UNIVERSITAT DE BARCELONA

Treball de Fi de Grau

GRAU D'ENGINYERIA INFORMÀTICA

Facultat de Matemàtiques i Informàtica
Universitat de Barcelona

Deep Knowledge

Álvaro Guillén González

Director: Jordi José Bazán
Realitzat a: Departament de
Matemàtiques i Informàtica
Barcelona, 13 de juliol de 2013

Tabla de contenido

1. Resumen y Objetivos

i. Versión española:

El presente Trabajo de Fin de Grado (TFG) consiste en desarrollar y explicar cómo se debería crear una página web cambiando el enfoque universitario, en el que se trabaja como una demo, por un enfoque propio de un entorno real y empresarial.

Por tanto, el objetivo principal de dicha página web no se centrará en qué es lo que puede ofrecer, sino en cómo lo está ofreciendo a nivel interno. Esta diferencia y este cambio de mentalidad son fundamentales para comprender la verdadera magnitud y trabajo de este proyecto.

La página web se va a inspirar en el concepto de Udemy, que es una página web dirigida a ofrecer una serie de recursos gratuitos y de pago a modo de tutoriales de diversos tipos y tópicos. En nuestro caso, uno de los objetivos secundarios de nuestra página web será que ofrezca la posibilidad de gestionar una serie de cursos con unas publicaciones asociadas para que cualquier interesado pueda comprar dicho material y disfrutar de su contenido; de forma similar a como lo hace Udemy. Por tanto, cada usuario podrá crear y publicar sus cursos, al mismo tiempo que puede comprar los publicados por otras personas.

Además, otro objetivo secundario es implementar un chat. Para ello, la página web ofrecerá una interfaz de ayuda y consulta para cada uno de los cursos que será en formato chat. En otras palabras, para cada curso creado por el propio usuario, o que haya comprado; podrá visualizar y participar en el chat de ese curso aportando comentarios, preguntas o las respuestas que el usuario considere.

Para realizar la página web otro de nuestros objetivos será utilizar distintas tecnologías que permitan crear en *Java* un *Backend* basado en microservicios con *Spring Boot*, mientras que el *Frontend* será desarrollado con microfrontends utilizando *SingleSpa*.

Por tanto, un objetivo secundario será aprender y dominar los microservicios a la par que será necesario aprender y dominar *SingleSpa* para la creación de un *Frontend* estructurado en microfrontends.

Mas adelante, una vez finalizada la página web, mandarla a producción utilizando la plataforma de “Amazon Web Services” (*AWS*) será nuestro próximo objetivo secundario, pues, de esa forma, la página web será accesible por cualquier usuario desde internet.

Uno de los objetivos secundarios más importantes consiste en definir cómo se desarrollarán los microservicios y los microfrontends. Es por eso por lo que nos pondremos como objetivo implementar la *Arquitectura Hexagonal* en nuestros dos microservicios principales (“auth-api” y “res-api”). Mientras que el objetivo para nuestros dos microfrontends principales (“mf-auth-react” y “mf-resources-angular”) será contener un *rounting* interno de navegación por los distintos componentes y así crear una capa de complejidad para cada uno de esos dos microfrontends.

Para lograr tal propósito, como objetivo secundario estará la implementación de los patrones: *Spring Cloud Config*, *Discovery Server* y *Gateway*, además de tener presente y en todo momento el tema de la seguridad y escalabilidad de la propia página web.

Otro objetivo de este TFG es que pueda servir como “template” para aprender qué es lo que puede llegar a pedir cualquier empresa a la hora de contratarte como desarrollador web. Por ello, se implementarán nuevas y actuales metodologías que, en su mayoría, están siendo implementadas por el mundo empresarial. De esa forma, con el objetivo en mente de que este trabajo pueda servir como “template”, se priorizará mantenerlo lo más flexible, modulable y escalable posible.

Además, tendremos el objetivo de trabajar con los tres *frameworks* clásicos en el Frontend. Es por ello por lo que al definir nuestros microfrontends en *SingleSpa* utilizaremos *React*, *Angular* y *Vue*.

