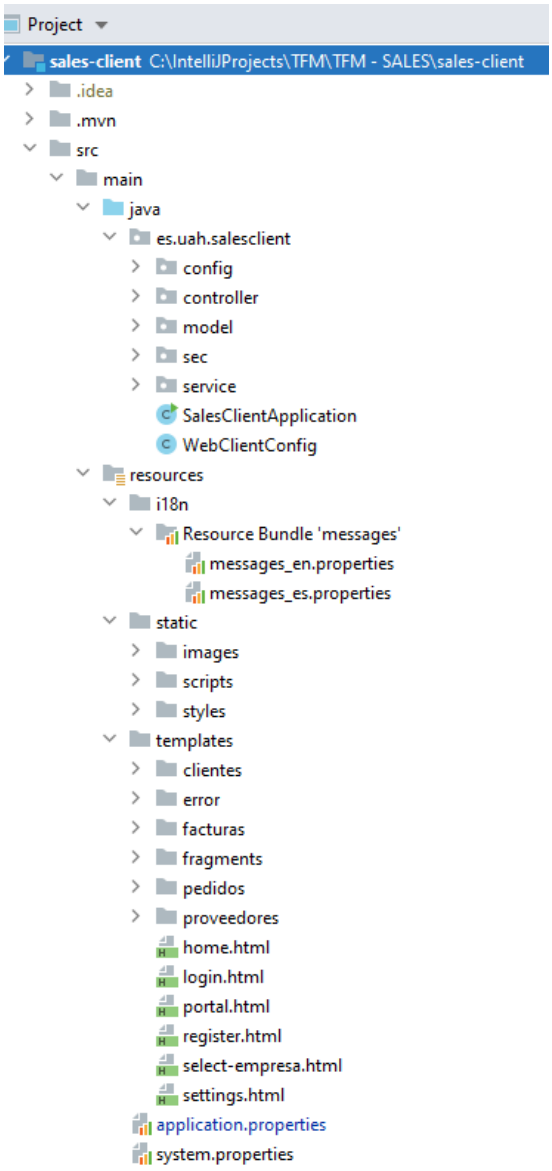



**Figura 45. Estructura de *sales-client***

Como se puede observar, este microservicio, al ser todo el frontal y contener la lógica de *Sales* es bastante más extenso que *sales-api*. El paquete Java principal es **es.uah.salesclient**, donde encontramos los paquetes que componen toda la funcionalidad *backend* del microservicio. Primero se hará una breve descripción de los estilos y las librerías que intervienen en la interfaz de *sales-client*.

#### 4.2.1.1 Estilos y librerías

*Sales* utiliza una serie de librerías y frameworks que ayudan entre otras cosas a que la interfaz de *Sales* sea más intuitiva, minimalista, sencilla de entender y fácil de implementar ciertos aspectos. Estas librerías se almacenan en el directorio **resources/static/scripts**, y son **BootStrap5 [52]**, un framework frontal que se utiliza para dar forma visual al sitio web y además incorpora soporte responsive, dando la posibilidad de visualizar el sitio web de forma correcta desde cualquier dispositivo. Por otra parte, **Font Awesome [53]**, una librería de iconos muy utilizada para el diseño web. Cuenta con una gran cantidad de iconos vectoriales y estilos css que ayudan a que la aplicación sea más intuitiva en muchos aspectos, entre otras

ventajas. Por último, se utiliza también **jQuery [54]**, librería de JavaScript que ayuda bastante a la hora de programar en JS ciertas funcionalidades.

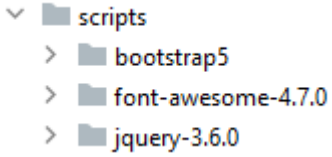



**Figura 46. Directorio resources/static/scripts**

Aunque aquí no aparezcan, en templates/fragments/general.html, un **fragment** que se utiliza en **todas las vistas de la aplicación**, se utilizan otra librería llamada **MapBox [55]**, utilizada para la visualización de mapas. Esto se usa en *sales-client* cuando un usuario visualiza la ubicación de un cliente, como se observa en la siguiente figura:

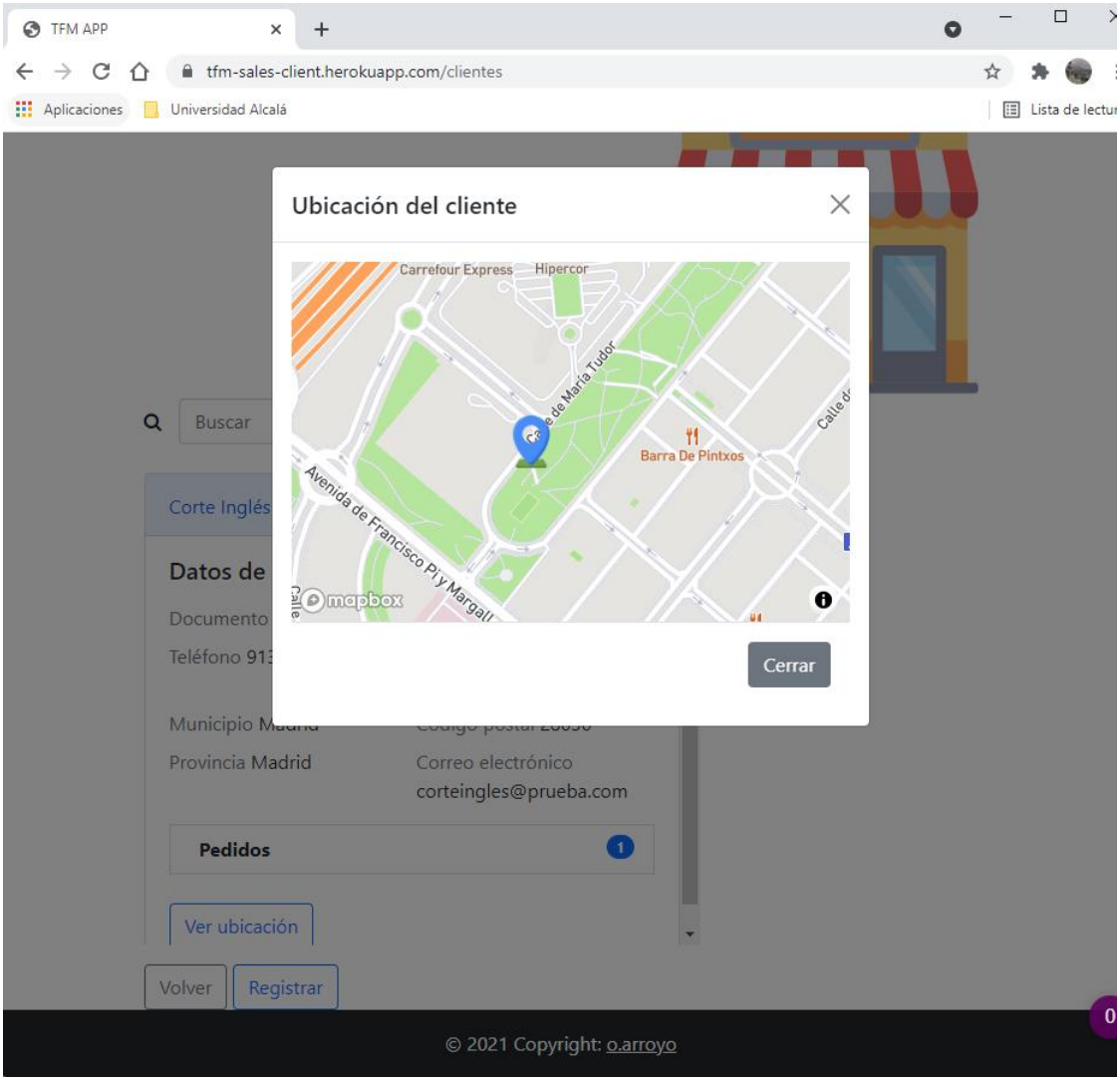



**Figura 47. Uso de MapBox en *sales-client***

Esta librería se ha implementado mediante JavaScript con el uso del API KEY que se obtiene al registrarse en la página web oficial, que permite de forma gratuita utilizar la librería hasta un límite de

consultas diarias. En la siguiente Figura se aprecia un fragmento en JavaScript de la implementación en clientes.html:

LoginController.java | login.html | general.html | clientes.html | ClientesController.java | WebClientConfig.java

```
<script th:inline="javascript">

  window.onload = function () {

   var clientes = /*[[${clientes}]]*/ 'default';
   var apiKey = /*[[${apikey}]]*/ 'default';

    mapboxgl.accessToken = apiKey;
    const map = new mapboxgl.Map({
        container: 'map',
        style: 'mapbox://styles/mapbox/streets-v11',
        center: [-3.3508448904281956, 40.51414509707626],
        zoom: 15.0
    });

  $(".flyer").on('click', function() {

    var term = $(this).attr('value');

    var cliente = clientes.find(c => c.nombre === term);

    map.flyTo({
        center: [
            cliente.ubicacion.longitud, cliente.ubicacion.latitud
        ],
        essential: true
    });

     var el = document.createElement('div');
        el.className = 'marker';

  new mapboxgl.Marker(el)
            .setLngLat({lng: cliente.ubicacion.longitud, lat: cliente.ubicacion.latitud})
          .setPopup(
                new mapboxgl.Popup()
                    .setHTML(
                        '<h4 class=\'mt-3\'>' +
                        cliente.documento.numero +
                        '</h4><p>' +
                        cliente.nombre +
                        '</p>'
                    )
```

**Figura 48. Fragmento de implementación MapBox (clientes.html)**

Por cuestiones de privacidad se oculta parte del API KEY utilizado durante el desarrollo mediante el uso del properties.

En relación con los estilos, además de **Bootstrap**, también se hace uso del directorio **resources/static/styles** y **resources/static/images:**

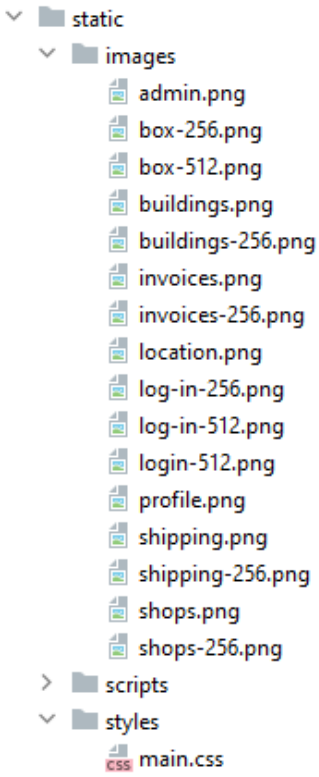



**Figura 49. Directorios resources/static/styles e images**

Las imágenes son de libre uso y los créditos a los autores están recogidos en el fichero **README.md** del proyecto. Estas imágenes se obtuvieron de flaticon [56]:

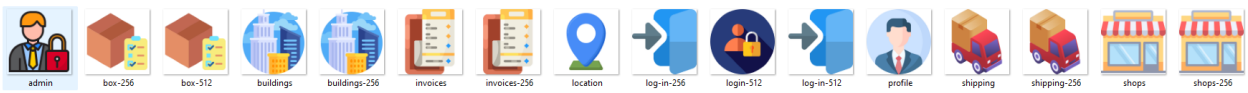



**Figura 50. Imágenes de Flaticon**

En cuanto al directorio **styles,** únicamente se incluye el fichero main.css, donde hay algunos estilos para las vistas:

main.css

*.css files are supported by IntelliJ IDEA Ultimate

```
.map {
    position: absolute;
    top: 0;
    bottom: 0;
    width: 100%;
}
.marker {
    background-image: url(/images/location.png);
    background-size: cover;
    width: 50px;
    height: 50px;
    border-radius: 50%;
    cursor: pointer;
}
.mapboxgl-popup {
    max-width: 200px;
}
.mapboxgl-popup-content {
    text-align: center;
}
```

**Figura 51. Main.css**

### 4.2.1.2 Internacionalización

El paquete **config** se encarga principalmente de la funcionalidad relacionada con la internacionalización mediante la clase **MessageConfig.java**.


**Figura 52. Paquete config**

La internacionalización permite que *sales*-client muestre la interfaz en los idiomas que se configure. Como se recogió en el **Análisis de Requisitos no funcionales**, el mínimo de idiomas que se escogió para soportar son el **español y el inglés**. En pocas líneas de código es posible implementar esta funcionalidad gracias a la interfaz **WebFluxConfigurer [51]**. Este interfaz permite personalizar la configuración de la aplicación en el marco reactivo de Spring WebFlux. En este Trabajo Fin de Máster no ha sido necesario utilizar la anotación **@EnableWebFlux** debido a que los microservicios inicialmente ya están con dicho marco de trabajo. Una vez establecida la implementación de la interfaz, podemos crear **Beans** que arrancarán una vez se inicialice el microservicio gracias al etiquetado **@Configuration**.

MessageConfig.java

```java
package es.uah.salesclient.config;

import org.springframework.context.MessageSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.reactive.config.WebFluxConfigurer;

@Configuration
/**
 * MessageConfig Necesaria para la internacionalización. Si se cambia el navegador de idioma,
 * cambiará la aplicación
 * @author o.arroyo
 */
public class MessageConfig implements WebFluxConfigurer {

  @Bean
  public MessageSource messageSource() {
    ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();
    messageSource.setBasenames("i18n/messages");
    messageSource.setDefaultEncoding("UTF-8");
    return messageSource;
  }
}
```

**Figura 53. MessageConfig.java**

Mediante el método **messageSource** establecemos una ruta donde estarán los ficheros de traducciones, concretamente en el directorio "i18n/messages". Posteriormente, para que no haya problemas de codificación de caracteres, se escoge por defecto UTF-8. El directorio messages no es más que un "Resource Bundle" o paquete de recursos, con ficheros .properties que siguen un patrón concreto:

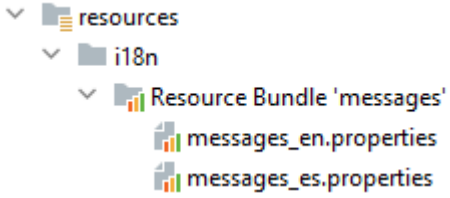



**Figura 54. Directorio i18n/messages**

Como se puede observar, los ficheros de idioma siguen el patrón messages_**XX**.properties, donde **XX** corresponde al término abreviado de cada idioma. Estos ficheros siguen el mismo formato que cualquier .properties del marco Spring. Si consultamos el fichero en inglés:

messages_en.properties

```
portal.title = SALES CLIENT
portal.eslogan=The application for commercial freelancers and sales, at your fingertips
portal.empezar=Get Started

administrador=Admin
usuario=User

home.breadcrumb=Home
home.acceso=Fast access to platform functionalities:
home.cambiarcliente=Choose/Change customer
home.cambiarproveedor=Choose/Change supplier

login.aviso=To access by username and password you must be registered in the platform.
login.signin=Sign in
login.signup=Sign up
login.username=Username
login.password=Password
login.error=There was an error trying to log in. Please try again.

register.confirmarpassword=Confirm password
register.completar=Complete registry

logout.ok=Session successfully completed.

registro.completar=Complete registration

select.breadcrumb=Select company
select.title=Companies represented
select.subtitle=Select the company
select.sinproveedores=No suppliers can be found where you are listed as a representative. Contact your administrator.
```

**Figura 55. Fragmento de messages_en.properties**

¿Cómo puede comprobarse que esta característica funciona correctamente? Si el usuario entra en el entorno actual de producción (https://tfm-sales-client.herokuapp.com/), observaría lo siguiente:

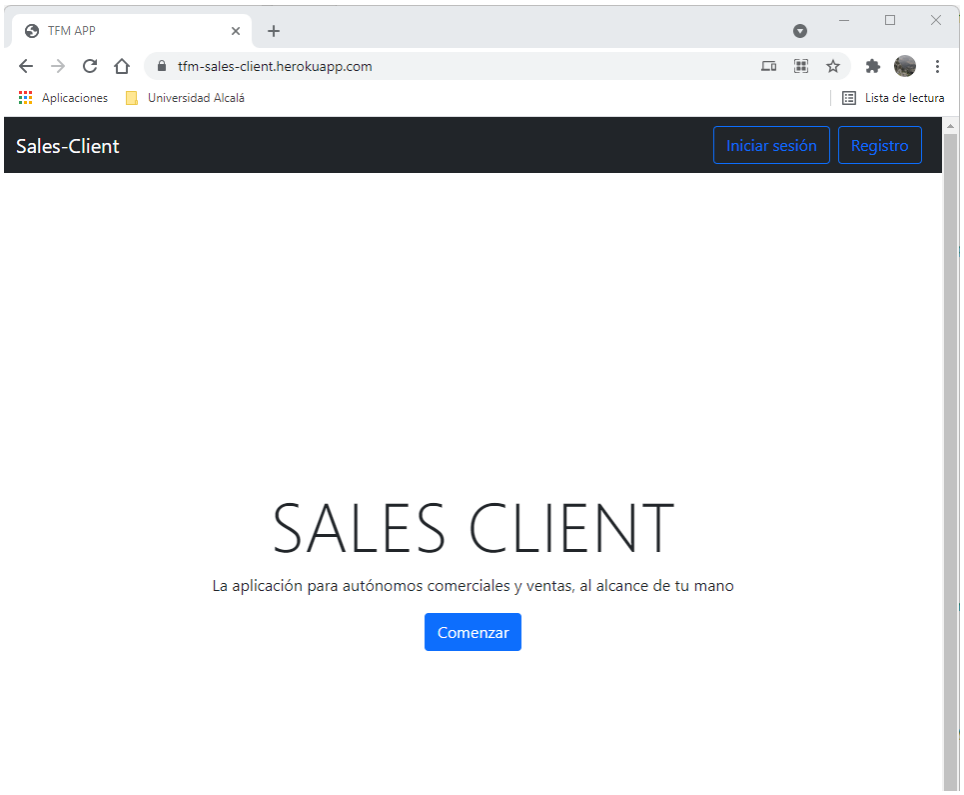


**Figura 56. Pantalla inicial de *sales-client***

Sin embargo, esto es porque en la configuración del dispositivo se tiene establecido el español como idioma predeterminado. En este caso, desde el navegador Chrome:

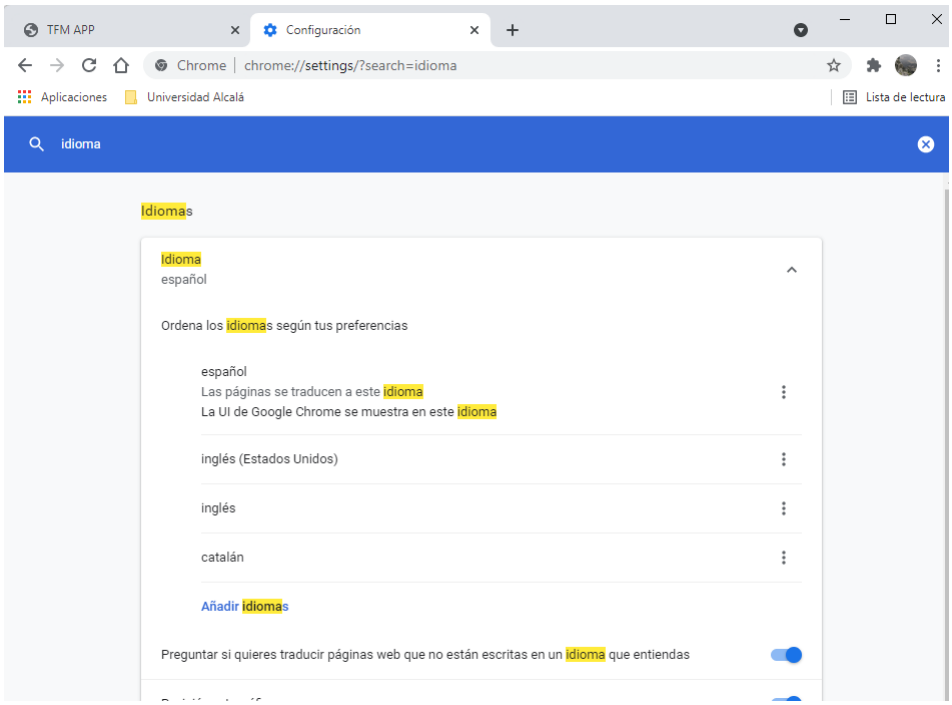



**Figura 57. Configuración del idioma del navegador**

Realizando el cambio de inglés (Estados Unidos) al primer lugar, el resultado del sitio web es el siguiente:

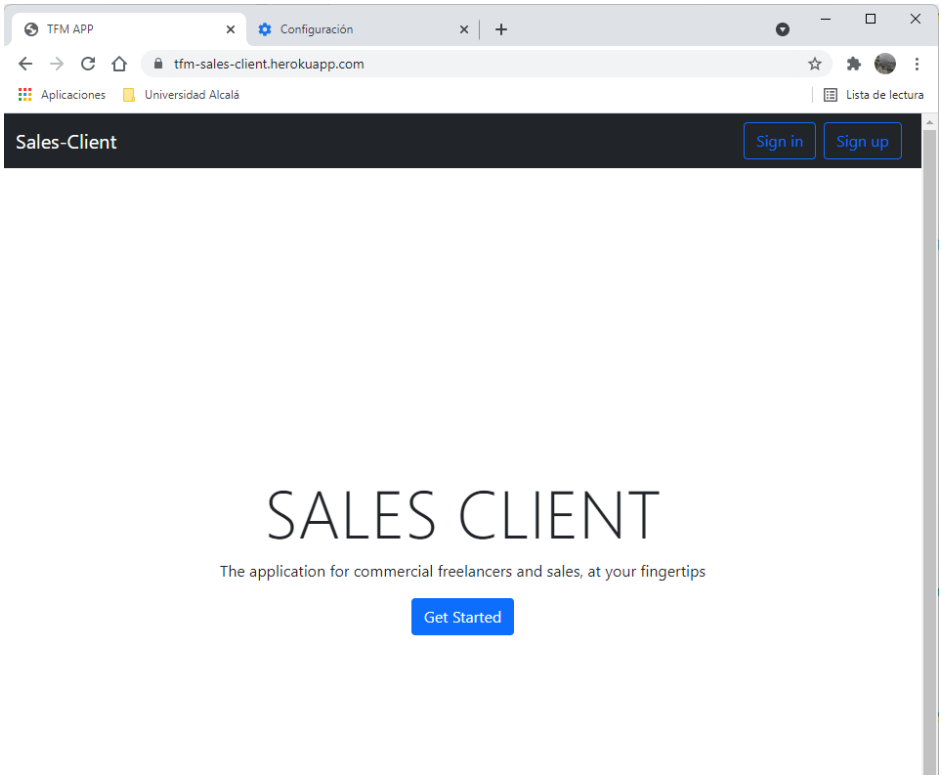



**Figura 58. Pantalla inicial de *sales-client (inglés)***

El resultado es que toda la interfaz de *Sales* a partir de ahora aparecerá traducida en su totalidad al idioma nativo del navegador, en este caso, inglés. Esto se mostrará con mayor detenimiento en uno de los vídeos del **Apéndice A.**

#### 4.2.1.3 Login y logout

El login es el sistema de autenticación que tiene este microservicio mediante credenciales que corresponden al documento de identidad del usuario y su contraseña. Al igual que vimos en el microservicio *sales-api,* en este caso también se utiliza el marco de trabajo Spring Security para Spring WebFlux, solo que con más configuraciones que el api debido a su complejidad. Toda la configuración se encuentra de igual forma en el paquete **sec**, clase **SecurityConfig:**

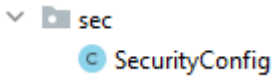



**Figura 59. Paquete sec**

```
SecurityConfig.java
32
33      @Autowired IUsuarioService usuarioService;
34
35      @Bean
36      public PasswordEncoder encoder() { return new BCryptPasswordEncoder(); }
39
40      @Bean
41  @   public SecurityWebFilterChain securityWebFilterChain(ServerHttpSecurity http) {
42        return http.csrf() ServerHttpSecurity.CsrfSpec
43            .disable() ServerHttpSecurity
44            .cors() ServerHttpSecurity.CorsSpec
45            .and() ServerHttpSecurity
46            .httpBasic() ServerHttpSecurity.HttpBasicSpec
47            .and() ServerHttpSecurity
48            .authorizeExchange() ServerHttpSecurity.AuthorizeExchangeSpec
49            .pathMatchers( ...antPatterns: "/proveedores/**") ServerHttpSecurity.AuthorizeExchangeSpec.Access
50            .hasRole("ADMIN") ServerHttpSecurity.AuthorizeExchangeSpec
51            .pathMatchers(
52                ...antPatterns: "/",
53               "/portal",
54               "/login",
55               "/register/**",
56               "/styles/**",
57               "/images/**",
58               "/scripts/**",
59               "/error") ServerHttpSecurity.AuthorizeExchangeSpec.Access
60            .permitAll() ServerHttpSecurity.AuthorizeExchangeSpec
61            .anyExchange() ServerHttpSecurity.AuthorizeExchangeSpec.Access
62            .authenticated() ServerHttpSecurity.AuthorizeExchangeSpec
63            .and() ServerHttpSecurity
64            .formLogin() ServerHttpSecurity.FormLoginSpec
65            .loginPage("/login")
66            .authenticationManager(manager())
67            .authenticationSuccessHandler(new WebFilterChainServerAuthenticationSuccessHandler())
68            .and() ServerHttpSecurity
69            .logout() ServerHttpSecurity.LogoutSpec
70            .logoutUrl("/logout")
71            .requiresLogout(ServerWebExchangeMatchers.pathMatchers(HttpMethod.GET,  ...patterns: "/logout"))
72            .and() ServerHttpSecurity
73            .build();
74  }
```

**Figura 60. securityWebFilterChain de SecurityConfig.java (*sales-client)***

La clase SecurityConfig no necesita implementar una interfaz en especial, sino que basta con que tenga el anotado **@EnableWebFluxSecurity**.

Al igual que en *sales-api*, el método **securityWebFilterChain** recoge las normas de conexión al microservicio. Como se puede apreciar en la figura anterior, tiene configuraciones como las siguientes:

- authorizeExchange()
- pathMatchers("/").permitAll()
- anyExchange().authenticated()

Estas dos configuraciones en particular permiten el acceso a cualquier dirección del microservicio **siempre que esté autenticado.**

- formLogin().loginPage("/login")

Esta configuración **habilita** el formulario de login en la ruta definida **/login**. Además, mediante la siguiente configuración:

- authenticationManager(manager())
- authenticationSuccessHandler(new WebFilterChainServerAuthenticationSuccessHandler())

Se establece un **login diferente al estándar** que implementa Spring Security, delegando el comportamiento de la autenticación al método **manager()**, visible en la **Figura 47**:

```
76  @   private ReactiveAuthenticationManager manager() {
77        return authentication -> {
78          String documento = authentication.getName();
79          String password = authentication.getCredentials().toString();
80
81          List<GrantedAuthority> authorities = new ArrayList<>();
82
83          return usuarioService
84              .findByDocumentoNumero(documento)
85              .doOnNext(
86                  u -> {
87                    if (encoder().matches(password, u.getDatosRegistro().getPassword())
88                        && u.getDocumento().getNumero().equals(documento)) {
89                      authorities.add(
90                          new SimpleGrantedAuthority(u.getDatosRegistro().getAuthority().name()));
91                    }
92                  })
93              .switchIfEmpty(Mono.just(new Usuario()))
94              .flatMap(
95                  usuario -> {
96                    if (usuario == null) return Mono.error(new UsernameNotFoundException(documento));
97                    else if (!authorities.isEmpty()) {
98                      Authentication auth =
99                          new UsernamePasswordAuthenticationToken(usuario, password, authorities);
100                     return Mono.just(auth);
101                   } else {
102                     return Mono.error(new BadCredentialsException("Error en las credenciales"));
103                   }
104                 });
105       };
106   }
107 }
```

**Figura 61. manager de SecurityConfig.java (*sales-client)***

Este método parece complejo pero se reduce en los siguientes pasos: primero consulta mediante el servicio **usuarioService** si existe un usuario con el número de documento que se pasó mediante el formulario login, recogido mediante **authentication.getName()**. Con el uso de **doOnNext** se establece la lógica a realizar una vez recibe un flujo de datos reactivo, en este caso, un objeto del tipo Mono<Usuario>. Mediante el método **encoder()**, utilizado también en *sales-api* y que utiliza **BCryptPasswordEncoder**, se comprueba que la contraseña, recogida mediante **authentication.getCredentials().toString()**, una vez cifrada, sea idéntica a la de la base de datos. A su vez, se verifica que el documento del usuario encontrado por el servicio coincida con el introducido en el formulario, y en caso de que ambas casuísticas sean correctas, se añade la autoridad del usuario encontrado, que correspondería al ROL que se almacena en la base de datos: o ROL_USER, o ROL_ADMINISTRADOR.

Gracias a Spring WebFlux y, como hemos visto en otros casos, es posible establecer casuísticas de lógica como alternativas inesperadas. Mediante **switchIfEmpty(Mono.just(new Usuario()))** establecemos el comportamiento que se realizará cuando la respuesta del servicio **usuarioService** sea vacía, o en otras palabras, no encuentre ningún usuario con dicho documento. En este caso, se utilizaría un objeto del tipo Mono<Usuario> vacío, para verificar que si la lista de roles estuviese vacía, se devolviese **Mono.error,** indicando un error de credenciales. Esto a nivel de seguridad es importante: no es recomendable dar al usuario pistas del error. Una mala práctica sería indicar al usuario que la contraseña del documento introducido es incorrecta, pues de esta manera, el usuario ya sabría que existe un documento con ese número en concreto. En caso de que la autenticación sea correcta, se devuelve un objeto estándar de Spring Security: **UsernamePasswordAuthenticationToken**, donde se le pasa el objeto usuario, el password y los roles. Gracias a esto, en cada método de los controladores donde utilicemos:

```
@AuthenticationPrincipal Mono<Usuario> u
```

**Figura 62. @AuthenticationPrincipal Mono<Usuario> u**

Podremos utilizar el usuario con el que se inició sesión con Spring Security. Para terminar con la funcionalidad login, también se ha implementado un **logout** mediante las siguientes configuraciones en securityWebFilterChain:

- logout()
- logoutUrl("/logout")
- requiresLogout(ServerWebExchangeMatchers.pathMatchers(HttpMethod.GET, "/logout"))

Las dos primeras configuraciones son muy claras: se habilita el logout en la url "/logout". La tercera configuración es la más compleja pero a su vez igual de sencilla: habilita el requisito de logout cuando se realicen llamadas GET a "/logout". De esta manera, si se llama a "/logout", el microservicio comprobará que es la URL que se usa para el logout, redirigirá a "/logout" y ahí se terminará con la funcionalidad.

Nada de esto tiene sentido si no se crea el .html de login y los controladores que valga la redundancia controlan las rutas a "/login" y "/logout". La primera de ellas está en el controlador **LoginController**, que puede observarse en la siguiente figura:

LoginController.java

```java
import es.uah.salesclient.service.usuario.IUsuarioService;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.annotation.AuthenticationPrincipal;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;

import reactor.core.publisher.Mono;

@Controller
public class LoginController {

  private final Logger logger = LoggerFactory.getLogger(LoginController.class);

  @Autowired IUsuarioService usuarioService;

  @GetMapping(value = {"/login"})
  public Mono<String> login(@AuthenticationPrincipal Usuario u) throws Exception {
    logger.info("/login");
    if (u != null) {
      return Mono.just("redirect:/home");
    }

    return Mono.just("login");
  }

  @PostMapping("/login")
  public Mono<String> login(@AuthenticationPrincipal Mono<Usuario> u, Model model) {

    logger.info("/login [POST]");
    model.addAttribute( s: "usuario", u);
    return Mono.just("redirect:/select-empresa");
  }
}
```

**Figura 63. LoginController.java**

Y la vista login.html:

LoginController.java | login.html

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org" lang="es" class="h-100">
<head th:insert="fragments/general::head"></head>

<body class="d-flex flex-column h-100">
<nav th:replace="fragments/general::nav"></nav>
<div class="container mt-3">
    <nav aria-label="breadcrumb">
        <ol class="breadcrumb">
            <li class="breadcrumb-item"><a href="#"></a></li>
            <li class="breadcrumb-item active" aria-current="page" th:text="#{login.signin}"></li>
        </ol>
    </nav>
    <br>
    <h4 th:text="#{login.signin}"></h4>
    <br>
    <div class="col sm-6">
        <div class="col-sm-10">
            <img th:src="@{/images/log-in-256.png}" class="img-fluid rounded float-end">
        </div>
        <p class="text-primary"><i class="fa fa-info"></i>
            <span th:text="#{login.aviso}"></span>
        </p>
        <form action="login" method='POST'>
            <div class="form-floating mb-3 col-md-4">
                <input type="text" class="form-control" name="username" id="documento" th:placeholder="#{empresa.documento}" required>
                <label for="documento" class="col-sm-2 col-form-label" th:text="#{empresa.documento}"></label>
            </div>
            <div class="form-floating mb-3 col-md-4">
                <input type="password" class="form-control" name="password" id="password" th:placeholder="#{login.password}" required>
                <label for="password" class="col-sm-2 col-form-label" th:text="#{login.password}"></label>
            </div>
            <div th:if="${param.error}" class="alert alert-danger col-md-4" role="alert" th:text="#{login.error}"></div>
            <div th:if="${param.logout}" class="alert alert-success col-md-4" role="alert" th:text="#{logout.ok}"></div>
            <button type="submit" class="btn btn-primary" th:text="#{login.signin}"></button>
        </form>
    </div>
</div>

<div class="position-fixed bottom-0 end-0 p-3" style="...">
    <div id="liveToastOk" class="toast hide" role="alert" aria-live="assertive" aria-atomic="true">
        <div class="toast-header">
            <i class="fa fa-pencil rounded me-2 text-success"></i>
            <strong class="me-auto text-success">Aviso</strong>
```

**Figura 64. Fragmento de login.html**

El fichero es más extenso pero se puede apreciar principalmente el uso de los **fragments** de Thymeleaf [61] que se mencionará ellos en puntos posteriores, y el formulario "login" que redirige una vez se envía al controlador explicado anteriormente que, gracias a Spring Security, se detectará y pasará por el **manager()** que se encargará de realizar la funcionalidad pertinente.

Por último se adjuntan capturas de pantalla cuando el usuario introduce unos datos correctos, cuando falla al escribir las credenciales y finalmente cuando cierra la sesión:

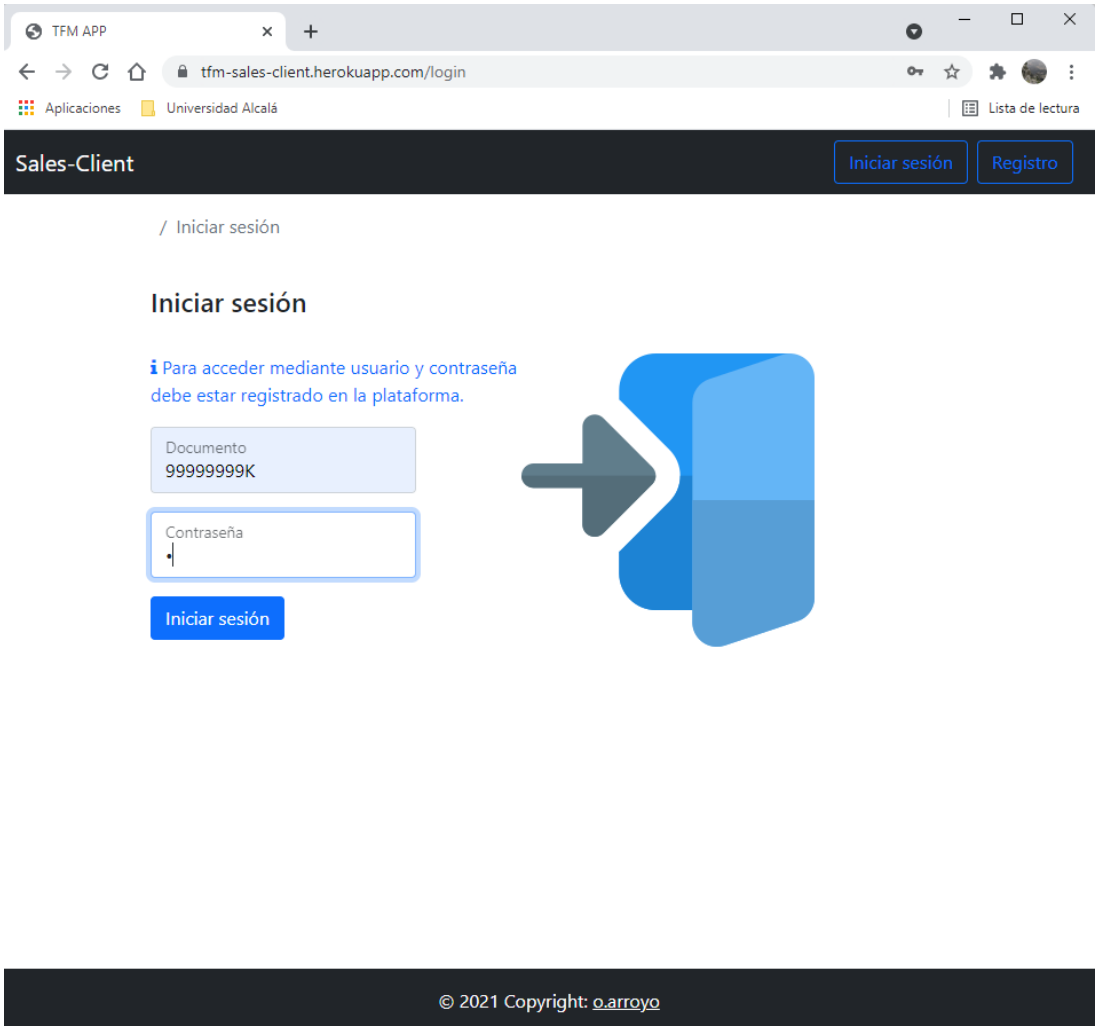



**Figura 65. Formulario login.html**

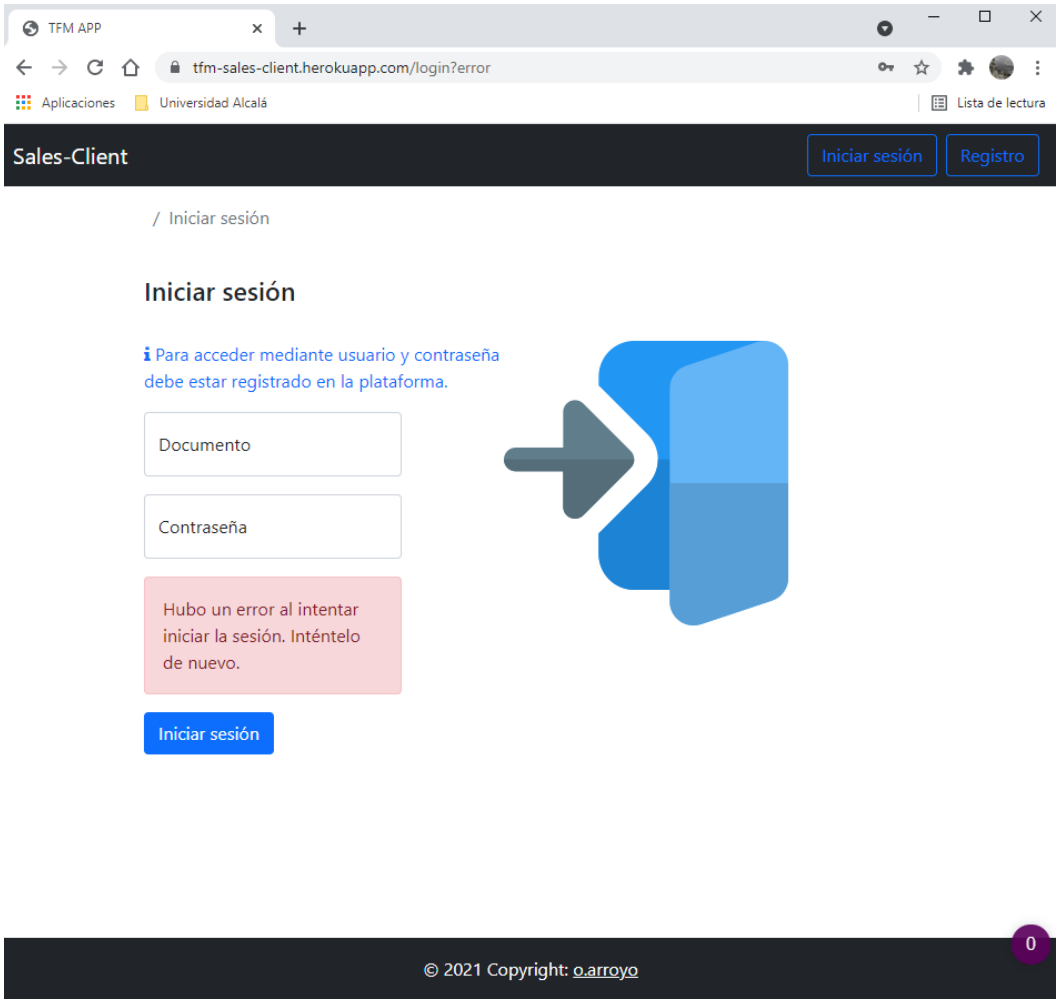



**Figura 66. Formulario login.html (error)**

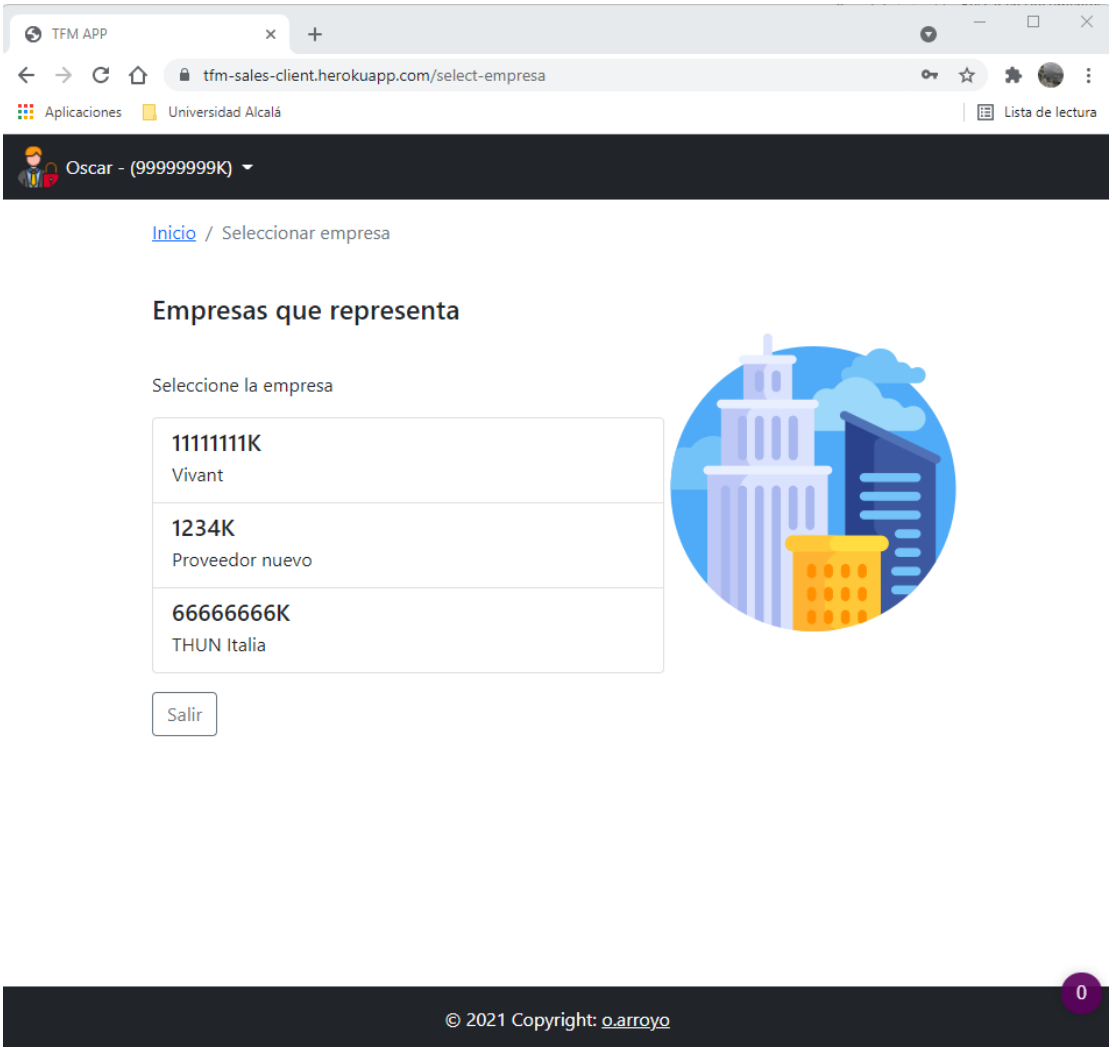



**Figura 67. Formulario login.html (éxito)**

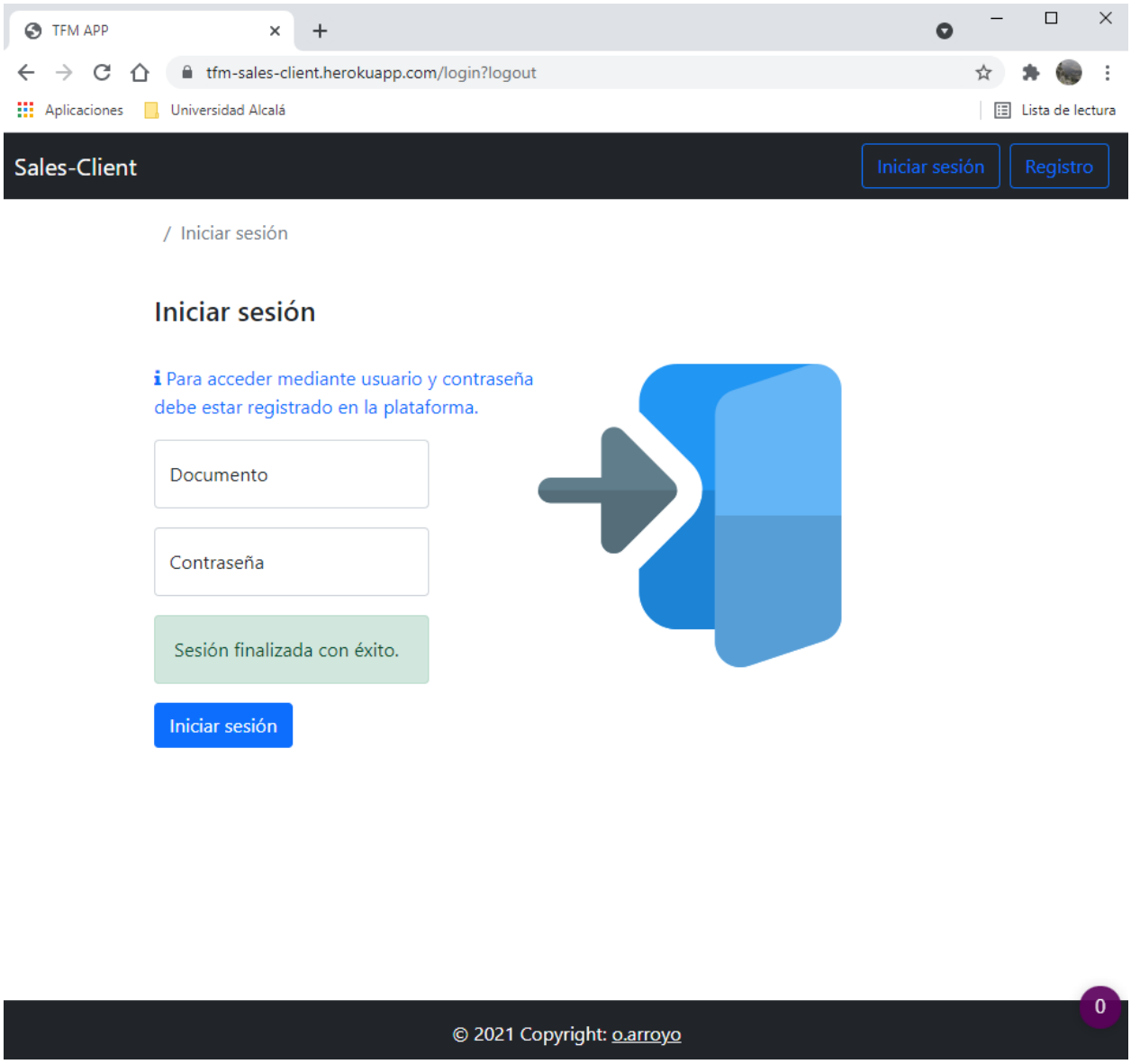



**Figura 68. Logout (éxito)**

#### 4.2.1.4 Registro

Una de las características más importantes que ofrece la plataforma es que cualquier usuario puede registrarse en la aplicación y utilizarla como desee. Los administradores desde dentro de la plataforma pueden realizar manualmente registros de usuarios con el Rol que deseen, mientras que un visitante de la web sólo podrá registrarse con el Rol de usuario (USER).

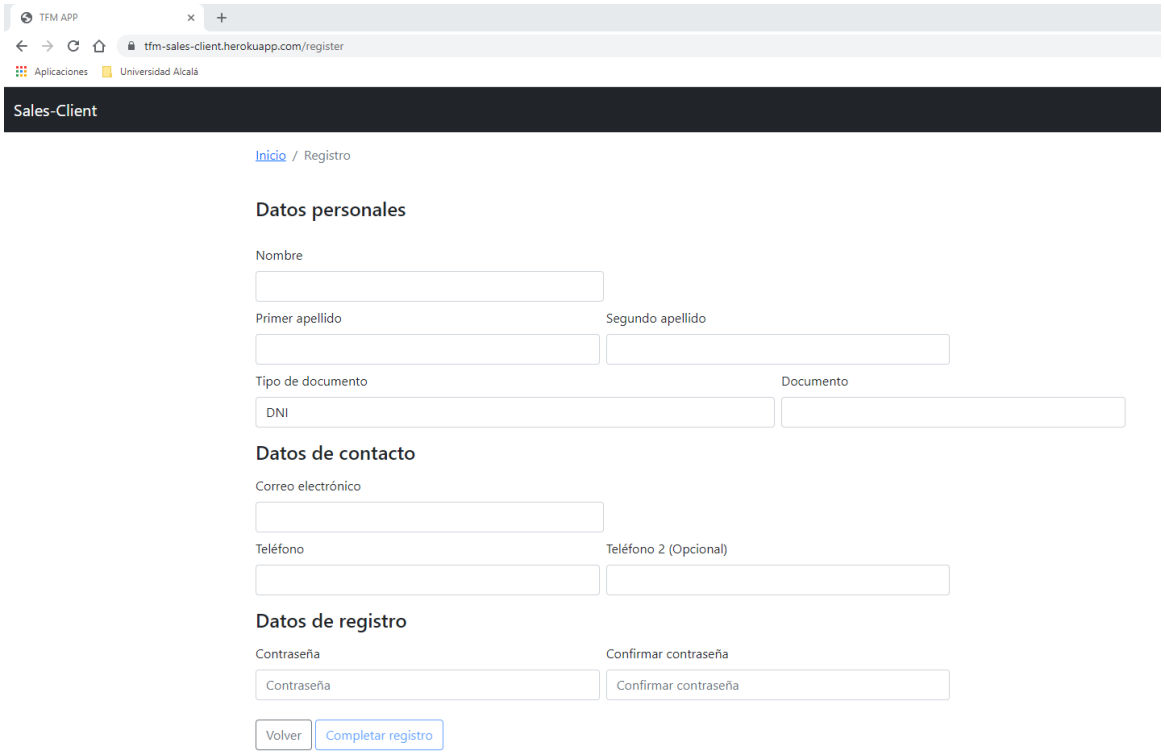



**Figura 69. Formulario de Registro (visitante)**

Una de las peculiaridades de este formulario es que utiliza jQuery para evitar que el usuario complete el registro si las contraseñas no coinciden:

```
<script th:inline="javascript">
$(document).ready(function() {
   $("input[type='password']").on('keyup', function() {
        if($("#password_confirm").val() === $("#registro_password").val()){
             $("#completarRegistro").prop('disabled', false);
        }else{
            $("#completarRegistro").prop('disabled', true);
        }
   });
});
```

**Figura 70. Comprobación de coincidencia entre contraseñas**

Además, se controla que un usuario **no se registre con un documento que ya tenga un usuario existente en la base de datos.** Para mostrar el aviso, se hace uso de los toast de Bootstrap 5:

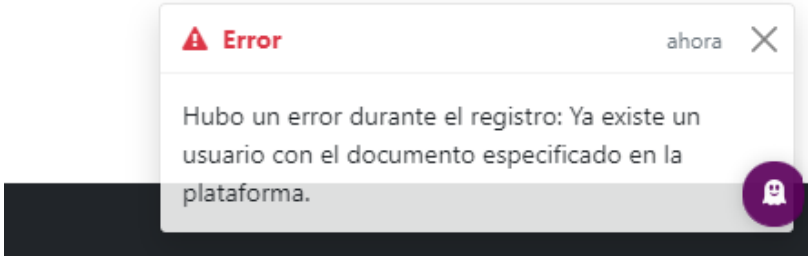



**Figura 71. Toast de error por usuario con documento existente**

Además de este toast, existen otras circunstancias en la plataforma donde se utiliza este tipo de elementos, como por ejemplo, durante la configuración del usuario administrador, si este cambia el Rol en el formulario:

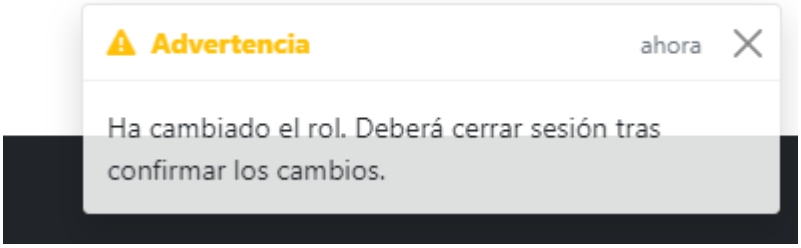



**Figura 72. Toast de advertencia por cambiar de Rol (ADMIN)**

Una vez el usuario completa el registro correctamente, puede entrar a la aplicación, pero no tendrá ningún proveedor asignado. Debe ser el administrador quien asigne uno o varios proveedores a este.

Por último, quien se ocupa de esta funcionalidad es un controlador llamado **RegisterController:**

RegisterController.java

```
33      model.addAttribute( s: "usuario", u == null ? new Usuario() : u);
34      model.addAttribute( s: "usuarioRegistro", new Usuario());
35      return Mono.just("register");
36    }
37
38    @PostMapping(value = "/save")
39 @  public Mono<String> save(
40        Model model, @ModelAttribute(value = "usuarioRegistro") Usuario usuario) {
41
42      logger.info("/save [POST]");
43
44      String passwordEncoded = encoder.encode(usuario.getDatosRegistro().getPassword());
45      usuario.getDatosRegistro().setPassword(passwordEncoded);
46
47      return usuarioService
48          .findByDocumentoNumero(usuario.getDocumento().getNumero())
49          .doOnNext(
50              u -> {
51                if (u != null && u.getId() != null) {
52                  logger.error(
53                      "Ya existe en la plataforma un usuario con el documento "
54                          + u.getDocumento().getNumero());
55                  model.addAttribute( s: "errorDocumento",  o: true);
56                }
57              })
58          .switchIfEmpty(Mono.just(new Usuario()))
59          .flatMap(
60              u -> {
61                if (u.getId() == null) {
62                  return usuarioService
63                      .save(usuario)
64                      .doOnNext(
65                          usuarioRegistrado -> {
66                            logger.info(
67                                "Usuario "
68                                    + usuarioRegistrado.getNombre()
69                                    + " registrado con éxito --> "
70                                    + usuarioRegistrado.getId());
71                            model.addAttribute( s: "exito",  o: true);
72                          })
73                      .onErrorResume(throwable -> Mono.error(throwable));
74                }
75                return Mono.empty();
76              })
77          .then(Mono.just("register"));
```

**Figura 73. Fragmento de RegisterController**

Este fragmento muestra cómo se completa el registro en el backend con programación reactiva: desde el encriptado de la contraseña, hasta la comprobación de que ese usuario no puede registrarse en la plataforma porque el documento ya existe. Si todo es correcto, se utiliza Thymeleaf para mostrar un Toast como mensaje de éxito que ayuda a redirigir al nuevo usuario al login:

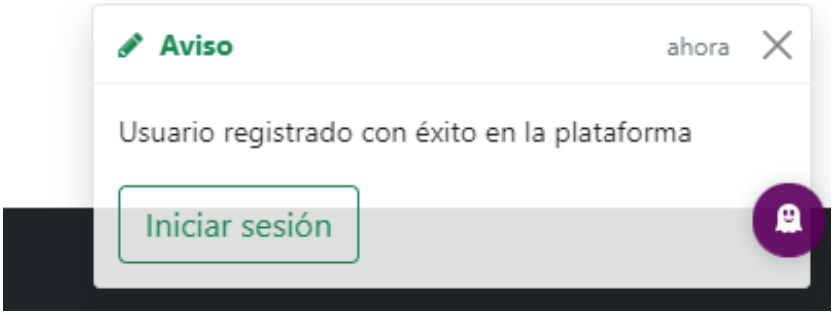



**Figura 74. Toast de aviso sobre el registro completado**

#### *4.2.1.5* Conexión con *sales-api*

Teniendo en cuenta la conexión por *BA* necesaria para las llamadas a *sales-api*, resulta evidente imaginar que *sales-client* necesitará establecer las conexiones con dicho microservicio mediante la misma estrategia.

En la aplicación conjunta *Sales*, ambos microservicios deben conocer las credenciales (usuario y contraseña) para compartir información.

Al igual que en *sales-api*, *sales-client* cuenta con un paquete llamado **service**, que a su vez se divide en paquetes que recogen la funcionalidad servicio para cada una de las colecciones de la base de datos. Esta es la estructura del paquete:

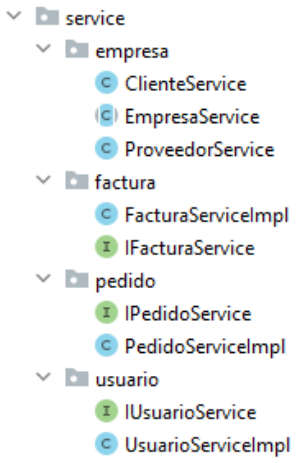



**Figura 75. Paquete service**

La principal diferencia entre estos servicios y los servicios del api son principalmente que en este caso, se utilizan **WebClients [57]**, interfaces de **org.springframework.web.reactive**, que implementan un cliente reactivo sin bloqueo entre solicitudes. De esta manera, estamos creando una conexión reactiva hacia un api reactivo.

Para inicializar el WebClient que establece la configuración de conexión con el microservicio *sales-api* se hace uso de la clase **WebClientConfig.** Es una clase que se inicia junto con la aplicación debido a su anotado **@Configuration**. Como veremos a continuación, consiste tan solo en un Bean y unos valores del fichero .properties, y con ello se consigue establecer una conexión mediante acceso básico al servicio api:

WebClientConfig.java

```java
package es.uah.salesclient;

import io.netty.channel.ChannelOption;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.client.reactive.ReactorClientHttpConnector;
import org.springframework.web.reactive.function.client.WebClient;
import reactor.netty.http.client.HttpClient;

@Configuration
public class WebClientConfig {

  @Value("${api.endpoint}")
  public String endpointUser;

  @Value("${api.authentication.username}")
  public String username;

  @Value("${api.authentication.password}")
  public String password;

  @Bean(value = "WebClient")
  public WebClient.Builder registrarWebClient() {
    HttpClient client = HttpClient.create()
            .option(ChannelOption.SO_KEEPALIVE, true);

    return WebClient.builder()
            .clientConnector(new ReactorClientHttpConnector(client))
            .defaultHeaders(header -> header.setBasicAuth(username, password))
            .baseUrl(endpointUser);
  }
}
```

**Figura 76. WebClientConfig.java**

Como puede observarse, en primer lugar se recogen mediante la etiqueta **@Value** las propiedades del .properties que corresponden al **endpoint** o dirección del microservicio sales-api y a las credenciales **username** y **password** de la autenticación básica.

Con estos datos inyectados en la clase y mediante el método **registrarWebClient** y el Bean, se crea la funcionalidad necesaria para la conexión.

Primero debe llamarse al método **builder**, interfaz estática encargada de construir el WebClient. Posteriormente se establece el **ClientHttpConnector [58]** para el servicio. A partir de esta interfaz se escoge la implementación del conector, y en este caso se ha escogido por **ReactorClientHttpConnector [59]**, debido a que es la implementación de Reactor-Netty del ClientHttpConnector, es decir, la implementación reactiva que busca este trabajo, conectando el microservicio *sales-client* con *sales-api* de forma no bloqueante.

El punto de seguridad más importante viene a continuación mediante el método **defaultHeaders**, utilizado como se puede apreciar en la imagen para establecer la *BA* con la expresión lambda y el método **setBasicAuth(username, password)**. De esta manera, toda petición que pase por este WebClient llevará la cabecera de basic authentication, necesarias para conectar con *sales-api*. En el navegador puede

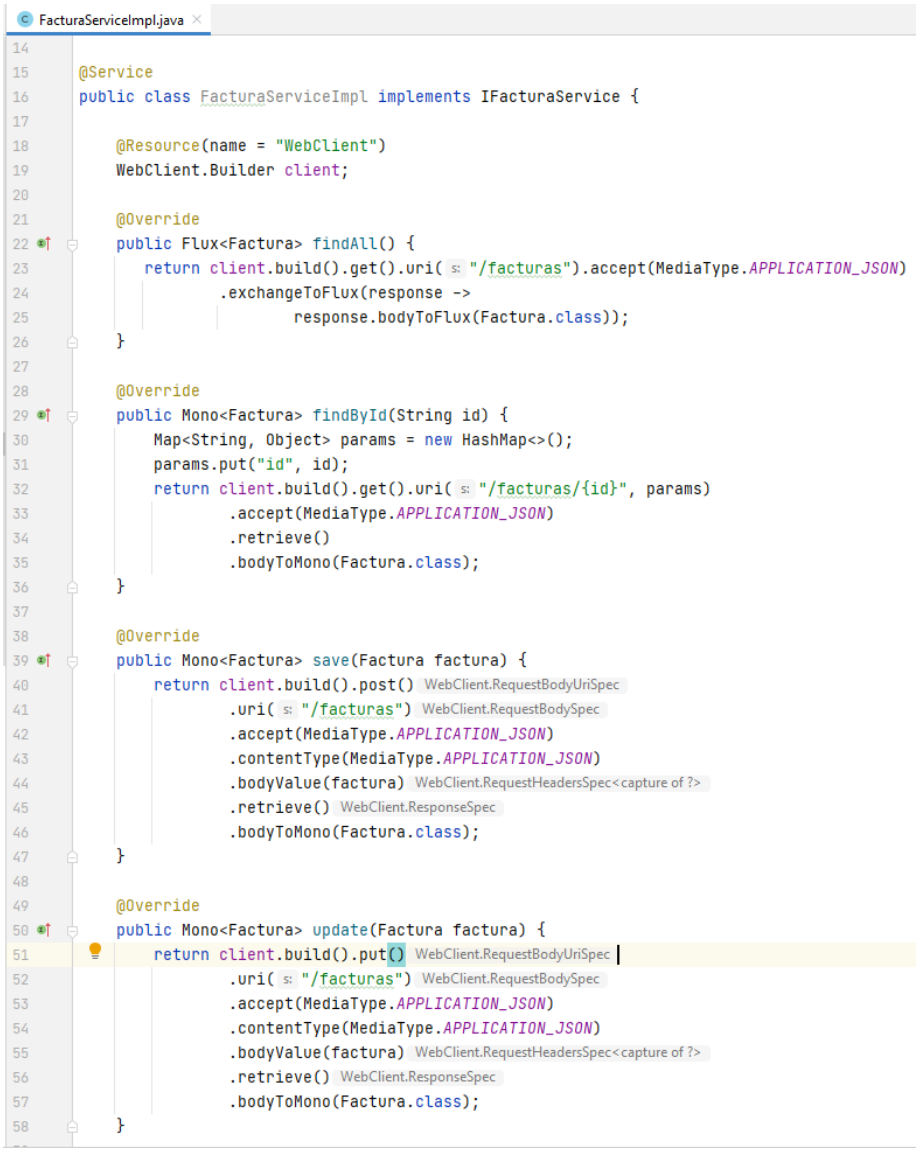
observarse esta cabecera si se realiza una petición al microservicio y se introducen correctamente las credenciales:


**Figura 77. Llamada a /facturas desde el navegador (éxito)**

Si por lo contrario, se prueba introduciendo las credenciales erróneas o no introduciéndolas, se puede apreciar cómo, a diferencia de responder el microservicio con un código 200 u OK, responde con un **401**, que corresponde a **Unauthorized (no autorizado)**:


**Figura 78. Llamada a /facturas desde el navegador (no autorizado)**

Por lo tanto, con el Bean del método registrarWebClient creado correctamente, bastaría con inyectarlo en las clases servicios y llamar a los métodos que necesite el microservicio. A continuación y a modo de ejemplo, se mostrará un fragmento de código del servicio de facturas, siendo la **clase FacturaServiceImpl.java e interfaz IFacturaService** las encargadas de la funcionalidad:



```
FacturaServiceImpl.java
14
15  @Service
16  public class FacturaServiceImpl implements IFacturaService {
17
18      @Resource(name = "WebClient")
19      WebClient.Builder client;
20
21      @Override
22      public Flux<Factura> findAll() {
23          return client.build().get().uri( s: "/facturas").accept(MediaType.APPLICATION_JSON)
24                  .exchangeToFlux(response ->
25                          response.bodyToFlux(Factura.class));
26      }
27
28      @Override
29      public Mono<Factura> findById(String id) {
30          Map<String, Object> params = new HashMap<>();
31          params.put("id", id);
32          return client.build().get().uri( s: "/facturas/{id}", params)
33                  .accept(MediaType.APPLICATION_JSON)
34                  .retrieve()
35                  .bodyToMono(Factura.class);
36      }
37
38      @Override
39      public Mono<Factura> save(Factura factura) {
40          return client.build().post() WebClient.RequestBodyUriSpec
41                  .uri( s: "/facturas") WebClient.RequestBodySpec
42                  .accept(MediaType.APPLICATION_JSON)
43                  .contentType(MediaType.APPLICATION_JSON)
44                  .bodyValue(factura) WebClient.RequestHeadersSpec<capture of ?>
45                  .retrieve() WebClient.ResponseSpec
46                  .bodyToMono(Factura.class);
47      }
48
49      @Override
50      public Mono<Factura> update(Factura factura) {
51          return client.build().put() WebClient.RequestBodyUriSpec
52                  .uri( s: "/facturas") WebClient.RequestBodySpec
53                  .accept(MediaType.APPLICATION_JSON)
54                  .contentType(MediaType.APPLICATION_JSON)
55                  .bodyValue(factura) WebClient.RequestHeadersSpec<capture of ?>
56                  .retrieve() WebClient.ResponseSpec
57                  .bodyToMono(Factura.class);
58      }
```

**Figura 79. Fragmento de FacturaServiceImpl.java**

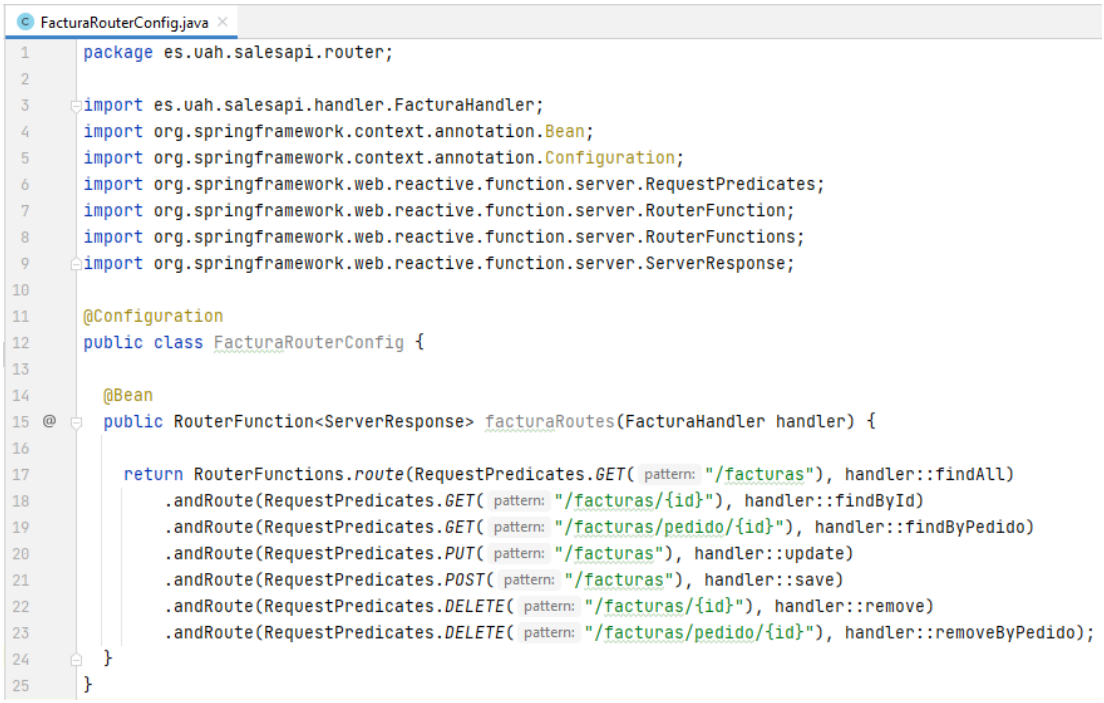
Observando la Figura, las líneas 18 y 19 corresponden a la inyección del WebClient instanciado anteriormente en la clase WebClientConfig.

Si por ejemplo el microservicio quiere solicitar a *sales-api* por todas las facturas, ejecutará el método **findAll**. Este método comienza con un **client.build()**, mediante el cual construye el WebClient y así puede utilizarse.

A continuación, se utiliza el método **get()** para crear una solicitud del tipo GET, y después llama a **uri()** para establecer la dirección de la petición, en este caso, "/facturas". De este modo, en este momento se estaría llamando al endpoint, que suponemos en este punto que fuese https://tfm-sales-api.herokuapp.com/ , y se le concatenaría la uri, resultando la siguiente petición:

- https://tfm-sales-api.herokuapp.com/facturas

Si se hace memoria, en el microservicio *sales-api*, las direcciones se manejaban con ayuda de las clases Router y se desarrollaban con las clases Handler. Consultando la siguiente Figura, correspondiente a la clase Router de las facturas:



```java
package es.uah.salesapi.router;

import es.uah.salesapi.handler.FacturaHandler;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RequestPredicates;
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.RouterFunctions;
import org.springframework.web.reactive.function.server.ServerResponse;

@Configuration
public class FacturaRouterConfig {

  @Bean
  public RouterFunction<ServerResponse> facturaRoutes(FacturaHandler handler) {

    return RouterFunctions.route(RequestPredicates.GET( pattern: "/facturas"), handler::findAll)
        .andRoute(RequestPredicates.GET( pattern: "/facturas/{id}"), handler::findById)
        .andRoute(RequestPredicates.GET( pattern: "/facturas/pedido/{id}"), handler::findByPedido)
        .andRoute(RequestPredicates.PUT( pattern: "/facturas"), handler::update)
        .andRoute(RequestPredicates.POST( pattern: "/facturas"), handler::save)
        .andRoute(RequestPredicates.DELETE( pattern: "/facturas/{id}"), handler::remove)
        .andRoute(RequestPredicates.DELETE( pattern: "/facturas/pedido/{id}"), handler::removeByPedido);
  }
}
```

**Figura 80. Clase FacturaRouterConfig**

Observamos que la petición o ruta GET que obedece a la dirección "/facturas" aparece en la línea 17, y mediante **RouterFunctions,** ésta llamará al método de la clase **FacturaHandler findAll**:

```java
public Mono<ServerResponse> findAll(ServerRequest req) {
  return ServerResponse.ok()
      .contentType(MediaType.APPLICATION_JSON)
      .body(facturaService.findAll(), Factura.class);
}
```

**Figura 81. Método findAll de FacturaHandler**

Este método devolverá a *sales-client* todas las facturas en formato JSON que encuentre mediante el servicio **facturaService** del api, como se aprecia en la Figura anterior. Volviendo al método **findAll** de *sales-client*, tras establecer la uri, se elige el tipo de formato que aceptará como respuesta de la petición, en este caso, **MediaType.APPLICATION_JSON**:

```java
@Override
public Flux<Factura> findAll() {
   return client.build().get().uri( s: "/facturas").accept(MediaType.APPLICATION_JSON)
          .exchangeToFlux(response ->
                  response.bodyToFlux(Factura.class));
}
```

**Figura 82. Método findAll de FacturaServiceImpl**

Una vez configurada la llamada, se debe recoger la información mediante el método **exhangeToFlux** y posteriormente esta información, recogida en el cuerpo de la respuesta, se transformará a un flujo de datos Flux<Factura> mediante el método **bodyToFlux**.

Con esta estrategia se obtienen todas las facturas de forma no bloqueante o reactiva. Cabe recordar que esto ha sido tan sólo un método ejemplo. Cada uno de los servicios del paquete **service** implementa

numerosos métodos para manejar funcionalidades diferentes según las necesidades de *sales-client*, pero abordar cada uno de ellos en esta memoria sería prácticamente imposible.

#### 4.2.1.6 Gestión de errores

Una parte importante de *sales-client* es que este microservicio es capaz de gestionar los errores de forma excelente. Para ello, hace uso de una serie de ficheros de errores personalizados .html:

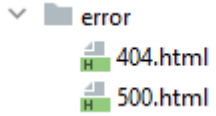



**Figura 83. Ficheros html personalizados para errores**

El marco de trabajo Spring implementa por defecto una página sin estilo que describe el error de la aplicación cuando ocurre durante su ejecución. Para evitar esta página estándar y adaptar los errores según nos convenga con la convención de la figura anterior, es necesario desactivar el **whitelabel** a partir del fichero .properties:

```
server.error.whitelabel.enabled=false
```

**Figura 84. Personalización de errores (.properties)**

Una vez establecida la propiedad a **false** , desde este momento los errores se redirigirán a las vistas ubicadas en **/templates/error** de forma estándar. Para personalizar cada vista dependiendo del error, basta con crear un .html con el código del error, como se observa en la **Figura 77**.

Ahora, para comprobar el funcionamiento correcto, se puede llamar a una ruta que no exista, por ejemplo: https://tfm-sales-client.herokuapp.com/prueba, resultando un **404** y mostrando lo siguiente:

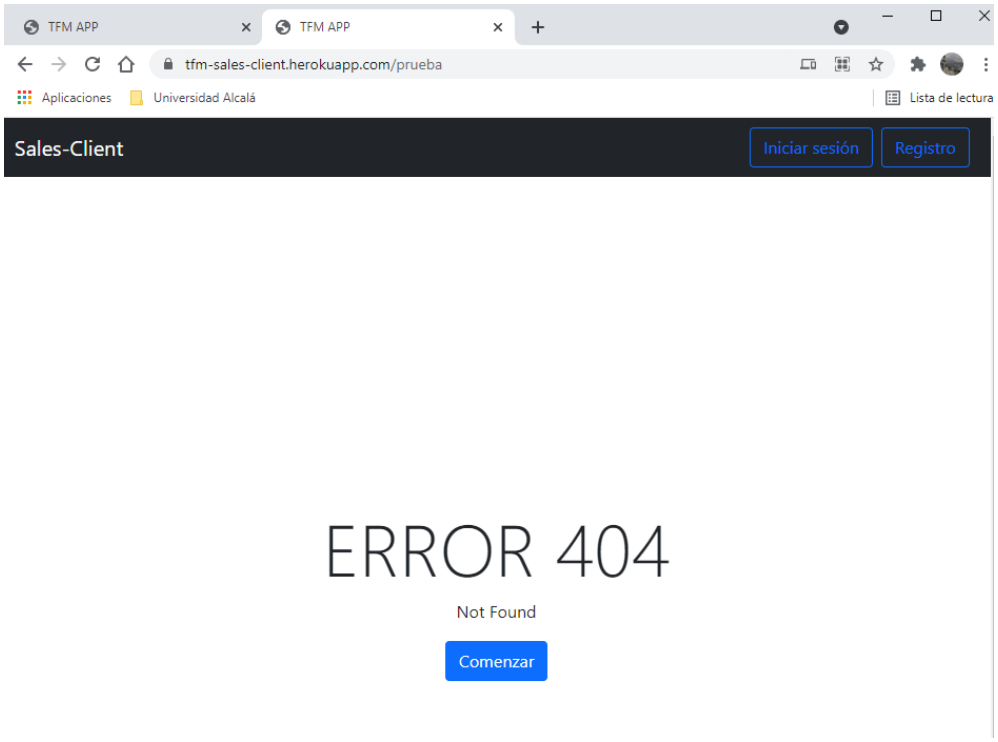



**Figura 85. Llamada a /prueba desde el navegador (no encontrado)**

Es una estrategia elegante para no mostrar al usuario ventanas de error no previstas por el desarrollador, y de esta manera, gestionar los errores inesperados de forma correcta.

#### 4.2.1.7 Interfaz de usuario/administrador

En este apartado se explicará parte de la aplicación que interviene en la interfaz y por tanto funcionalidad del usuario o el administrador, es decir, aquél comercial autónomo o usuario que inicia la sesión en la plataforma con el Rol **USER** o aquél administrador con el Rol **ADMIN**. De todas formas, es importante recalcar **que el cómo funciona la aplicación se explicará con detalle en el Apéndice A,** donde habrá manuales de usuario/administrador y montaje.

Antes de comenzar con cada funcionalidad, se adjunta una captura de los ficheros.html que intervienen como interfaz en todo el proyecto, ubicados en **/templates**:

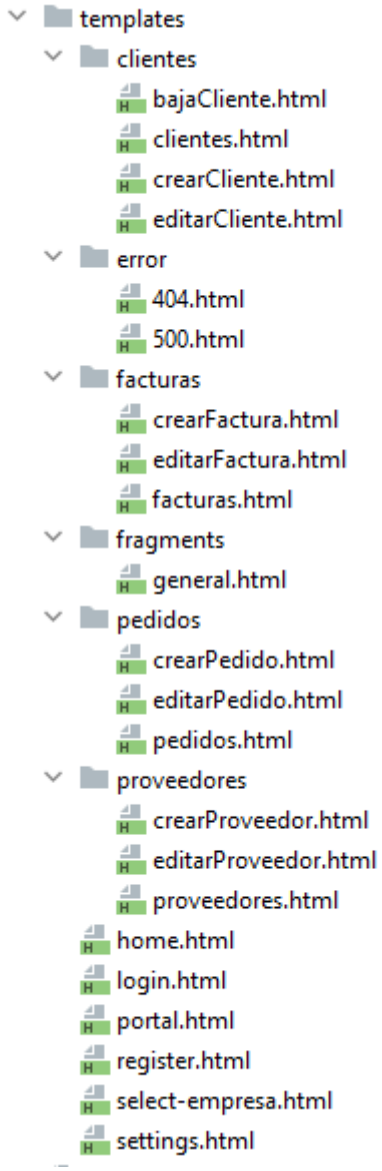



**Figura 86. Templates**

A continuación se explicará de forma breve la funcionalidad de la que se ocupan los ficheros, agrupando por carpetas:

- **Clientes**: Se ocupa de dar de baja, mostrar, modificar y crear un cliente. La eliminación del cliente no dispone de vista, sino que se realiza únicamente en el controlador y un botón de eliminación desde **clientes.html**.
- **Error**: Encargado de gestionar los errores que pueden suceder durante el uso de la aplicación, explicado en el punto **4.2.1.6**. Cada fichero representa la vista del error en cuestión: si necesitase manejarse el error 403 de forma personalizada, tendría que crearse **403.html**
- **Facturas:** Comprende la creación, edición, visualización y eliminación de las facturas, de la misma forma que clientes.

- **Fragments:** El fichero **general.html** es un fichero que utiliza lo que se conoce en Thymeleaf como fragments [61], muy utilizados en Spring para la reutilización de código en forma de componentes.
- **Pedidos:** Encargado de crear, modificar, visualizar o eliminar los pedidos.
- **Proveedores:** Se ocupa de la creación, modificación, visualización o eliminación de los proveedores.
- **Raíz:** En la raíz de templates se encuentran elementos comunes: **home** para el inicio, **portal** como página de acceso rápido, **login** para el inicio de sesión, **register** para el registro de usuarios, **select-empresa** para la selección de proveedores y **settings** para la modificación de los datos del usuario registrado.

#### 4.2.1.7.1 Selector de empresa/proveedor

Como la funcionalidad **Login** es compartida por todos los tipos de usuarios y se explicó en el apartado **4.2.1.3**, se comenzará por el **selector de proveedores**, que aparece una vez se inicia la sesión:

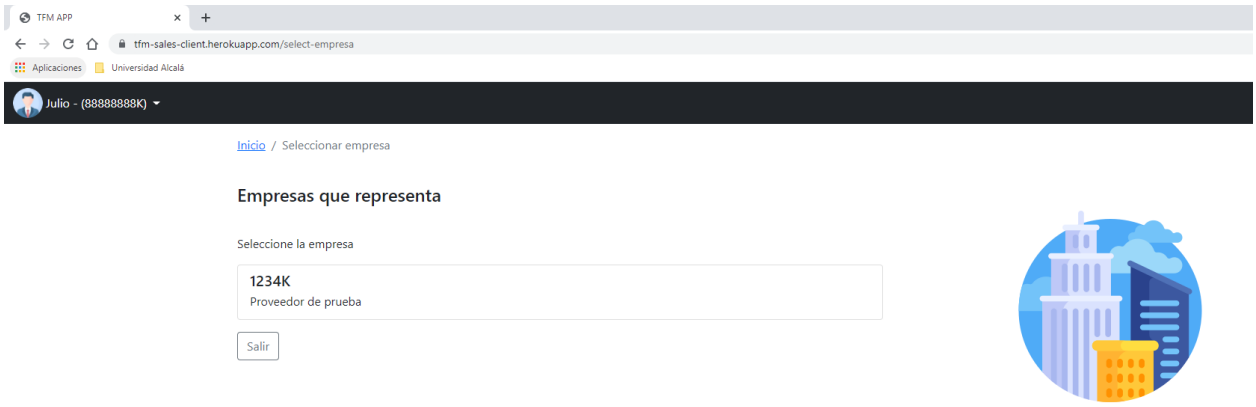



**Figura 87. Selector de proveedor (ROL_USER)**

Esta funcionalidad la describe el controlador **SelectorController**, que al igual que todos los controladores de los que se hablarán, forma parte del paquete **controller**. Los controladores son los que manejan el comportamiento de la aplicación: las vistas a las que dirigir al usuario, la información que mostrar, así como la lógica de negocio a seguir.

SelectorController.java

```java
package es.uah.salesclient.controller;

import ...