Finalmente, el último objetivo secundario será implementar una pasarela de pago, pues esta tecnología será la que usa cualquier empresa o página web real en la que haya alguna transacción de por medio.

ii. Versió catalana:

El present Treball de Fi de Grau (TFG) consisteix a desenvolupar i explicar com s'hauria de crear una pàgina web canviant l'enfocament universitari, en el qual es treballa com una demo, a un enfocament propi d'un entorn real i empresarial.

Per tant, l'objectiu principal d'aquesta pàgina web no es centrarà en què és el que pot oferir, sinó en com ho està oferint a nivell intern. Aquesta diferència i aquest canvi de mentalitat són fonamentals per a comprendre la veritable magnitud i treball d'aquest projecte.

La pàgina web s'inspirarà en el concepte d’Udemy, que és una pàgina web dirigida a oferir una sèrie de recursos gratuïts i de pagament en el format de tutorials de diversos tipus i tòpics. En el nostre cas, un dels objectius secundaris de la nostra pàgina web serà que ofereixi la possibilitat de gestionar una sèrie de cursos amb unes publicacions associades perquè qualsevol interessat pugui comprar aquest material i gaudir del seu contingut; de manera similar a com ho fa Udemy. Per tant, cada usuari podrà crear i publicar els seus cursos, al mateix temps que pot comprar els publicats per altres persones.

A més, un altre objectiu secundari és implementar un xat. Per a això, la pàgina web oferirà una interfície d'ajuda i consulta per a cadascun dels cursos que serà en format xat. En altres paraules, per a cada curs creat pel mateix usuari, o que hagi comprat; podrà visualitzar i participar en el xat d'aquest curs aportant comentaris, preguntes o les respostes que l'usuari consideri.

Per a realitzar la pàgina web un dels nostres objectius secundaris serà utilitzar diferents tecnologies que permetin crear a *Java* un *Backend* basat en microserveis amb *Spring Boot*, mentre que el *Frontend* serà desenvolupat amb microfrontends utilitzant *SingleSpa*.

Per tant, un objectiu secundari serà aprendre i dominar els microserveis al mateix temps que serà necessari aprendre i dominar *SingleSpa* per a la creació d'un *Frontend* estructurat en microfrontends.

Una vegada finalitzada la pàgina web, manar-la a producció fent servir la plataforma d'"Amazon Web Services" (*AWS*) serà el nostre pròxim objectiu secundari, doncs, d'aquesta forma, la pàgina web serà accessible per qualsevol usuari des d'internet.

Un altre dels objectius secundaris més importants consisteix a definir com es desenvoluparan els microserveis i els microfrontends. És per això que ens posarem com a objectiu implementar *l'Arquitectura Hexagonal* en els nostres dos microserveis principals ("auth-api" i "res-api"). Mentre que l'objectiu per als nostres dos microfrontends principals ("mf-auth-react" i "mf-resources-angular") serà contenir un rounting intern de navegació pels diferents components i així crear una capa de complexitat per a cadascun d'aquests dos microfrontends.

Per a aconseguir tal propòsit, com a objectiu secundari estarà la implementació dels patrons: *Spring Cloud Config*, *Discovery Server* i *Gateway*, a més de tenir present i en tot moment el tema de la seguretat i escalabilitat de la mateixa pàgina web.

Un altre objectius d'aquest TFG és que serveixi com a "template" per a aprendre què és el que pot arribar a demanar qualsevol empresa a l'hora de contractar-te com a desenvolupador web. Per això, s'implementaran noves i actuals metodologies que, majoritàriament, estan sent implementades pel món empresarial. D'aquesta forma, amb l'objectiu en ment que aquest treball pugui servir com "template", es prioritzarà mantenir-lo el més flexible, modulable i escalable possible.

Un dels nostres objectius serà treballar amb els tres *frameworks* clàssics en el *Frontend*. És per això que en definir nostres microfrontends en *SingleSpa* utilitzarem *React*, *Angular* i *Vue*.

Finalment, l'últim objectiu secondari serà implementar una passarel·la de pagament, perquè aquesta tecnologia serà la que usa qualsevol empresa o pàgina web real en la qual hi hagi alguna transacció pel mig.

iii. English version:

This Final Degree Project (TFG) consists of developing and explaining how a web page should be created, changing the university approach, in which it works as a demo, to an approach of a real and business environment.

Therefore, the main objective of our website will not focus on what it can offer, instead of that on how it is offering it internally. This difference and this change of mentality are fundamental to understand the real magnitude and work of this project.

The website is going to be inspired by the Udemy concept, which is a website aimed at offering a series of free and paid resources in the format of tutorials of different types and topics. In our case, one of the secondary objectives of our website will be to offer the possibility of managing a series of courses with associated publications so that anyone interested can purchase such material and enjoy its content; in a similar way as Udemy does. Therefore, each user will be able to create and publish their own courses, while at the same time being able to purchase those published by other people.

In addition, another secondary objective is to implement a chat. To this purpose, the website will offer a help and interactive interface for each of the courses that will be in a chat format. In other words, for each course created by the user, or the ones that he has purchased; he will be able to visualize and participate in the chat of that course providing comments, questions or the answers that the user considers.

To create the web page one of our secondary objectives will be to use different technologies that allow us to create in *Java* a *Backend* based on microservices with *Spring Boot*, while the *Frontend* will be developed with microfrontends using *SingleSpa*.

Therefore, a secondary objective will be to learn and to gain a great deal of knowledge and experience on microservices. Also it will be necessary to learn and master *SingleSpa* for the creation of a *Frontend* structured on microfrontends.

Later, once the website is finished, sending it to production using the "Amazon Web Services" (AWS) platform will be our next secondary objective, because, in that way, the website will be accessible by any user from the internet.

One of the most important secondary objectives is to define how the microservices and microfrontends will be developed. That is why we will aim to implement the Hexagonal Architecture in our two main microservices ("auth-api" and "res-api"). While the goal for our two main microfrontends ("mf-auth-react" and "mf-resources-angular") will be to contain an internal routing of navigation through the different components and thus create a layer of complexity for each of those two microfrontends.

To achieve this purpose, as a secondary objective will be the implementation of the patterns: *Spring Cloud Config*, *Discovery Server* and *Gateway*; in addition to keeping in mind, and at all times, the issue of security and scalability that have the website itself.

Another objective is that this TFG can serves as a "template" to learn what any company (that wants a website) can ask you (as a web developer). Therefore, we will implement new and current methodologies that, in most cases, are being implemented by the business world. Thus, with the goal in mind that this work can serve as a "template", we will prioritize keeping it as flexible, modular and scalable as possible.

One of our objectives will be to work with the three classic *frameworks* in the *Frontend*. That is why when defining our microfrontends in SingleSpa we will use *React*, *Angular* and *Vue*.

Finally, the last secondary objective will be to implement a payment gateway, since this technology will be the one used by any real company or web page in which there is a transaction involved.

2. Introducción

Tal como podemos apreciar, este trabajo final de grado es muy ambicioso dado que, para llevar a cabo y de forma satisfactoria cada uno de los objetivos principales y secundarios mencionados, debemos adquirir un conocimiento muy extenso y profundo en muchos y muy diversos ámbitos de la programación web.

Este TFG parte de los métodos y metodologías vistas hasta ahora en la universidad y se embarcará en el mundo real empresarial aplicando nuevos métodos, patrones y arquitecturas.

Si bien es cierto que para algunos conceptos este documento puede resultar algo técnico, para aliviar esa carga se seguirá un hilo conductor global por el que recorreremos todos los apartados como si realizásemos un viaje en tren. De esa forma, como si fuéramos

parando en las distintas localidades desde el inicio hasta nuestro objetivo final, iremos viendo cada uno de los distintos apartados.

3. Motivación y Planificación

El punto de partida y nuestra primera estación en este viaje la encontramos dos años atrás: cuando realicé la asignatura de “Software Distribuido”, así comenzó la planificación del TFG. En esta asignatura teníamos que crear una página web utilizando la metodología *Agile* (de *Scrum*) y *Kanban* en un grupo de siete personas. Se nos daba total libertad para poder crearla. Sin embargo, optamos por los conocimientos aprendidos en la propia universidad, por lo que desarrollamos una página web monolítica tanto a nivel de *Backend* usando *Python*, como monolítica a nivel de *Frontend* utilizando *Vue*.