@Controller
public class SelectorController {

  private final Logger logger = LoggerFactory.getLogger(SelectorController.class);

  @Autowired EmpresaService<Proveedor> proveedorService;

  @GetMapping(value = {"/select-empresa"})
  public Mono<String> clientes(@AuthenticationPrincipal Mono<Usuario> u, Model model) {


    return u.doOnNext(
            usuario -> {
              Flux<Proveedor> proveedorFlux = proveedorService.findAll();
              proveedorFlux =
                  usuario.getDatosRegistro().getAuthority().equals(Rol.ROLE_USER)
                      ? proveedorFlux.filter(x -> x.getRepresentantes().contains(usuario.getId()))
                      : proveedorFlux;

              logger.info("Empresas que representa " + usuario.getNombre() + ":");
              proveedorFlux.subscribe(prod -> logger.info(prod.getNombre()));

              model.addAttribute( s: "empresas", proveedorFlux);
              model.addAttribute( s: "usuario", usuario);
            })
        .then(Mono.just("select-empresa"));
  }

  @GetMapping(value = {"/seleccionar/{id}"})
  public Mono<String> seleccionar(
      @AuthenticationPrincipal Mono<Usuario> u, Model model, @PathVariable(name = "id") String id) {

    // En la sesión mantendremos el idProveedor
    return u.doOnNext(
            usuario -> {
              usuario.setDatosSesion(new DatosSesion(id));
            })
        .then(Mono.just("redirect:/clientes"));
  }
}
```

**Figura 88. SelectorController.java**

Como puede apreciarse en la Figura anterior, este controlador obtiene todos los proveedores, pero los filtra dependiendo del Rol del usuario.

- Cuando el Rol es un usuario normal, se obtienen los proveedores cuyos representantes contengan el usuario que inició sesión
- Cuando el Rol es de administrador, se obtienen todos los proveedores sin distinción. De esta manera, el administrador puede gestionar o consultar cualquier proveedor desde la misma interfaz.

#### 4.2.1.7.2 Clientes

Una vez se selecciona un proveedor, se pasa a la ruta "/clientes", gestionada en su totalidad por el controlador **ClientesController**:

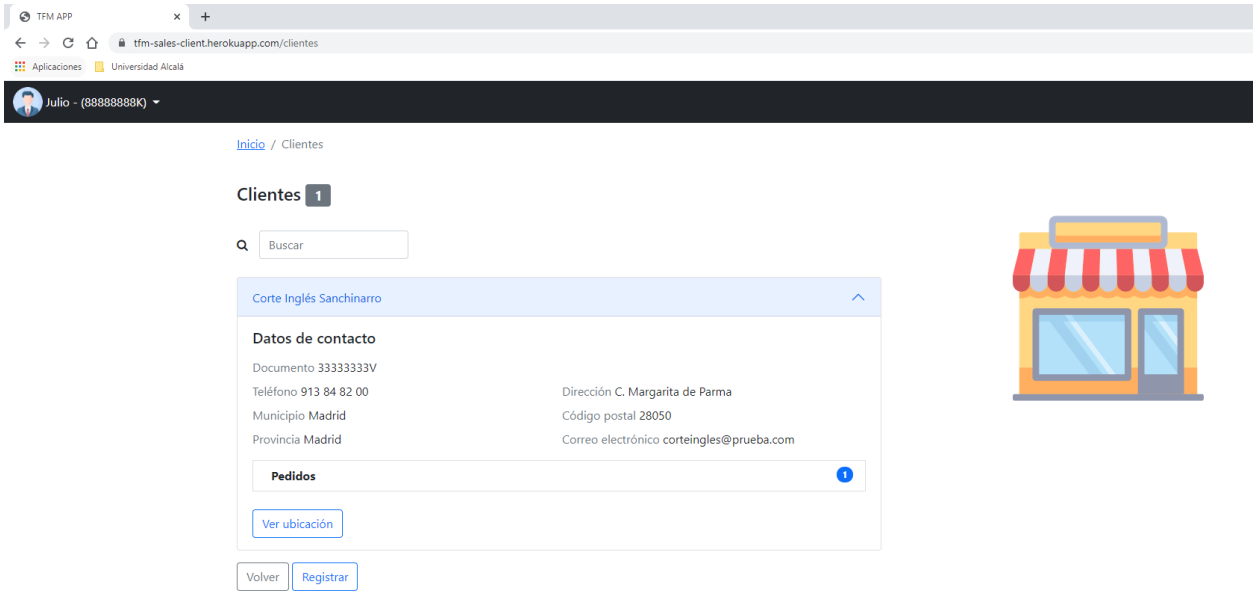



**Figura 89. Clientes (ROL_USER)**

En esta página un usuario puede consultar la información de los clientes que pertenecen al proveedor elegido: el documento, teléfono, datos de dirección, correo electrónico…

Además, pueden consultarse los pedidos como se observa en la Figura anterior, habiendo en este caso un pedido. Estos pedidos sólo son visibles por el usuario normal, ya que el administrador no tendrá acceso a ellos, como se puede apreciar en la **Figura 90**.

Cuando el cliente tiene definidos en la base de datos los campos **latitud** y **longitud**, aparece el botón "Ver ubicación", que despliega mediante un **modal** de Bootstrap un mapa implementado con **Mapbox** mostrando la ubicación exacta del cliente.

Aunque no se vea en la figura, cuando el usuario es un administrador, estos clientes puede modificarlos o darlos de baja de la plataforma:

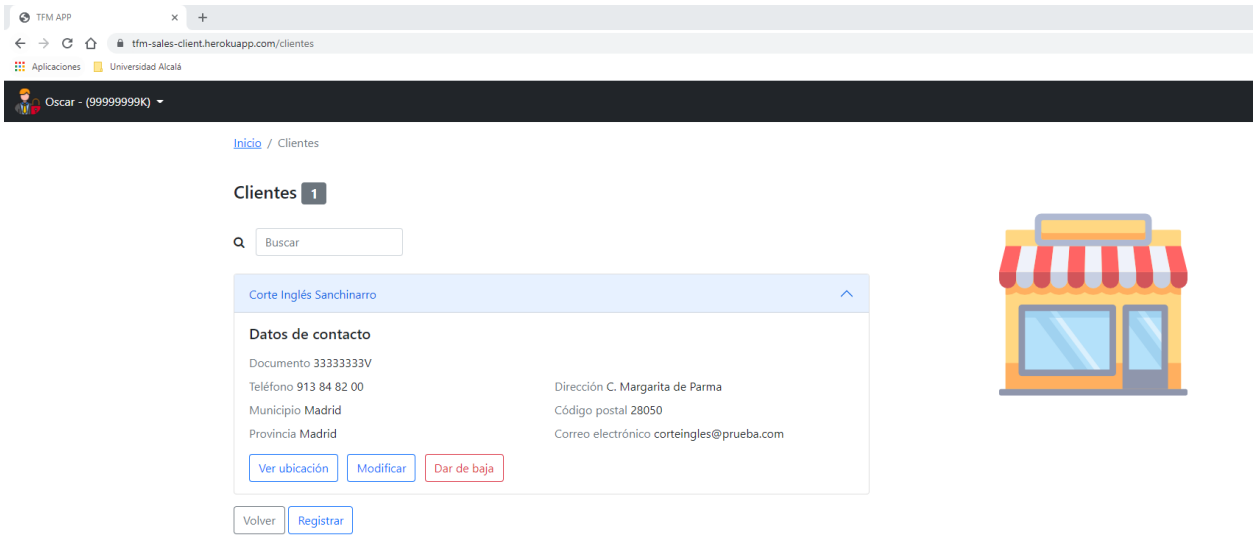



**Figura 90. Clientes (ROL_ADMIN)**

Por último, tanto el usuario como el administrador pueden registrar clientes nuevos o ya existentes en la plataforma insertando su documento de identidad:

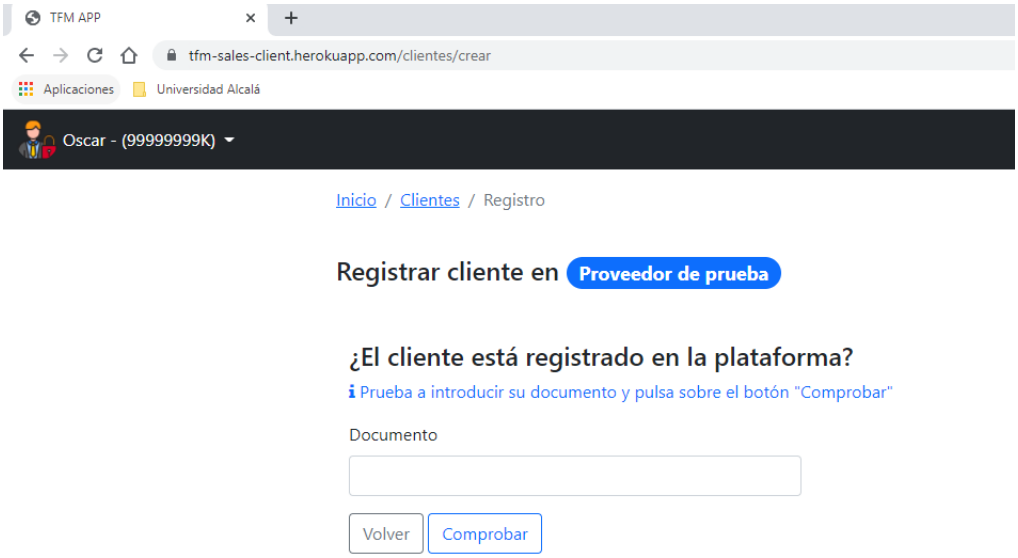



**Figura 91. Registrar cliente**

Esta funcionalidad, como se ha mencionado antes, se explicará con mayor detalle en el **Apéndice A**. Por último, el aspecto del controlador **ClientController** que se encarga de toda la funcionalidad anterior es el siguiente, teniendo en cuenta que es un fragmento, debido a que solamente este controlador ocupa más de 300 líneas de código:

ClientesController.java

```
                        clienteBaja -> {
                          logger.info("ClienteActual a Baja -->" + clienteBaja.getId());
                          model.addAttribute( s: "clienteActual", clienteService.findById(idCliente));
                          model.addAttribute( s: "proveedorActual", clienteBaja);
                        })
                    .then(Mono.just("clientes/bajaCliente")));
    }

    @GetMapping(value = {"/confirmarbaja/{idCliente}"})
    public Mono<String> confirmarbaja(
        @AuthenticationPrincipal Mono<Usuario> u, @PathVariable String idCliente) {

      logger.info("/confirmarbaja");
      return u.doOnNext(usuario -> usuario.getDatosSesion().setIdCliente(idCliente))
          .flatMap(
              usuario ->
                  clienteService
                      .findById(idCliente)
                      .flatMap(
                          cliente -> {
                            cliente
                                .getProveedores()
                                .remove(usuario.getDatosSesion().getIdProveedor());
                            return clienteService
                                .update(cliente)
                                .doOnNext(
                                    c -> {
                                      logger.info(
                                          "Cliente "
                                              + c.getDocumento().getNumero()
                                              + " dado de baja del proveedor"
                                              + usuario.getDatosSesion().getIdProveedor());
                                    });
                          })
                      .then(Mono.just("redirect:/clientes")));
    }

    @GetMapping(value = "/editar/{id}")
    public Mono<String> editar(
        @AuthenticationPrincipal Mono<Usuario> u, Model model, @PathVariable String id) {
      model.addAttribute( s: "usuario", u);

      logger.info("/editar");
      Mono<Cliente> clienteActual = clienteService.findById(id).defaultIfEmpty(new Cliente());
```

**Figura 92. Fragmento de ClientesController**

En concreto, se puede apreciar cómo se realiza la baja del cliente para el proveedor seleccionado, así como parte del método GET encargado de la modificación del cliente.

#### 4.2.1.7.3 Pedidos

Cuando el usuario con el Rol USER selecciona en el recuadro de pedidos de la **Figura 89**, se pasa a la ruta "/pedidos", gestionada por el controlador **PedidosController**. La idea es prácticamente la misma que los clientes:

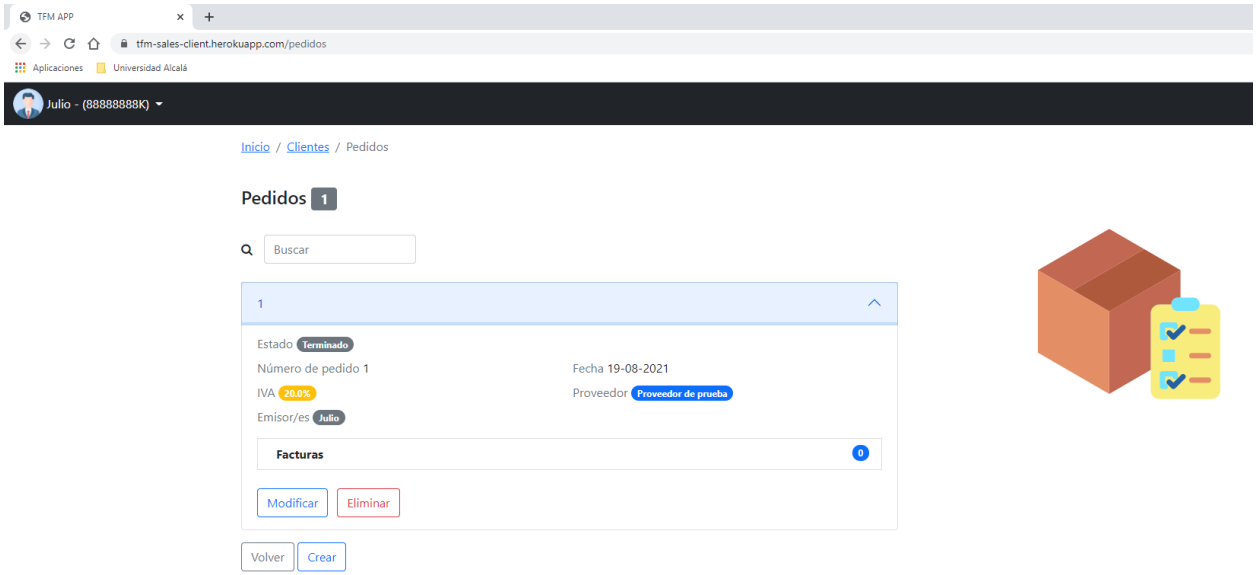



**Figura 93. Pedidos**

En la Figura se puede apreciar la misma estructura que en el caso de los clientes, solo que en los pedidos la información que se muestra es diferente: el estado del pedido, su número, la fecha de emisión, el IVA aplicado para todas sus facturas, el proveedor con el que hemos iniciado que corresponde como es evidente al del pedido, así como los emisores del pedido.

Además, cuando el pedido se encuentra en estado "Terminado", aparece la sección de facturas, que podrá visitarse posteriormente para gestionarlas, aunque esto ya formaría parte del controlador **FacturasController**.

Un usuario puede crear, modificar o eliminar pedidos. A continuación, se muestra el formulario de modificación:

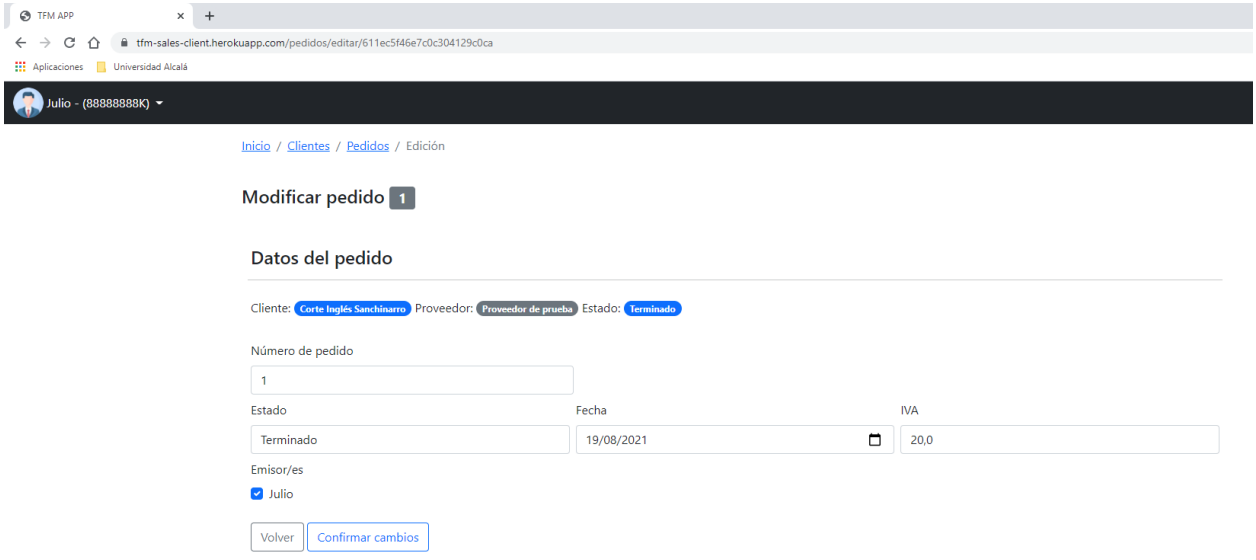



**Figura 94. Modificar pedido**

Y por último, el siguiente fragmento de código corresponde al controlador:

PedidosController.java

```
@PostMapping(value = "/crear")
public Mono<String> crear(@ModelAttribute("pedidoActual") Pedido pedidoActual) {

  logger.info("/crear [POST]");
  return pedidoService
      .save(pedidoActual)
      .doOnNext(
          pedido -> {
            logger.info("Pedido creado: " + pedido.getNumeroPedido() + " Id: " + pedido.getId());
          })
      .onErrorResume(throwable -> Mono.error(throwable))
      .then(Mono.just("redirect:/pedidos"));
}

@PostMapping(value = "/editar")
public Mono<String> editar(
    @AuthenticationPrincipal Mono<Usuario> u,
    Model model,
    @ModelAttribute("pedidoActual") Pedido pedidoActual) {

  logger.info("/editar [POST]");
  return pedidoService
      .update(pedidoActual)
      .doOnNext(
          pedido -> {
            model.addAttribute( s: "usuario", u);
            logger.info(
                "Pedido modificado: " + pedido.getNumeroPedido() + " Id: " + pedido.getId());
          })
      .onErrorResume(throwable -> Mono.error(throwable))
      .then(Mono.just("redirect:/pedidos"));
}

public Mono<Pedido> getPedidoConProveedor(String id) {
  Mono<Pedido> pedidoMono = pedidoService.findById(id);
  return pedidoMono.flatMap(
      pedido -> {
        Mono<Proveedor> proveedorMono = proveedorService.findById(pedido.getProveedor());
        return proveedorMono.map(
            proveedor -> {
              pedido.setProveedorObj(proveedor);
              return pedido;
            });
      });
}
```

**Figura 95. Fragmento de PedidosController**

En este fragmento pueden observarse métodos como obtener el pedido con el proveedor, editar un pedido, o guardar el pedido, todo ello con programación reactiva mediante el marco de trabajo Spring WebFlux.

#### 4.2.1.7.4 Facturas

Por último, cuando el usuario pulsa sobre las facturas cuando el pedido está en estado "Terminado", se redirige la aplicación a la ruta "/facturas", gestionada por el controlador **FacturasController:**

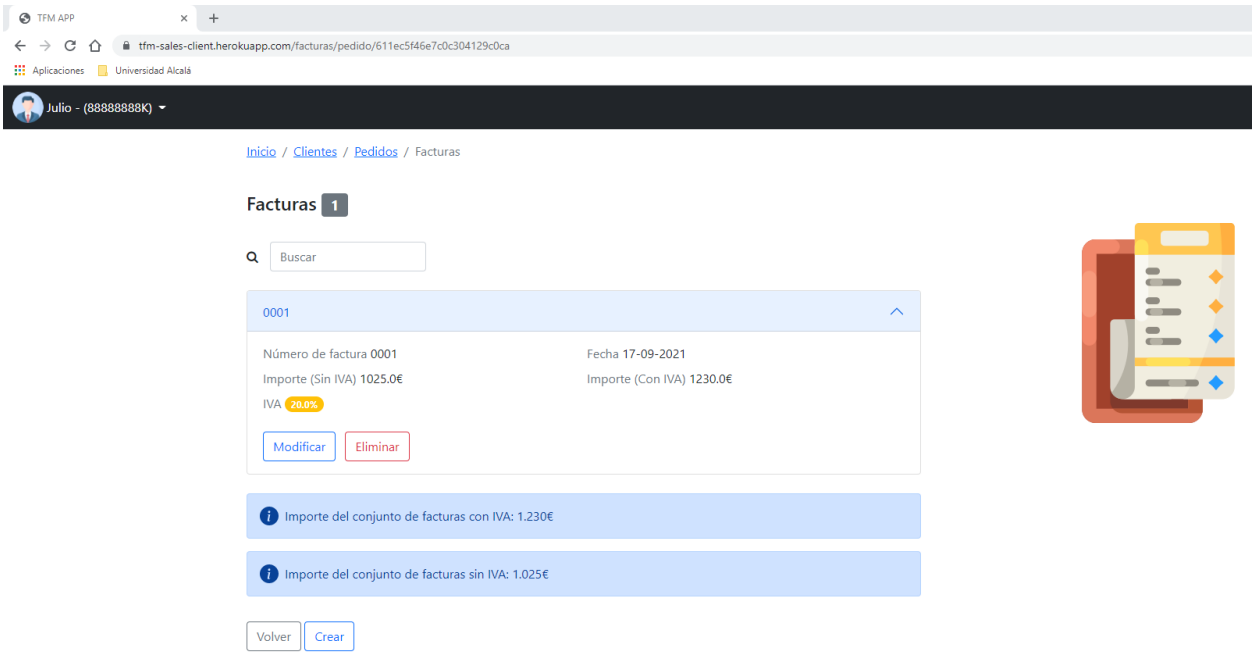



**Figura 96. Facturas**

Facturas también tiene una interfaz muy similar a la de los clientes y pedidos, y la información que se muestra es diferente: el número de la factura, la fecha de creación, el IVA aplicado que proviene del pedido padre, así como el importe con este IVA aplicado o sin aplicar.

Además se informa con advertencias como las de la figura anterior de los importes con y sin IVA de todo el conjunto de facturas. En la siguiente figura se puede comparar que respecto a la anterior, al existir varias facturas, los importes se actualizan y se recalculan en conjunto.

Un usuario puede crear, modificar o eliminar facturas, y de toda la funcionalidad relacionada con ellas se encarga **FacturasController**. A continuación, se muestra el formulario de creación a modo de ejemplo:

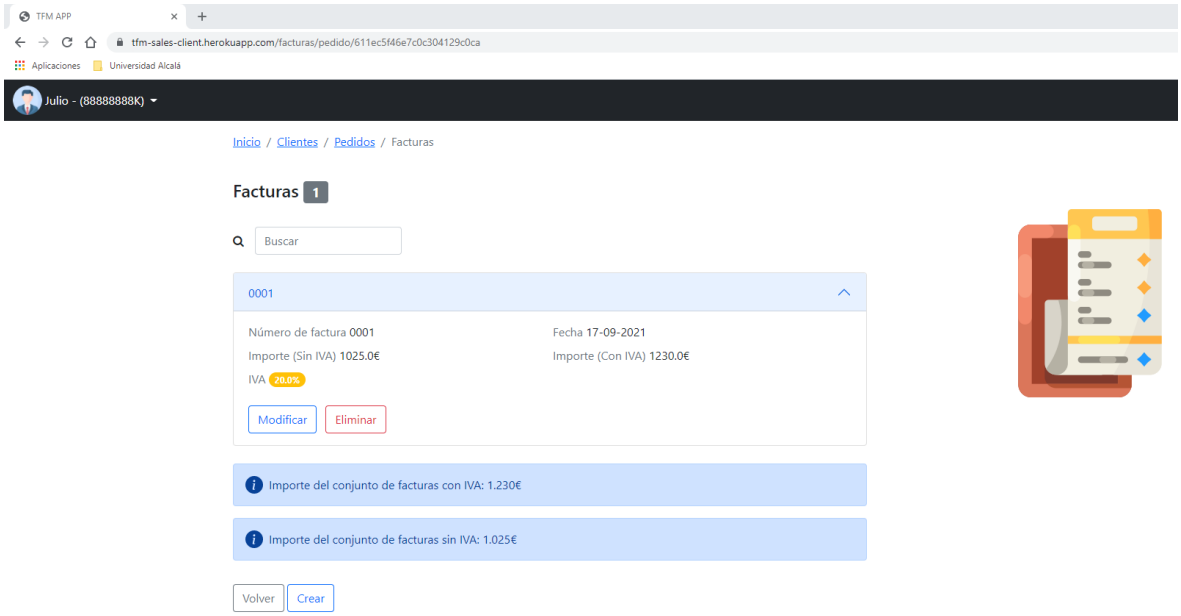



**Figura 97. Crear factura**

La **Figura 98** muestra parte del controlador de las facturas:

FacturasController.java