El trabajo realizado en esa página web de dicha asignatura podría ya ser un TFG, ahora bien, fui consciente de algunos problemas sobre todo a nivel de escalabilidad y seguridad. Es por ello por lo que, desde entonces, emprendí una búsqueda ampliando el conocimiento para aprender cómo se estructuran o, mejor dicho, cómo se deberían estructurar las páginas webs en la actualidad.

A medida que iba aprendiendo las nuevas tecnologías que estaban siendo desarrolladas en la creación de páginas webs, llegué a la conclusión de que sería mejor posponer la asignatura del TFG al año siguiente. Además, decidí realizar prácticas de empresa en el primer semestre de ese año, para aprender de primera mano lo que es un entorno laboral empresarial y el nivel que se espera de una página web y, posteriormente, dedicarme en exclusiva a este trabajo de final de grado durante el segundo semestre.

En la “Feria de Empresas” que se celebró hace un año para facilitar la comunicación entre estudiantes y empresas (y así ayudarles a que entrasen en el entorno laboral), mi principal objetivo fue adquirir información sobre cómo estaban estructuradas sus páginas webs. La conclusión que obtuve de los más de 25 “stands” distintos que visité, fue que predomina por excelencia *Java Spring Boot Backend* (tanto a nivel monolítico como a nivel de microservicios). En cambio, en el *Frontend* había más diversidad, aunque en su mayoría se centraban en *Angular* y *React*. Aun así, todos a quienes pregunté, y sin ninguna excepción, comentaron que estructuraban el *Frontend* de forma monolítica.

A partir de ese momento, supe con toda certeza que mi trabajo final de grado trataría de una página web en *Java Spring Boot Backend* basada en microservicios, y un *Frontend* del que lo único que sabía, por aquel entonces, era que quería alejarme de *Vue* pues es lo que ya dominaba a partir de lo aprendido en la universidad. Por tanto, lo que tenía claro es que quería hacerlo en *Angular* o en *React*. Sería un año más tarde cuando me decantaría por hacer el *Frontend* en *Angular*, ya que es lo que aprendí haciendo mis prácticas de empresa.

Después de recorrer durante más de doce meses un largo sendero que nos llevó a lo que se convertiría en la segunda parada de esta gran aventura, había llegado el momento de empezar mi trabajo final de grado.

4. Foto inicial de conocimiento

Como le ocurre a todo escritor, lo más difícil es el momento de empezar a redactar en una hoja en blanco. Así me sentí a la hora de empezar mi TFG. Los conocimientos previos con los que contaba antes de su realización han sido los siguientes:

A nivel de *Backend* dominaba *Python* tras mi formación en la universidad. Así mismo, gracias a las prácticas de empresa, sabía defenderme con *Java Spring Boot Backend Microservicios*, pero no lo dominaba pues no realicé ninguna tarea a nivel de configuración, sino que todo fue más a nivel de desarrollo de características o "debug", lo que se traduce en trabajar con *Java*. Además, también realicé varias formaciones sobre un nuevo tipo de arquitectura limpia: la *Arquitectura Hexagonal*. En ese momento, utilizar esa arquitectura se me hizo un trabajo invisible pues, en realidad, ya estaba creada toda la jerarquía de ficheros y entidades.

A nivel de *Frontend* dominaba *Vue* por la universidad, y gracias a las prácticas de empresa aprendí a defenderme en *Angular Monolítico*. A pesar de no dominarlo, fui aprendiendo varias cosas como el tema de las traducciones automáticas (aunque a nivel de configuración, en aquel momento no sabía cómo se hacía, sólo sabía que funcionaba en ese el estado actual). Es por ello que, posteriormente, me interesé en aprender cómo implementar esta tecnología y utilizarla en sincronía con las otras presentes en el proyecto.

A nivel de despliegue de la aplicación web (*deploy*) únicamente conocía la herramienta *Heroku* pues es la que habíamos utilizado siempre en el entorno universitario. Desafortunadamente, *Heroku* dejó de ofrecer sus servicios de forma gratuita por lo que tuve que ampliar horizontes y buscar otras alternativas. La primera alterativa fue *Hostinger*, y hacer un despliegue web basado en contenedores de *Docker* utilizando *Kubernetes*. No obstante, tras todo un día de pruebas y errores y posteriormente hablar directamente con soporte, llegué a la conclusión de que mi plan de utilizar *Kubernetes* en *Hostinger* era inviable, pues *Hostinger* no soporta la utilización de dicha tecnología. Por tanto, la decisión se centraba en los servicios que ofrecía Google (*Google Cloud*), o en los servicios de Amazon Web Service (*AWS*). Finalmente opté por *AWS*, aunque en ese momento no tenía ningún tipo de conocimiento ni experiencia. Si bien es cierto que en prácticas de empresa utilizábamos *AWS* precisamente, todo ese tema para mí me pasó totalmente desapercibido por lo que partía desde cero.

Después de haber hecho las maletas y empaquetar todos los conocimientos que poseía en ese momento, llegamos a la tercera estación de este viaje, donde debíamos planificar la ruta que seguiría nuestro tren.

5. Foto esperada de conocimiento

En esta nueva localidad hice una proyección hacia el futuro, pensando qué conocimientos llegaría a asimilar y aprender al final de este gran viaje. Por consiguiente, y a modo de resumen: el objetivo del TFG que pensé en ese momento era el desarrollar una página web con *Java Spring Boot Microservicios* en el *Backend*, un *Frontend* monolítico en *Angular* y el *deploy* realizado en *AWS*.

A continuación, observaremos como este objetivo se ha cumplido a excepción de un pequeño, pero muy significativo, detalle: pasar de un *Frontend* monolítico, a uno basado en

microfrontends utilizando *SingleSpa*. Que es una tecnología de la que no había oído hablar en ese momento ni sobre la que tenía ningún tipo de conocimiento ni experiencia práctica.

Sin embargo, tuve que realizar una parada de emergencia, la que se convertiría en la cuarta etapa antes de empezar a adentrarnos en la aventura. Este nuevo parón era de vital importancia ya que primero debía hacer una recopilación del significado de algunos puntos vitales del mapa que recorreríamos en dicha aventura.

6. Glosario

Con la finalidad de esclarecer algunos de los conceptos clave que se mencionan a lo largo de esta memoria (pues algunos pueden ser algo técnicos), brindaremos el siguiente glosario para facilitar su lectura.

- **Arquitectura monolítica**: el código se estructura en forma de un gran bloque todo unido.
- **Arquitectura en microservicios**: el código monolítico en un gran bloque del *Backend* se disgrega y separa según sus funcionalidades en bloques más pequeños e independientes. Cada uno de estos bloques independientes se llama microservicio.
- **Arquitectura en microfrontends**: mismo concepto que los microservicios, pero aplicado al *Frontend*. El gran bloque monolítico del código del *Frontend* se disgrega y separa en bloques más pequeños e independientes. Cada uno de estos bloques independientes se llama microfrontend. Para la creación de microfrontends existen dos estrategias: *MF* y *SingleSpa*. En *SingleSpa* existen tres tipos (en realidad son cuatro, pero en este proyecto únicamente mencionaremos tres pues serán los utilizados) de microfrontend: *Root Config*, *Parcela* y módulo de Utilidad.
- ***Backend***: la parte del Servidor en la página web que la forman todos los microservicios. Nuestro *Backend* está formado por los micorservicios: “config-service”, “eureka-service”, “gateway-service”, “auth-api”, “braintree-api” y “res-api”. Además, en nuestro *Backend* estará presente una carpeta aislada “config-data” que no es un microservicio, sino que representa un contenedor (un repositorio de *Github*) aislado de todo el resto de los ficheros y microservicios.
- ***Frontend***: la parte con la que interactuará el Cliente en la página web que la forman todos los microfrontends. Nuestro *Frontend* está formado por los microfrontends: “micro-frontend-app” (microfrontend de tipo *Root Config*), “mf-navbar-angular”, “mf-footer-vue”, “mf-auth-react”, “mf-braintree-angular”, “mf-resources-angular” y “mf-utility” (este último es nuestro microfrontend de tipo módulo de utilidad). Todos los otros microfrontends a excepción del “micro-frontend-app” y “mf-utility” serán los microfrontends de tipo *Parcela*.
- **Monorepo**: organización y estructuración interna del *Backend* o del *Frontend* en la que el repositorio es un único repositorio, por lo que todos los microservicios o microfrontends estarán bajo un único repositorio Github. La estructuración del repositorio no afecta a si los microservicios o microfrontends son independientes. Tan sólo es una forma de agrupar todas esas carpetas.
- **Multirepo**: de forma similar a una estructura monorepo, en este caso cada uno de los microservicios del *Backend* o microfrontends del *Frontend* tiene su propio repositorio Github.
- ***SPA***: *SPA* o “Single Page Application” significa que en el momento de que el *Frontend* deba visualizar nuevos datos, no recargará la página, sino que en la misma página se visualizarán los nuevos datos.