```
@GetMapping(value = {"/pedido/{idPedido}", ""})
public Mono<String> facturas(
    @AuthenticationPrincipal Usuario u,
    Model model,
    @PathVariable(required = false) String idPedido) {

  logger.info("/facturas");

  // Si el idPedido está nulo o vacío, significa que listemos todas las facturas del Cliente
  // actual
  if (idPedido != null && !idPedido.isEmpty()) u.getDatosSesion().setIdPedido(idPedido);

  Mono<Pedido> pedidoActual = pedidoService.findById(u.getDatosSesion().getIdPedido());
  model.addAttribute( s: "pedidoActual", pedidoActual);

  // Obtenemos el proveedorActual y el clienteActual para el resto de acciones
  model.addAttribute( s: "clienteActual", clienteService.findById(u.getDatosSesion().getIdCliente()));
  model.addAttribute(
      s: "proveedorActual", proveedorService.findById(u.getDatosSesion().getIdProveedor()));

  return facturaService
      .findAll() Flux<Factura>
      .filter(x -> x.getPedido().equals(u.getDatosSesion().getIdPedido()))
      .flatMap(
          factura ->
              pedidoActual.map(
                  pedido -> {
                    factura.setPedidoDto(pedido);
                    return factura;
                  }))
      .collectList() Mono<List<Factura>>
      .doOnNext(
          facturas -> {
            model.addAttribute( s: "usuario", u);
            model.addAttribute( s: "facturas", facturas);
            double totalImporte = 0;
            double totalImporteIVA = 0;
            for (Factura f : facturas) {
              totalImporte += f.getImporte();
              totalImporteIVA += f.getImporteIVA();
            }
```

**Figura 98. Fragmento de FacturasController**

Este fragmento sólo contiene un método encargado de obtener todas las facturas a partir del identificador de un pedido padre. Siguiendo una estrategia similar a la del resto de componentes, mediante los diferentes servicios vamos obteniendo los flujos de datos y añadiéndolos al model para manejarlos en el frontal.

#### 4.2.1.7.5 Proveedores

Esta funcionalidad no es accesible por usuarios estándares o con Rol USER, sino que es una funcionalidad destinada a los administradores.

Para que acceda el administrador a esta funcionalidad, debe utilizar el menú desplegable a partir del icono de usuario:

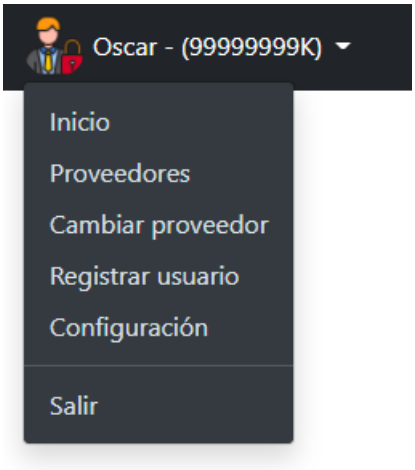



**Figura 99. Menú desplegable (ADMIN)**

Este menú se detallará en siguientes apartados. En concreto, para acceder a los proveedores, el administrador debe pulsar sobre "Proveedores", y aparecerá la siguiente vista:

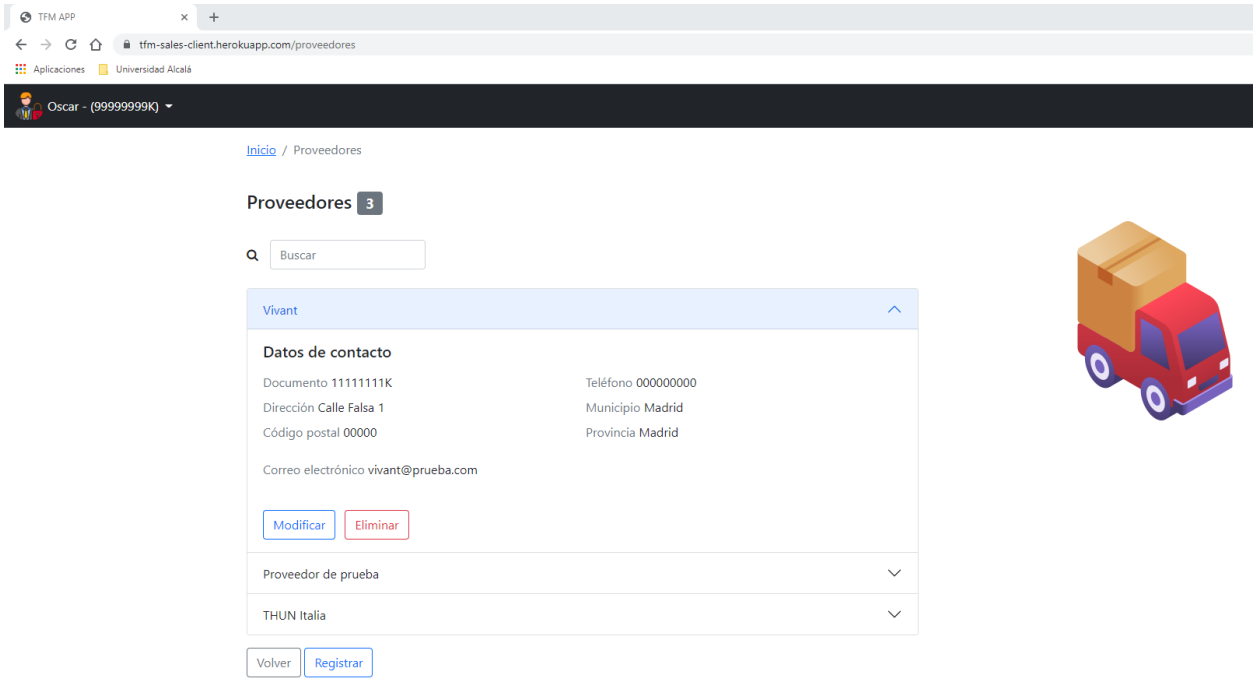



**Figura 100. Proveedores**

En este caso, la información que se muestra es la misma que la de un cliente, pues como se explicó en la sección de *sales-api*, ambas clases heredan de la clase Empresa.

El administrador puede registrar, modificar o eliminar proveedores, y es el responsable de asignar estos a los usuarios para que posteriormente puedan acceder a través del proveedor.

Imaginemos que el administrador crea una empresa llamada **UAH_EMPRESA** a partir del siguiente formulario de registro:

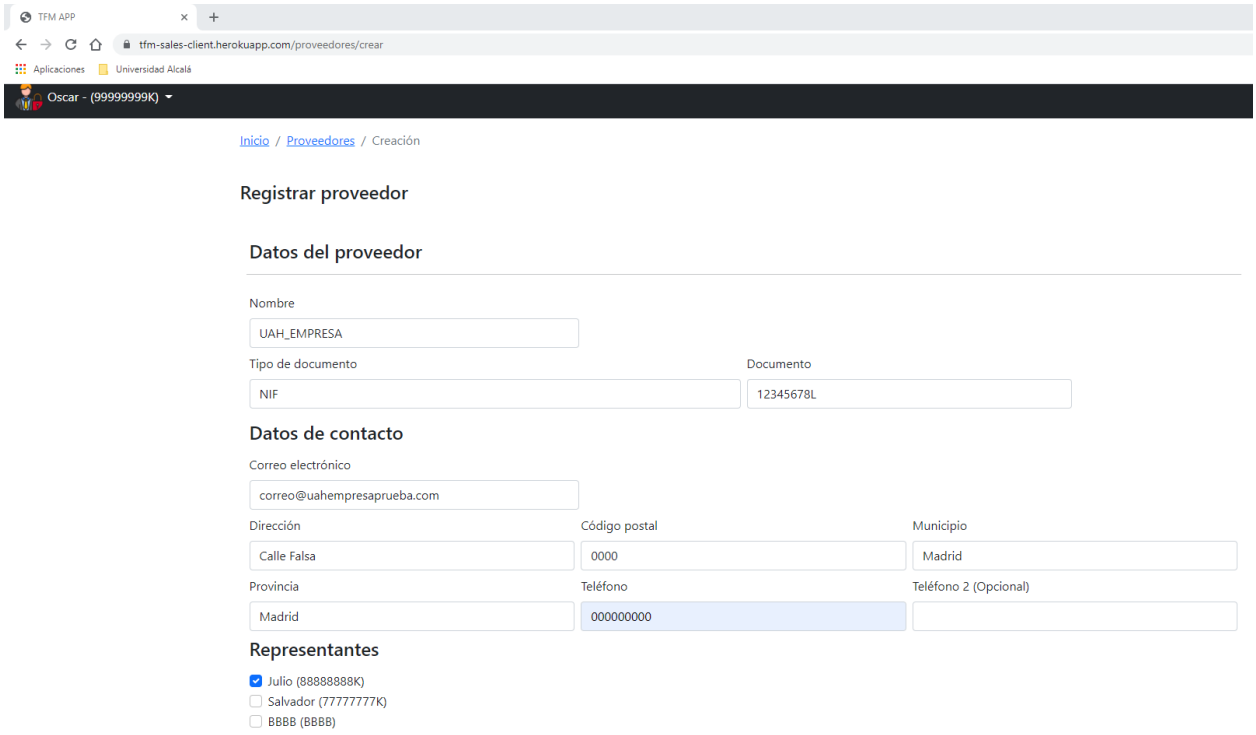



**Figura 101. Registrar proveedor**

La **Figura 101** muestra el formulario con el registro del nuevo proveedor. Existen actualmente controles para que el administrador no registre proveedores con el documento de identidad repetido, al igual que a la hora de registrarse en la plataforma, o registrar clientes. A continuación se muestra un ejemplo del error: aplicado en clientes:

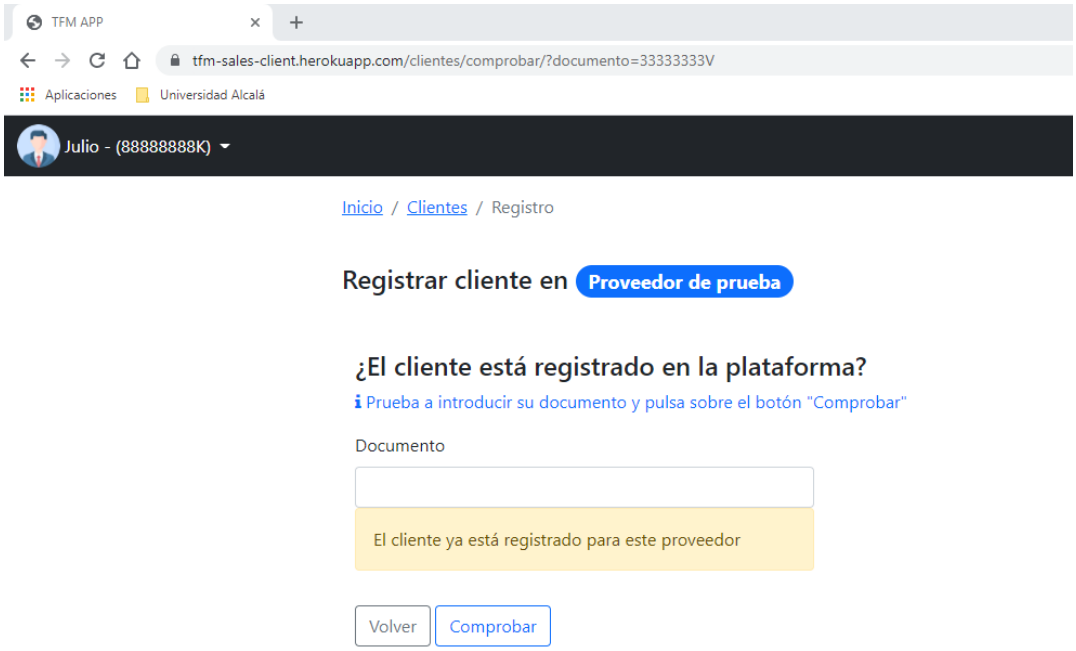



**Figura 102. Ejemplo de comprobación de documento repetido**

ProveedoresController.java

```
83                  throwable -> Mono.error(throwable))
84              .then(Mono.just("redirect:/proveedores"));
85      }
86
87      @GetMapping(value = "/editar/{id}")
88  @   public Mono<String> editar(
89          @AuthenticationPrincipal Usuario u, Model model, @PathVariable String id) {
90          model.addAttribute( s: "usuario", u);
91
92          logger.info("/editar");
93          Mono<Proveedor> proveedorActual = proveedorService.findById(id).defaultIfEmpty(new Proveedor());
94
95          model.addAttribute( s: "proveedorActual", proveedorActual);
96
97          return proveedorActual.flatMap(
98              proveedor -> {
99                  Flux<Usuario> representantes =
100                     usuarioService
101                         .findAll()
102                         .filter(x -> x.getDatosRegistro().getAuthority().equals(Rol.ROLE_USER))
103                         .doOnNext(
104                             representante ->
105                                 logger.info("Representante --> " + representante.getNombre()));
106
107                 model.addAttribute( s: "listarepresentantes", representantes);
108
109                 return Mono.just("proveedores/editarProveedor");
110             });
111     }
112
113     @GetMapping(value = "/eliminar/{id}")
114     public Mono<String> eliminar(
115         @AuthenticationPrincipal Usuario u, Model model, @PathVariable String id) {
116
117         logger.info("/eliminar");
118         return proveedorService
119             .remove(id)
120             .doOnNext(
121                 unused -> {
122                     model.addAttribute( s: "usuario", u);
123                     logger.info("Proveedor " + id + " eliminado con éxito");
124                 })
125             .then(Mono.just("redirect:/proveedores"));
126     }
127
```

**Figura 103. Fragmento de ProveedoresController**

En la figura se observan dos métodos: la edición y la eliminación de los proveedores de forma reactiva.

#### 4.2.1.7.6 Menú desplegable

Como se mencionó anteriormente, al iniciar sesión en la plataforma aparece un icono del usuario dependiendo de su Rol:

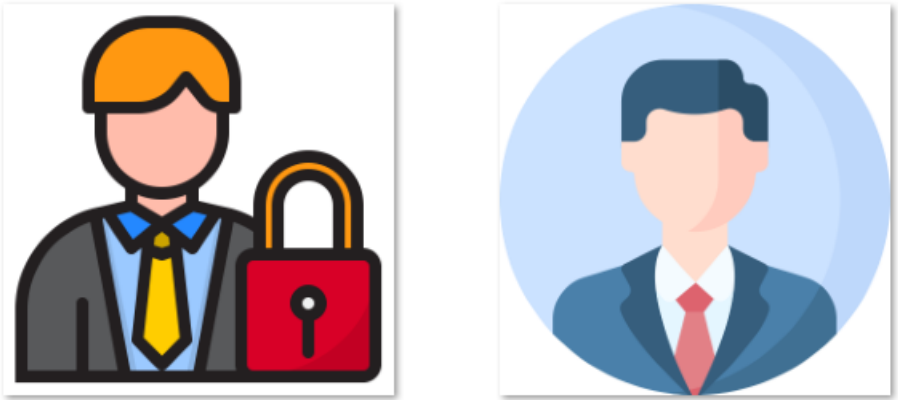


**Figura 104. Iconos de usuarios con Rol ADMIN y USER**

Dependiendo del Rol que tenga el usuario, el menú desplegable ofrecerá unas u otras funcionalidades. La **primera figura** corresponde al menú cuando inicia sesión un usuario con Rol USER, mientras que la **segunda** corresponde al caso del usuario con Rol ADMIN:

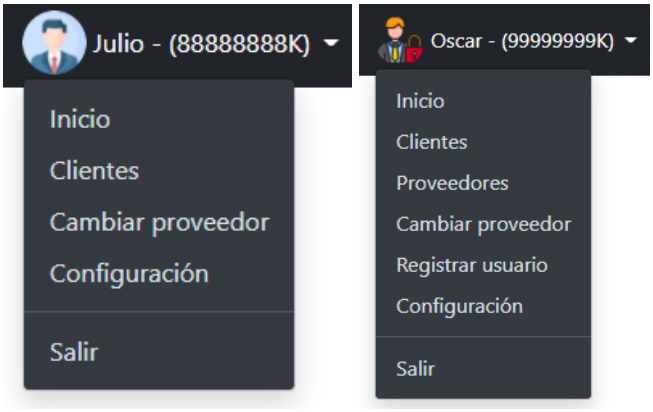



**Figura 105. Menú desplegable dependiendo del Rol del usuario**

Ambos tipos de usuario comparten funciones: Cuando inician sesión y seleccionan un proveedor, les aparece el enlace de **Clientes**. Pueden cambiar de proveedor en cualquier momento, o configurar sus datos personales mediante el enlace **Configuración**, que se explicará en el siguiente punto.

Además, el administrador puede ver los proveedores y registrar usuarios con el Rol que desee, pues como se ha visto anteriormente, en el registro de usuarios no se puede seleccionar Rol, registrándose por defecto con el Rol USER.

#### 4.2.1.7.7 Configuración

En cada uno de los menús el usuario tenía el enlace a **Configuración**, un formulario idéntico al de registro pero utilizado para modificar los datos de un usuario ya registrado. Como punto adicional, el administrador tiene la opción de cambiar su Rol. Esta es la interfaz de un usuario:

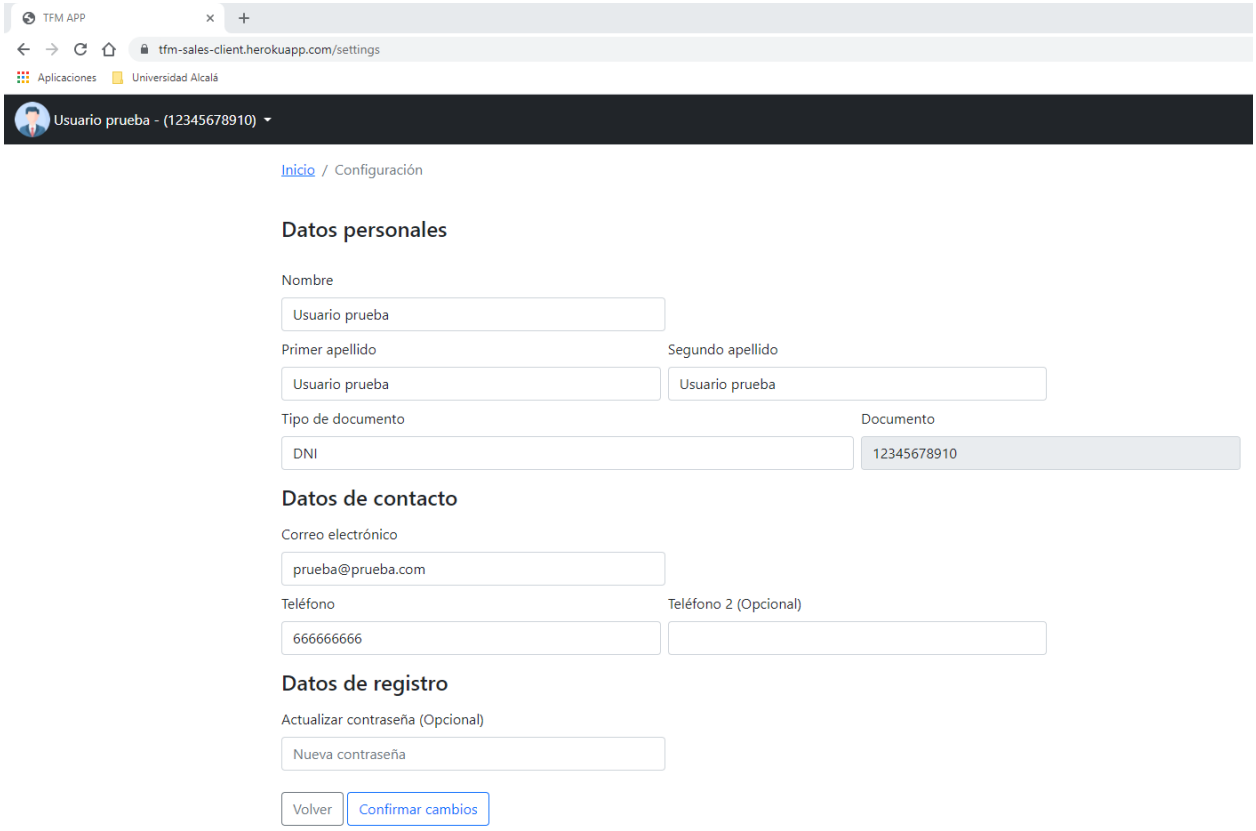



**Figura 106. Configuración (USER)**

Como se puede observar, no es posible modificar el documento de identidad, ya que si un usuario se registra, se interpreta que se registra con un documento único y válido.

Esta ventana corresponde a la ruta "/Settings", y quien se ocupa de su funcionalidad es el controlador **SettingsController**, mostrado a continuación:

SettingsController.java

```java
package es.uah.salesclient.controller;

import ...