- ***MPA***: *MPA* o “Multi Page Application” significa que en el momento de que el *Frontend* deba visualizar nuevos datos, recargará la página, y por tanto mostrará una página nueva y diferente a la anterior que será la que contenga los nuevos datos.
- ***MF***: siglas correspondientes a “Webpack Module Federation” que es una estrategia para la creación de microfrontends. Esta estrategia se caracteriza por exportar e importar módulos, funciones… que actuarán como microfrontends. Su implementación es totalmente compatible con *SingleSpa*.
- ***SingleSpa***: es la estrategia adoptada para la creación de microfrontends. En donde cada uno de nuestros microfrontends podrán ser de los tipos: *Root Config*, *Parcela* o módulo de Utilidad.
- **Equivalencias microservicios y microfrontends**: los microservicios “config-service”, “eureka-service” y “gateway-service” del *Backend* tendrán el mismo propósito que el microfrontend “micro-frontend-app” (microfrontend de tipo *Root Config*). Los microservicios “auth-api”, “braintree-api” y “res-api” se corresponderán con los microfrontends “mf-auth-react”, “mf-braintree-angular”, “mf-resources-angular” (microfrontends de tipo *Parcelas*) respectivamente.
- ***LocalStorage****:* es una propiedad del navegador que permiten almacenar información por parte del Cliente qué será accesible por cualquier elemento del *Frontend* durante nuestra sesión.

Como es lógico, estos conceptos serán desarrollados a lo largo de este documento con mayor profundidad. Con ello, pretendo como objetivo esclarecer al máximo y proporcionar un mapa conceptual que facilite la comprensión de aquello a lo que nos estamos refiriendo, para que no haya ningún atisbo de duda.

Después de avanzar en nuestro viaje, nos detuvimos para visitar los distintos modelos arquitectónicos dado que allí el camino se bifurcaría y tendríamos que escoger entre uno y otro para lograr nuestro objetivo.

7. Arquitectura Monolítica vs Microservicios

Una página web es un mundo muy extenso y se puede analizar desde diversos puntos de vista y en diversos niveles y profundidades. A continuación, muestro un diagrama en el que quedan plasmados los dos modelos arquitectónicos más característicos.

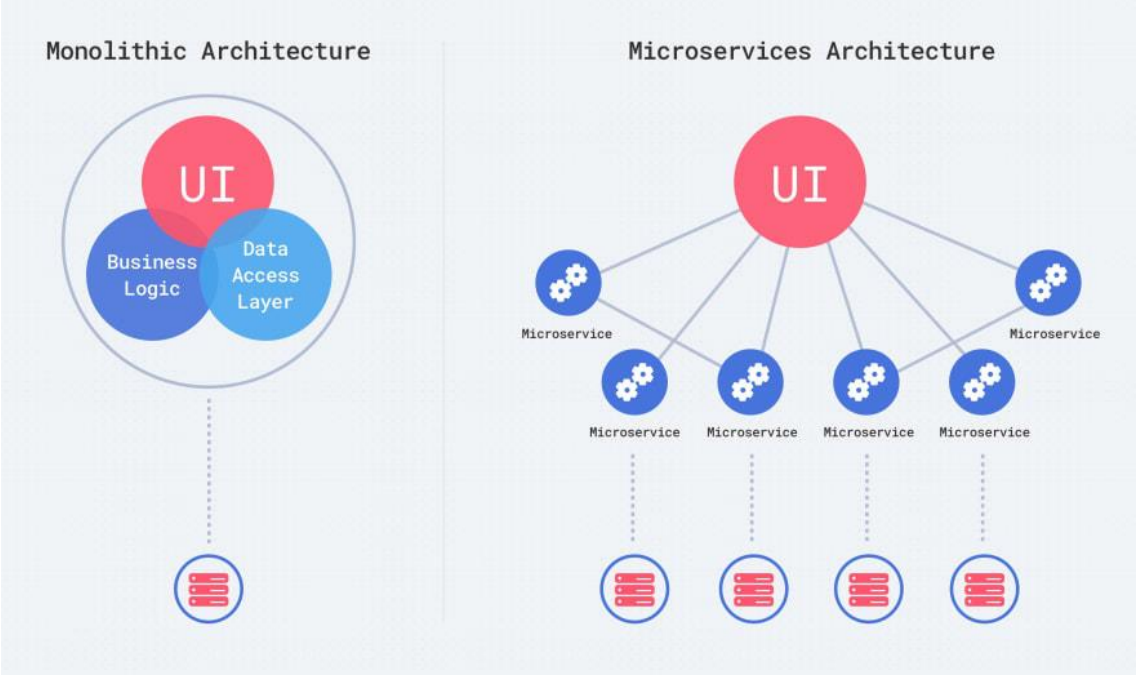

Ilustración 1: Representación de la arquitectura monolítica (izquierda) y la arquitectura basada en microservicios (derecha).

El modelo tradicional para la creación de una página web se basa en la arquitectura monolítica. En este tipo de arquitectura nuestro programa es un bloque grande y único que está unificado y que es autónomo e independiente. En este gran bloque están presentes todas las partes del código, por lo que ante cualquier cambio es necesario volver a compilar todo el bloque.

La ventaja principal de la arquitectura monolítica es la velocidad de desarrollo pues al ser sólo un bloque, el trabajo interno de *DevOps* (configurar las comunicaciones entre cada uno de los componentes que conforman la web) es menor. De todos modos, cabe destacar también la existencia de una serie de ventajas como, por ejemplo:

- Tiene una implementación más sencilla pues es un único archivo o directorio ejecutable.
- Su desarrollo es más rápido al estar todo unificado.
- El rendimiento es centralizado.
- Sus pruebas son más simplificadas al ser una única unidad y centralizada.
- La depuración es más sencilla al estar todo ubicado en un mismo lugar.

A pesar de las ventajas, no debemos olvidarnos de lado las desventajas. Pues que analizándolas descubrimos que, precisamente se centran en los aspectos y objetivos de este TFG:

- La escalabilidad representa un desafío cuando crecen demasiado.
- La fiabilidad es un reto pues en caso de haber un fallo en cualquier parte de los módulos del código, afectaría a la disponibilidad de toda la aplicación provocando que, en caso de fallar una característica, todas las otras dejen de funcionar.
- Existe una barrera en la adopción de nuevas tecnologías ya que cualquier cambio en el marco o el lenguaje o la propia versión, afectará a la totalidad de la aplicación.
- Carece de flexibilidad pues al incorporar una nueva tecnología resulta un auténtico desafío para que no colisione o sea incompatible con otras ya añadidas.
- En la implementación, a cada pequeño cambio es necesaria una nueva implementación de todo el conjunto del bloque.

Por otra parte, se encuentra el nuevo modelo: la arquitectura de microservicios, que se basa en una serie de servicios que pueden implementarse de forma independiente los unos de los otros. Estos servicios tienen su propia lógica y base de datos con un objetivo específico. Por tanto, tanto el código, como las pruebas, la implementación y el escalado, son independientes y específicos de cada uno de los microservicios.