@Controller
@RequestMapping(value = "/settings")
@SessionAttributes({"usuario"})
public class SettingsController {

  @Autowired IUsuarioService usuarioService;

  @Autowired PasswordEncoder encoder;

  private final Logger logger = LoggerFactory.getLogger(SettingsController.class);

  @GetMapping(value = "")
  public Mono<String> settings(@AuthenticationPrincipal Usuario u, Model model) {

    logger.info("/settings");
    model.addAttribute( s: "usuario", u);
    return Mono.just("settings");
  }

  @PostMapping(value = "/update")
  public Mono<String> update(
      @AuthenticationPrincipal Usuario u, Model model, @ModelAttribute("usuario") Usuario usuario) {

    logger.info("/update [POST]");

    if (usuario.getDatosRegistro().getNewPassword() != null
        && !usuario.getDatosRegistro().getNewPassword().isEmpty()) {
      String passwordEncoded = encoder.encode(usuario.getDatosRegistro().getNewPassword());
      usuario.getDatosRegistro().setPassword(passwordEncoded);
    }

    return usuarioService
        .update(usuario)
        .doOnNext(
            usuarioActualizado -> {
              logger.info("Usuario " + usuarioActualizado.getNombre() + " actualizado con éxito");
              model.addAttribute( s: "update",  o: "ok");
            })
        .then(Mono.just("settings"));
  }
}
```

**Figura 107. SettingsController**

Este es un controlador bastante simple, ya que únicamente tiene que actualizar los datos del usuario mediante el método de servicio **usuarioService.**

#### 4.2.1.8 Model

Por último, al igual que el microservicio *sales-api*, este microservicio también dispone de un paquete **model**, pero en vez de encargarse de interactuar con la base de datos, ocupa la funcionalidad DTO: Data transfer Object. Estas clases son idénticas a las de *sales-api*, con la diferencia de que son más simples al no tener que poner etiquetados de mongoDB, o de otras configuraciones.

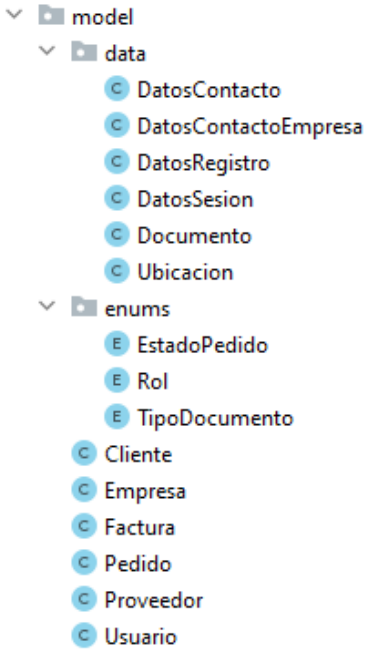



**Figura 108. Paquete model (*sales-client)***

Basta con tener una clase cuyas variables coincidan con las de *sales-api* (para que los datos transferidos se asienten en este objeto) y se pueden añadir todas las variables que se necesiten para posteriormente utilizarlas durante la lógica de negocio o en el front. La siguiente figura corresponde a la clase Factura:

Factura.java

```java
package es.uah.salesclient.model;

import ...

public class Factura {

  private String id;

  private String numeroFactura;
  private Double importe;

  @DateTimeFormat(pattern = "yyyy-MM-dd")
  private Date fechaFactura;

  private String pedido;
  private Pedido pedidoDto;
  private String cliente;
  private String proveedor;

  public Factura() {}

  public String getId() { return id; }

  public void setId(String id) { this.id = id; }

```

**Figura 109. DTO de Factura**

Aunque esté recortada la figura, esta clase tiene todos los métodos get y set por cada variable utilizada. Un ejemplo de variable que usa esta clase pero no corresponde al objeto de la colección que maneja *sales-api* es **pedidoDto**. Esto se utiliza para, durante la lógica de negocio en *sales-client*, tener un objeto factura con su pedido padre.

En esta figura también se aprecia la utilización de **@DateTimeFormat**, muy frecuente cuando necesita utilizarse el tipo Date, ya que suele traer problemas de compatibilidad por su formato.

# INTEGRACIÓN Y PRUEBAS

## 5.1 Integración en producción

Una vez desarrollada la aplicación *Sales*, se eligió **Heroku** como alternativa para realizar un despliegue de los dos microservicios en la nube y que así se pudiese probar la aplicación desde un entorno que no fuese **local**. Por otra parte, para el despliegue de la base de datos se utilizó **MongoDB Atlas.** Ambos montajes se explicarán en el **Apéndice A,** en los manuales relacionados con el montaje, que concretamente corresponde al punto **A.3**.

Actualmente, los microservicios son accesibles desde las siguientes direcciones:

a) *Sales-api:* https://tfm-sales-api.herokuapp.com/

b) *Sales-client:* https://tfm-sales-client.herokuapp.com/

Mientras que la base de datos conecta de forma interna con *sales-api* a través de las variables de entorno configuradas en Heroku, explicadas en el **Estado del Arte**.

## 5.2 Pruebas funcionales

Una vez desplegada *Sales* en la nube y siendo accesible por cualquier usuario, se informó a una serie de voluntarios que utilizasen la aplicación para realizar una serie de pruebas específicas. Algunas de estas pruebas funcionales dieron resultados negativos o sugerencias por parte de los clientes, con lo que las acciones que se tomaron después fueron **corregir incidencias** o **anotar sugerencia**. Las pruebas se realizaron para los dos tipos de usuario , por lo que para las funcionalidades en común se verificó el funcionamiento en ambos Roles.

Las pruebas consistieron en lo siguiente:

| Descripción de las pruebas | Resultados | Consecuencias |
|---|---|---|
| Registro:<br>1) Registrarse<br>2) Repetir con el mismo documento<br>3) Con campos vacíos | **1.** OK<br>**2.** ERROR<br>**3.** OK | Corregido mediante el Toast mencionado en apartados anteriores. |
| Iniciar sesión:<br>4) Credenciales erróneas<br>5) Con campos vacíos | **4.** OK<br>**5.** OK | |
| Seleccionar proveedor:<br>6) Seleccionar proveedor<br>7) Salir de la aplicación | **6.** OK<br>**7.** ERROR | |
| Clientes:<br>8) Búsqueda<br>9) Modificar<br>10) Registrar nuevo<br>11) Registrar existente<br>12) Con campos vacíos<br>13) Ver ubicación<br>14) Baja<br>15) Enlace a pedidos | **8.** ERROR<br>**9.** OK<br>**10.** OK<br>**11.** ERROR<br>**12.** OK<br>**13.** SUGERENCIA<br>**14.** OK<br>**15.** OK | La búsqueda se modificó por jQuery cambiando la estrategia para buscar por todo el contenido del cliente en vez de sólo por identificador, que es como fallaba. Cuando el usuario intentaba registrar un cliente que ya existía en el proveedor, fallaba, ahora se comprueba si ese ya existe y le impide avanzar.<br><br>Un usuario que era programador sugirió que el APIKEY de MapBox no estuviese incrustado como literal en el navegador, por lo que se optó por utilizar una |

| | | |
|---|---|---|
| | | variable privada traída desde Heroku. |
| Proveedores:<br>16) Búsqueda<br>17) Modificar<br>18) Registrar nuevo<br>19) Registrar existente<br>20) Con campos vacíos<br>21) Eliminación<br>22) Acceder a la ruta con usuario de Rol USER | **16. ERROR**<br>**17. OK**<br>**18. OK**<br>**19. ERROR**<br>**20. OK**<br>**21. OK**<br>**22. ERROR** | En cuanto a la búsqueda, se aplicó el mismo cambio que en la prueba anterior, ya que el componente era el mismo y por lo tanto, el error también.<br>Ocurrió lo mismo en el registro de un existente, ya que no comprobaba que hubiese proveedores registrados con el mismo documento.<br>Respecto al error 22, se corrigió modificando la configuración de Spring Security para que la ruta no fuese accesible si el usuario que intentase acceder fuese ADMIN. |
| Pedidos:<br>23) Búsqueda<br>24) Modificar<br>25) Registrar nuevo<br>26) Con campos vacíos<br>27) Eliminación | **23. ERROR**<br>**24. OK**<br>**25. OK**<br>**26. OK**<br>**27. INCIDENCIA** | En cuanto a la búsqueda, se aplicó el mismo cambio que en la prueba anterior, ya que el componente era el mismo y por lo tanto, el error también.<br>Con el tiempo se observó que al eliminar los pedidos, no se eliminaba en cascada, es decir, no eliminaba las facturas hijas. |
| Facturas:<br>28) Búsqueda<br>29) Modificar<br>30) Registrar nueva<br>31) Con campos vacíos<br>32) Eliminación | **28. ERROR**<br>**29. OK**<br>**30. OK**<br>**31. OK**<br>**32. OK** | En cuanto a la búsqueda, se aplicó el mismo cambio que en la prueba anterior, ya que el componente era el mismo y por lo tanto, el error también. |
| Menú desplegable:<br>33) Cambiar proveedor<br>34) Registrar usuario<br>35) Clientes | **33. OK**<br>**34. OK**<br>**35. OK** | |

| | | |
|---|---|---|
| 36) Configuración<br>37) Cerrar sesión | **36. OK**<br>**37. OK** | |
| *Sales-api:*<br>38) No es accesible de forma externa | **38. OK** | |
| Otros:<br>39) Acceder a rutas inexistentes<br>40) Acceso de varios usuarios a la vez (pruebas de estrés)<br>41) Probar desde dispositivos diferentes | **39. ERROR**<br>**40. OK**<br>**41. SUGERENCIA** | El error 39 fue por lo que se tomó posteriormente la decisión de sobrescribir las páginas de error creando propias para cada tipo de error. De esta forma, no salía una página de error no controlado.<br>La sugerencia 41 vino de un usuario que comentaba que las imágenes parecían adaptarse al dispositivo, pero "seguían siendo de gran tamaño". Esta sugerencia fue recogida en Kanban para su futura implementación. |

**Figura 110. Pruebas funcionales**

Las pruebas anteriores fueron en parte, pruebas que realizaron usuarios ajenos al trabajo. Es evidente que durante la fase anterior de **Desarrollo** se realizaron también pruebas similares por cada funcionalidad, corrigiendo otros fallos que posteriormente no aparecieron durante el **Mantenimiento.**

## 5.3 Incidencias y mejoras

Durante la fase de desarrollo se detectaron incidencias que fueron corregidas, y durante el mantenimiento de *Sales*, que comenzó oficialmente una vez realizado los despliegues de los microservicios en la nube, se empezaron a realizar las **pruebas funcionales** mencionadas anteriormente. Todo ello resultó ser de vital importancia para la detección y corrección de **incidencias**.

Por otra parte, a la vez que surgían incidencias, también aparecían ideas de **mejorar** la aplicación mediante nuevas funcionalidades o correcciones.

A continuación se mostrarán algunas de estas incidencias y mejoras que fueron recogidas en el tablero **Kanban** de Trello**.** Cada una de estas tarjetas contienen un título, descripción del problema y lista de subtareas a realizar:


**Figura 111. Incidencia Trello 1**


**Figura 112. Incidencia Trello 2**


**Figura 113. Incidencia Trello 3**

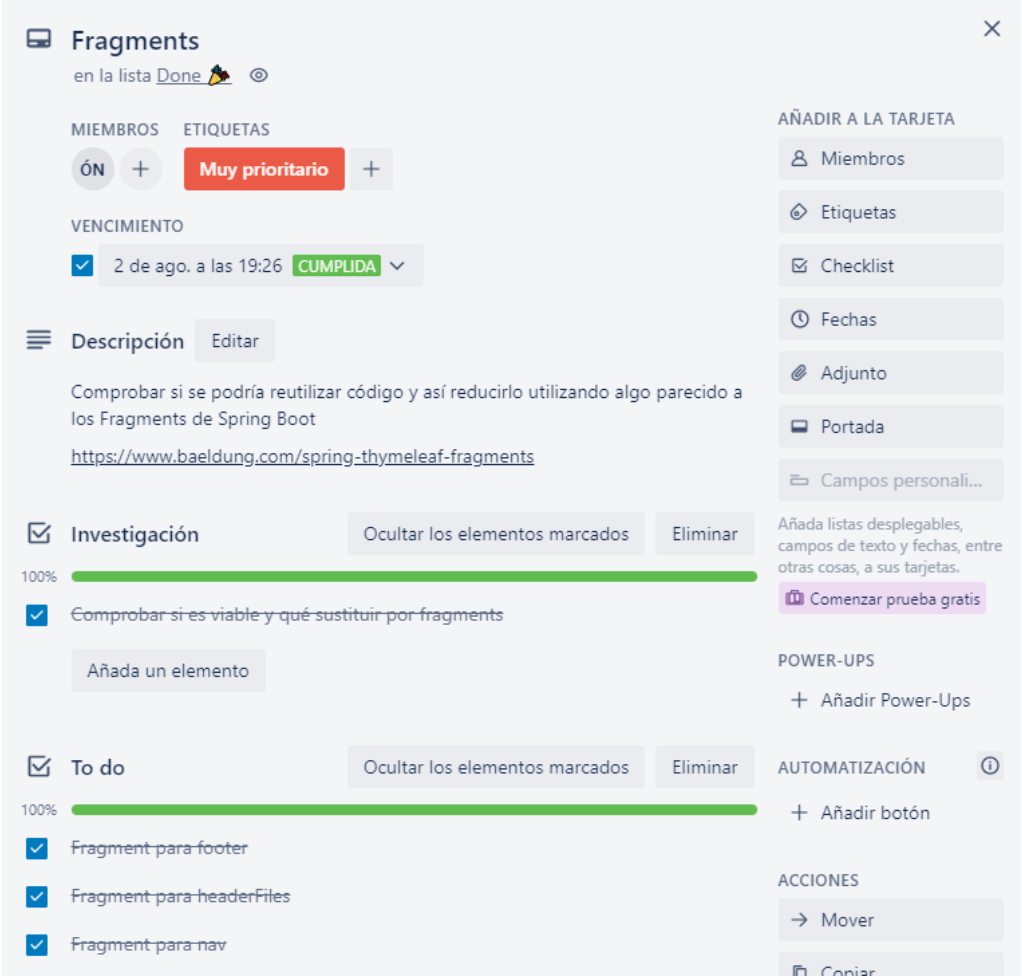



**Figura 114. Mejora Trello 1**

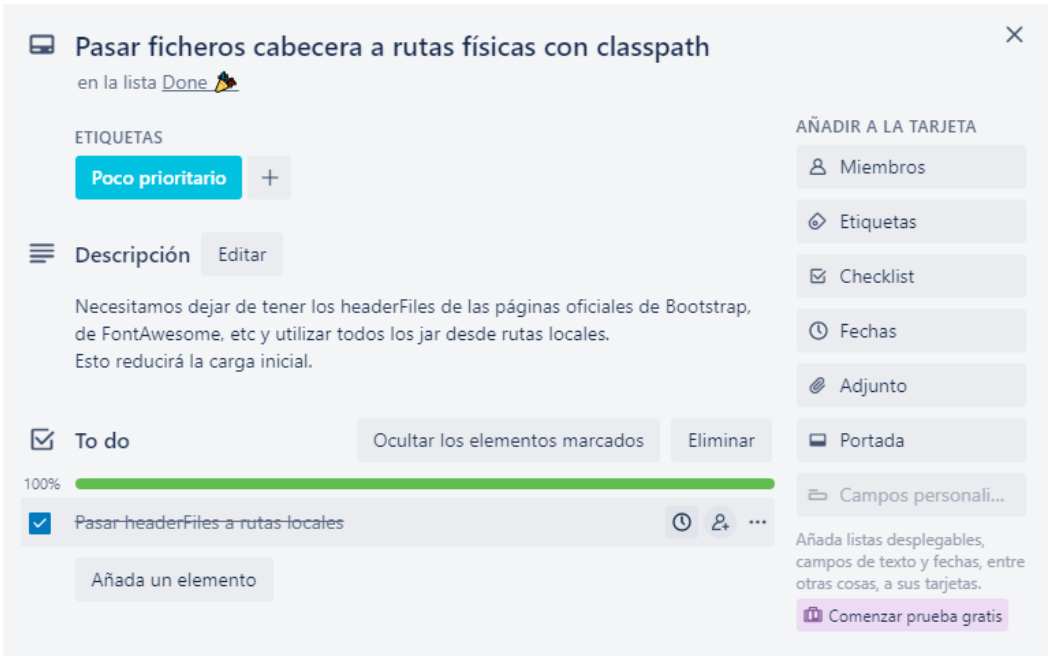



**Figura 115. Mejora Trello 2**

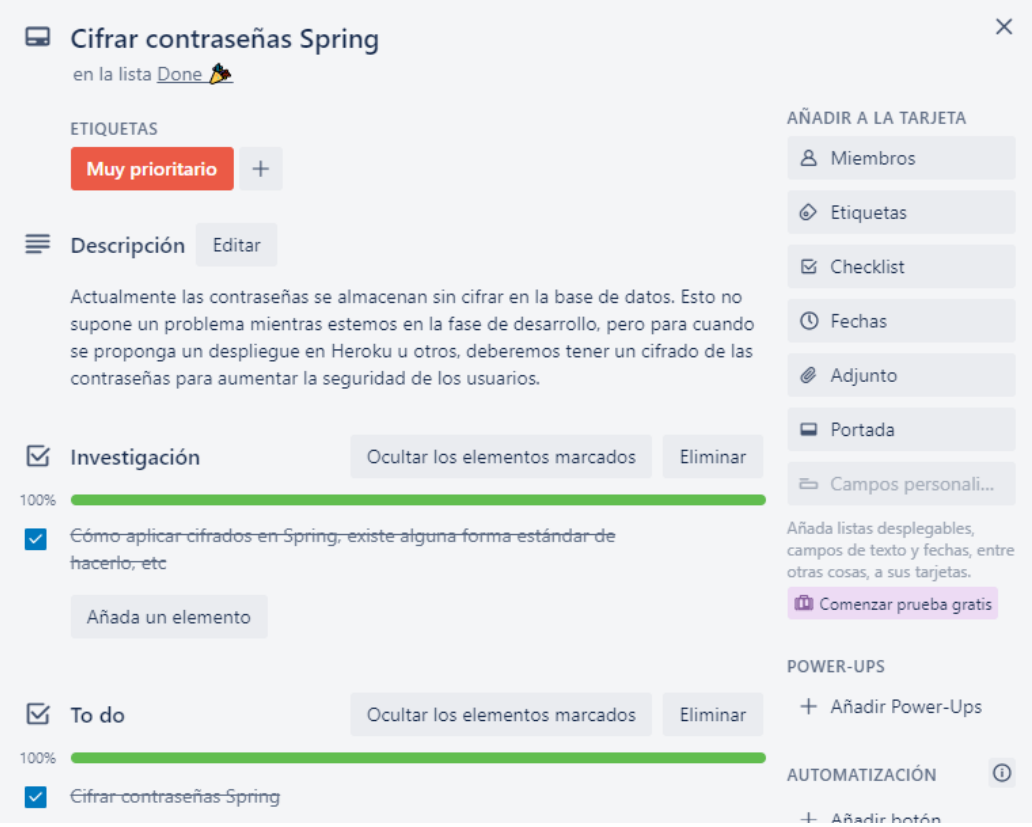



**Figura 116. Mejora Trello 3**


**Figura 117. Mejora Trello 4**

Como se puede observar, todos los ejemplos expuestos son actualmente parte de *Sales*, ya que fueron completados durante las fases de desarrollo y mantenimiento.

# RESUMEN Y CONCLUSIÓN

## 6.1 Resumen

*Sales* nace como un proyecto de Trabajo de Fin de Máster con muchas posibilidades para los comerciales autónomos. Intenta agilizar los procedimientos que suelen necesitar a la hora de gestionar cada proveedor con sus clientes, pedidos y facturas.

Se han conseguido desarrollar el conjunto de microservicios que intervienen en *Sales* con programación reactiva gracias a Spring WebFlux, marco de trabajo que ha demostrado ser muy eficiente a la hora de realizar las peticiones no bloqueantes en el tiempo.

Es evidente que *Sales* no es más que el comienzo de algo mucho más grande.

## 6.2 Conclusiones y Futuras Líneas de Trabajo

Los conocimientos adquiridos en este Trabajo Fin de Máster han sido muy satisfactorios. Jamás había programado con un marco reactivo en Java. Únicamente me limité a utilizar marcos como Angular o Vue, sin saber que Java y Spring tenían tantas posibilidades en la programación reactiva.

Es verdad que con Spring WebFlux se limitan las posibilidades, ya que no es tan completo como Spring Framework, y además, considero que desarrollar una misma aplicación cuesta menor trabajo en Spring Framework que en Spring WebFlux.

Sin embargo, creo firmemente en que llegará muy lejos por todas esas posibilidades que puede alcanzar, y además, puede lograr ser un marco muy atractivo si se combina con Vue o Angular en la parte frontal, en vez de utilizar Thymeleaf de Spring 5 como he hecho en este trabajo.

Por este motivo las posibles líneas de futuro que habría para mejorar la aplicación serían implementar un frontal como Vue para reutilizar componentes. Reduciría muchas líneas de código y sería más sencillo.

Otra posible mejora sería implementar tiempos de sesión activa mediante el uso de tokens de expiración como JWT [60].

# BIBLIOGRAFÍA

**[1]** Alex Rodríguez, *"Servicios Web de RESTful: Los aspectos básicos"*, IBM Developer, febrero 2015, https://developer.ibm.com/es/articles/ws-restful/

**[2]** MongoDB, https://www.mongodb.com/es

**[3]** Java (Lenguaje de programación), https://es.wikipedia.org/wiki/Java_(lenguaje_de_programaci%C3%B3n)

**[4]** Baeldung, *"Spring 5"*, Baeldung, julio 2021, https://www.baeldung.com/spring-5

**[5]** Cecilio Álvarez Caules, *"¿Qué es Spring WebFlux?"*, Arquitectura Java, diciembre 2019, https://www.arquitecturajava.com/que-es-spring-webflux/

**[6]** Jonathan Faber Romero, *"Introducción a Spring Security"*, Adictos al trabajo, mayo 2020, https://www.adictosaltrabajo.com/2020/05/21/introduccion-a-spring-security/

**[7]** Robo 3T, https://robomongo.org/

**[8]** MongoDB Atlas, https://www.mongodb.com/es/cloud/atlas

**[9]** *"¿Qué son los microservicios?"*, https://aws.amazon.com/es/microservices/

**[10]** The Heroku Platform, https://www.heroku.com/platform

**[11]** Amrita Pathak, *"¿Qué es GitLab y dónde alojarlo?"*, Geekflare, febrero 2021, https://geekflare.com/es/gitlab-hosting/

**[12]** IntelliJ IDEA, https://www.jetbrains.com/es-es/idea/

**[13]** Craig Walls, *"Spring Boot IN ACTION"*, Manning, diciembre 2015, 265 páginas

**[14]** Spring Framework, https://spring.io/projects/spring-framework

**[15]** Gradle Build Tool, https://gradle.org/

**[16]** Apache Maven Project, https://maven.apache.org/

**[17]** Tomcat, https://es.wikipedia.org/wiki/Tomcat

**[18]** Jetty, https://es.wikipedia.org/wiki/Jetty

**[19]** Netty Project, https://netty.io/

**[20]** Spring Initializr, https://start.spring.io/

**[21]** Cecilio Álvarez Caules, *"Spring Boot Starter, un concepto fundamental"*, Arquitectura java, julio 2019, https://www.arquitecturajava.com/spring-boot-starter-un-concepto-fundamental/

**[22]** Baeldung, *"Spring Data Reactive Repositories with MongoDB"*, Baeldung, septiembre 2021, https://www.baeldung.com/spring-data-mongodb-reactive

**[23]** Alejandro Ladera, *"La programación reactiva en Spring"*, Deloitte, https://www2.deloitte.com/es/es/pages/technology/articles/la-programacion-reactiva-en-spring.html

**[24]** Eduardo Patiño Rodríguez, *"¿Qué es la programación asíncrona y síncrona?"*, AnexSoft, diciembre 2019, https://anexsoft.com/que-es-la-programacion-asincrona-y-sincrona

**[25]** Concurrencia (informática), https://es.wikipedia.org/wiki/Concurrencia_(inform%C3%A1tica)

**[26]** Java Servlets 3.1, WebSphere Application Server Liberty, IBM, https://www.ibm.com/docs/es/was-liberty/nd?topic=features-java-servlets-31

**[27]** Undertow, https://undertow.io/

**[28]** Project Reactor, https://projectreactor.io/

**[29]** Cecilio Álvarez Caules, *"Flux vs Mono, Spring y la programación reactiva"*, Arquitectura java, junio 2019, https://www.arquitecturajava.com/flux-vs-mono-spring-y-la-programacion-reactiva/

**[30]** Spring Framework Documentation, https://docs.spring.io/spring-framework/docs/current/reference/html/web-reactive.html

**[31]** Reactive Streams Support, https://docs.spring.io/spring-integration/reference/html/reactive-streams.htm

**[32]** MySQL, https://www.mysql.com/

**[33]** Andrés Araujo, *"¿Qué es una base de datos NoSQL?"*, Blogs Oracle, abril 2016, https://blogs.oracle.com/spain/qu-es-una-base-de-datos-nosql

**[34]** Interface ReactiveMongoRepository<T,ID>, https://docs.spring.io/spring-data/mongodb/docs/current/api/org/springframework/data/mongodb/repository/ReactiveMongoRepository.html

**[35]** *"Clústeres para computación técnica a gran escala en la nube"*, Google Cloud, https://cloud.google.com/architecture/using-clusters-for-large-scale-technical-computing?hl=es

**[36]** *"Qué es SQL, sintaxis, conceptos básicos y características"*, bigdata-analytics, https://bigdata-analytics.es/sql/

**[37]** María Tena, *"¿Qué es la metodología 'agile'"?*, BBVA, agosto 2020, https://www.bbva.com/es/metodologia-agile-la-revolucion-las-formas-trabajo/

**[38]** *"¿Qué es el big data?"*, Oracle, https://www.oracle.com/es/big-data/what-is-big-data/

**[39]** Class BCryptPasswordEncoder, https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/crypto/bcrypt/BCryptPasswordEncoder.html

**[40]** Basic Access authentication, https://en.wikipedia.org/wiki/Basic_access_authentication

**[41]** Baeldung, *"The DAO Pattern in Java"*, Baeldung, marzo 2020, https://www.baeldung.com/java-dao-pattern

**[42]** Baeldung, *"Introduction to Spring Data JPA"*, Baeldung, julio 2021, https://www.baeldung.com/the-persistence-layer-with-spring-data-jpa

**[43]** Baeldung, *"Guide to Spring @Autowired"*, Baeldung, septiembre 2021, https://www.baeldung.com/spring-autowire

**[44]** *"Spring Boot @Repository"*, ZetCode, julio 2020, https://zetcode.com/springboot/repository/

**[45]** Class RouterFunctions, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/server/RouterFunctions.html

**[46]** *"¿Qué son los Beans?"*, Java desde 0, julio 2019, https://javadesde0.com/introduccion-a-los-beans/

**[47]** Class RequestPredicates, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/server/RequestPredicates.html

**[48]** Interface HandlerFunction<T extends ServerResponse>, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/server/HandlerFunction.html

**[49]** Annotation Type EnableWebFluxSecurity, https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/config/annotation/web/reactive/EnableWebFluxSecurity.html

**[50]** Annotation Type Value, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/beans/factory/annotation/Value.html

**[51]** Interface WebFluxConfigurer, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/config/WebFluxConfigurer.html

**[52]** Bootstrap 5, https://getbootstrap.com/docs/5.0/getting-started/introduction/

**[53]** Font Awesome, https://fontawesome.com/

**[54]** jQuery, https://jquery.com/

**[55]** MapBox, https://www.mapbox.com/

**[56]** Flaticon, https://www.flaticon.com/

**[57]** Interface WebClient, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/reactive/function/client/WebClient.html

**[58]** Interface ClientHttpConnector, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/client/reactive/ClientHttpConnector.html

**[59]** ReactorClientHttpConnector, https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/http/client/reactive/ReactorClientHttpConnector.html

**[60]** JWT, https://jwt.io/

**[61]** Baeldung, *"Working with Fragments in Thymeleaf"*, Baeldung, diciembre 2020, https://www.baeldung.com/spring-thymeleaf-fragments

**[62]** Dataedo, https://dataedo.com/

# APÉNDICE A. MANUALES DE LA APLICACIÓN

# A.1 Manual de usuario

Con el objetivo de facilitar la utilización de la plataforma para los usuarios estándar (no administradores), se adjunta a continuación el enlace a un vídeo tutorial, explicando cada una de las funcionalidades con ejemplos prácticos: https://youtu.be/2uVoFnovi5U

## A.2 Manual de administrador

Con el objetivo de facilitar la utilización de la plataforma para los usuarios administradores, se adjunta a continuación el enlace a un vídeo tutorial, explicando cada una de las funcionalidades con ejemplos prácticos: https://youtu.be/f8Te1ZRCToc

# A.3 Manual de montaje

Con el objetivo de facilitar el montaje de la aplicación en local se explicará en detalle cómo arrancar la aplicación *Sales*.

En primer lugar, la aplicación se divide en dos microservicios: sales-api y sales-client. Ambos están desarrollados con el **JDK 15.0.1**, como se aprecia en la siguiente figura:

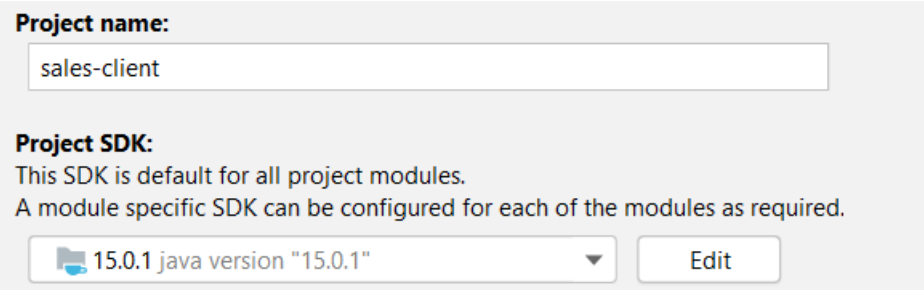



**Figura 118. JDK del proyecto sales-client**

Por lo tanto, el primer paso para un desarrollador es configurar este JDK. Para ello, se debe descargar del siguiente enlace https://www.oracle.com/java/technologies/javase/jdk15-archive-downloads.html y buscar "Java SE Development Kit 15.0.1". Dependiendo del sistema operativo, se debe elegir una u otra versión. Suponiendo que se hace en Windows, se debe instalar el archivo y una vez instalado, se tienen que modificar las variables de entorno. Para ello, hay que entrar en "Propiedades del sistema":

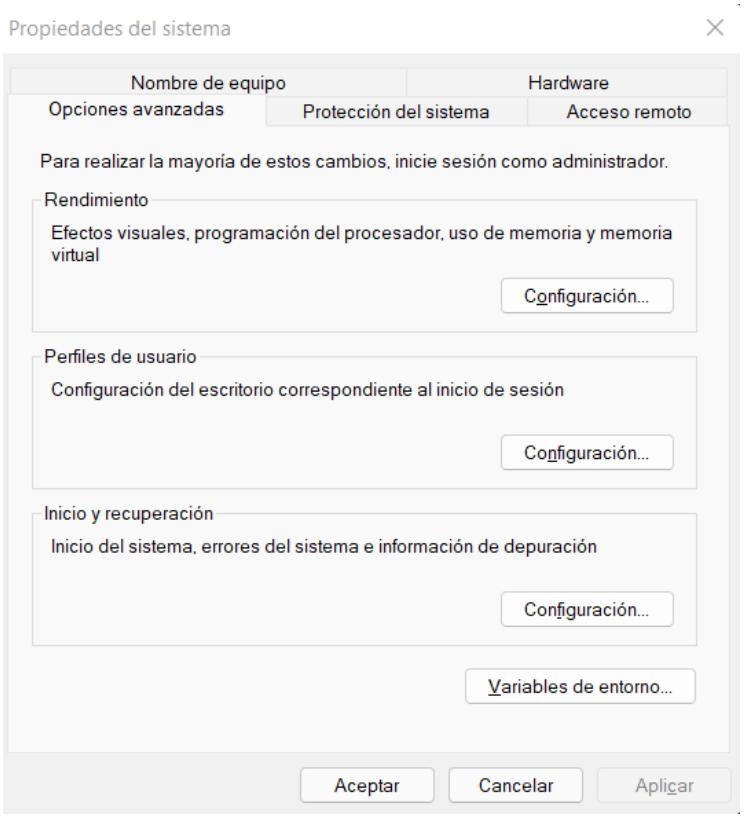



**Figura 119. Propiedades del sistema**

Una vez en esta ventana, se pulsará sobre "Variables de entorno" y aparecerá la siguiente interfaz:

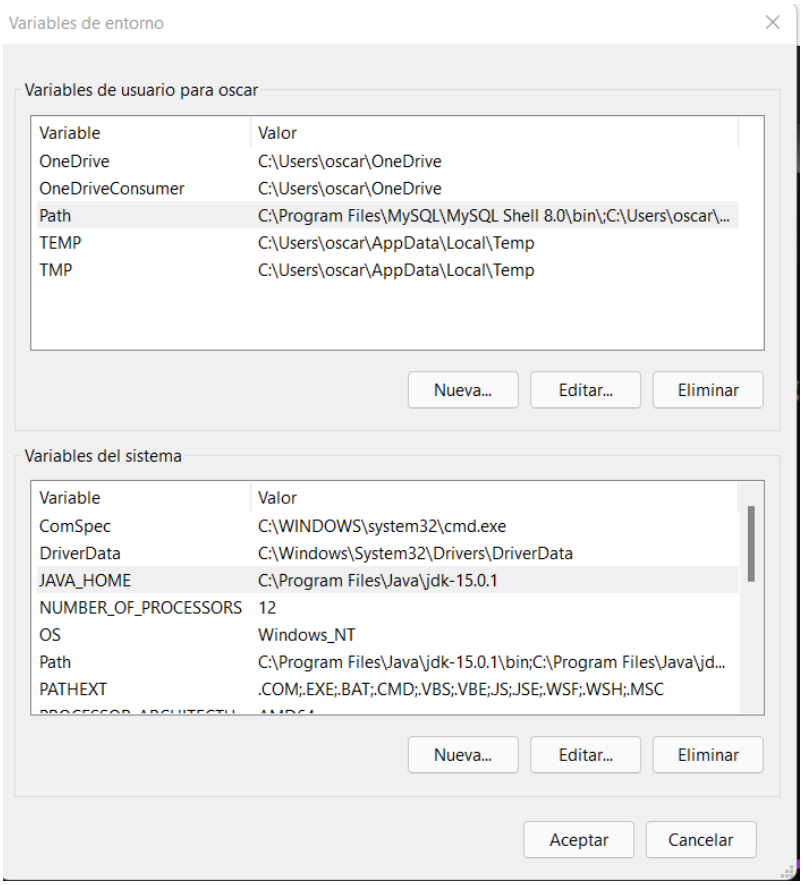



**Figura 120. Variables de entorno**

Se deberá crear una nueva variable del sistema llamada **JAVA_HOME** cuyo valor será la ubicación del JDK instalado anteriormente. Además, se debe modificar la variable del sistema Path e introducir la ubicación del directorio bin del JDK instalado anteriormente, como se puede observar en la siguiente imagen:

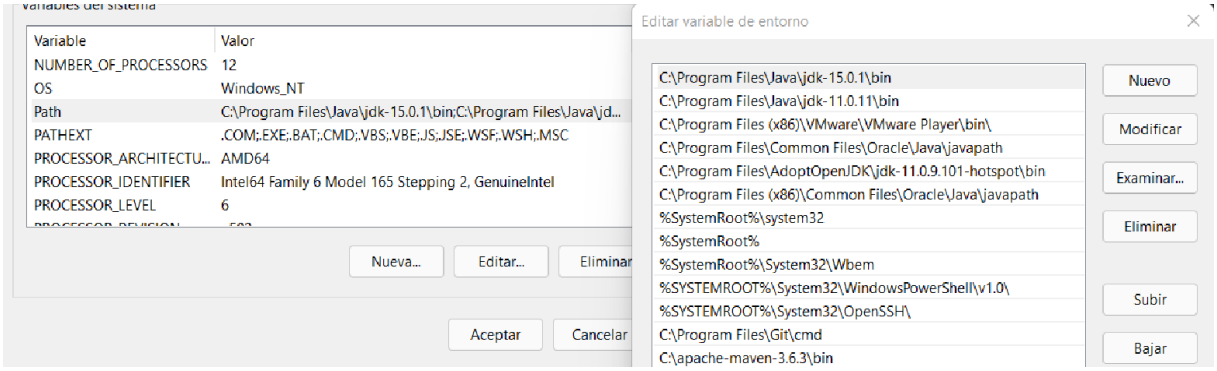



**Figura 121. Variable de entorno Path**

Una vez instalado el JDK, comprobaremos que todo ha ido correctamente utilizando el comando **javac -version** en la consola:

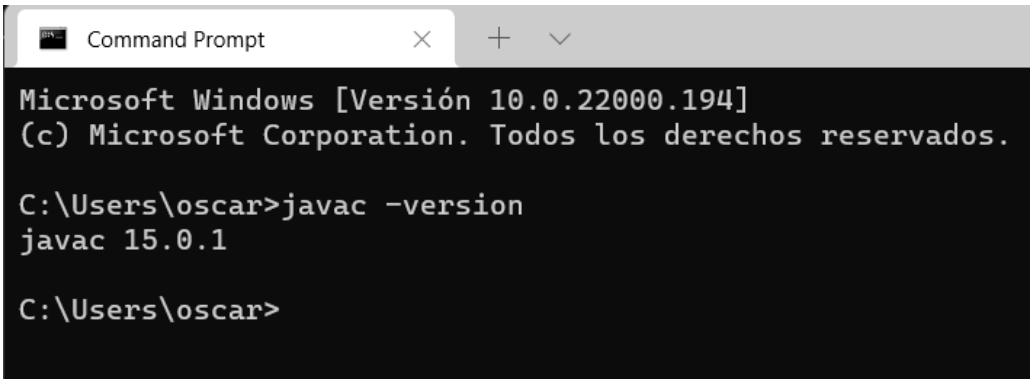



**Figura 122. Comando javac -version**

Ahora, debemos abrir los dos microservicios mediante un IDE, siendo lo más recomendable utilizar IntelliJ. Como estos microservicios están desarrollados con Spring Boot, no habría que configurar más que las propiedades del fichero .properties de cada uno. Las variables que se deben modificar dependiendo del entorno son las siguientes:

- Microservicio sales-api

- **spring.data.mongodb.uri** : URI de la base de datos siguiendo el formato de MongoDB, consultable desde https://docs.mongodb.com/manual/reference/connection-string/ .
- **server.port** : valor del puerto sobre el que se desplegará la aplicación.
- **authentication**.username : Credencial username para la seguridad BA.
- **authentication**.password: Credencial password para la seguridad BA.

- Microservicio sales-client

- **server.port** : valor del puerto sobre el que se desplegará la aplicación
- **api.endpoint** : dirección del microservicio sales-api. Si se despliega en local, un endpoint ejemplo sería http://localhost:8001
- **api.authentication.username** : Credencial username para la seguridad BA del microservicio *sales-api*, por lo que debe coincidir con la introducida en el microservicio anterior.
- **api.authentication.password** : Credencial password para la seguridad BA del microservicio *sales-api*, por lo que debe coincidir con la introducida en el microservicio anterior.

Para montar una base de datos MongoDB en local se recomienda utilizar la herramienta Robo 3T, y en el siguiente enlace se explica cómo desplegar una base de datos: https://parzibyte.me/blog/2018/10/29/instalar-robomongo-en-windows-10-mongodb/

Si por el contrario se desea montar la base de datos MongoDB para entornos en la nube, se recomienda consultar la documentación oficial a partir del siguiente enlace: https://docs.atlas.mongodb.com/getting-started/

Si se desea desplegar los microservicios en la nube, puede utilizarse Heroku. En su documentación hay una categoría específica para aplicaciones Spring Boot. Puede consultarse a través del siguiente enlace: https://devcenter.heroku.com/categories/working-with-spring-boot

# APÉNDICE B. GLOSARIO

**A**

**API**. **A**pplication **P**rogramming **I**nterface.

**B**

**BA**. **B**asic **A**uthentication es un proceso mediante el cual se implementa seguridad en el tráfico HTTP. Concretamente, se emplean credenciales de acceso: usuario y contraseña.

**F**

**Framework**. Entorno o marco de trabajo a partir del cual se desarrolla un proyecto.

**I**

**IP**. **I**nternet **Protocol**. Es el protocolo mediante el cual se identifican las redes y dispositivos en Internet.

**J**

**JDK. J**ava **D**evelopment **K**it.

**JSON**. **J**ava**S**cript **O**bject **N**otation. Formato de texto utilizado para el intercambio de estructuras de datos.

**JS. J**ava**S**cript.

**JWT**. **J**SON **W**eb **T**oken. Estándar de securización entre peticiones utilizando tokens en formato JSON.

**P**

**PAAS**. **P**latform **a**s **a S**ervice. Concepto utilizado en la tecnología cloud que hace referencia a las plataformas que se alojan en la nube ofreciendo servicios a un usuario.

**PyME**. **P**equeña **y M**ediana **e**mpresa.

**R**

**Responsive**. Significa adaptable y se utiliza en el contexto del diseño web. Un diseño adaptable significa que es accesible en todos los dispositivos: teléfonos móviles, tablets, portátiles, etcétera.