La funcionalidad principal de los microservicios es desacoplar cada una de las principales características de la base monolítica. Esto no reduce la complejidad, pero sí ayuda a que sea mucho más visible y fácil de gestionar ya que se separan en procesos menores e independientes. La adopción de los microservicios va de la mano con el trabajo de *DevOps*, pues es el encargado de comunicar y gestionar cada uno de estos núcleos independientes.

La ventaja principal de la arquitectura basada en microservicios es que resuelven la mayoría de los programas a nivel de empresas y del crecimiento del software. También, están presentes otra serie de ventajas que conviene destacar como, por ejemplo:

- Son mucho más ágiles dado que promueven la forma de trabajar con equipos pequeños haciendo una metodología de trabajo ágil. Favoreciendo el uso de la metodología de trabajo *Agile*, utilizada por la inmensa mayoría de equipos de desarrollo.
- Tienen un escalado flexible debido a que, cuando un microservicio esté alcanzando su capacidad máxima de carga, puede implementarse rápidamente una nueva instancia de dicho microservicio para aliviar la carga de tráfico, permitiendo respaldar tamaños de instancias mucho mayores.
- Trabajan en un sistema de implementación continúa. Como resultado, debido a su tamaño, es posible hacer ciclos de lanzamientos más frecuentes y rápidos.
- Son muy sencillos de mantener y probar debido a que son más pequeños. Lo que permite poder tirar atrás con facilidad, en caso de un error. Además, su implementación es continua y rápida permitiendo incorporar nuevas actualizaciones. Esto se debe a que es muy fácil de aislar y corregir fallos y errores, pues son servicios individuales.
- Permite una alta flexibilidad tecnológica porque cada microservicio contará con las herramientas y versiones que requiera.
- Son de alta fiabilidad pues puedes implementar cambios en un microservicio concreto sin la necesidad o el riesgo de modificar o alterar ningún otro.
- A nivel de equipo de trabajo es más satisfactorio pues es más sencillo de entender. Además, únicamente debes dominar esa parte concreta y no te es necesario comprender ni entender la totalidad del código.

De forma equivalente, a pesar de las ventajas, no debemos dejar de lado las desventajas. Pues estas, se centran precisamente en la gestión interna a nivel de DevOps pues incrementa la carga de trabajo en ese ámbito:

- Consta de un incremento en la carga de desarrollo debido a que deben configurarse diversas aplicaciones en lugar de una sola.
- Puede haber una mayor carga organizativa en el caso de que distintos equipos de distintos microservicios deban ponerse en contacto y colaborar para coordinar una funcionalidad en conjunto.
- Hay un mayor desafío en la depuración en el caso que se requiera la intercomunicación de varios microservicios.
- Puede haber una falta de estandarización en algunos ámbitos como, por ejemplo, en el caso de los idiomas, por lo que se requiere de un registro y una supervisión.

Con todo ello, podemos observar cómo la arquitectura basada en microservicios se adecúa más a los requisitos de los objetivos de este trabajo final de grado.

Recordemos que este TFG pretende responder a la pregunta de: ¿Qué pasaría si una empresa real me pidiera crear una página web? Debido a que buscamos no sólo facilitar una metodología *Agile*, sino que además sea flexible, mantenible, sostenible, fiable, escalable y que puedan añadirse nuevas tecnologías fácilmente a medida que se vaya requiriendo.

Una vez hechas las maletas con el conocimiento actual, pensada la ruta que queríamos realizar en este viaje y aclarados los lugares que iríamos visitando, ahora sí que empezaba nuestra aventura. Partiríamos hacia un nuevo país, debíamos cruzar la frontera de lo conocido y adentrarnos en mundo del *Backend*.

8. Implementación del Backend

a. Resumen Global

Cualquier página web se caracteriza por tener dos partes bien diferencias. El lado del servidor y el lado del cliente. O, en otras palabras, mientras que el cliente o usuario interactúa con una interfaz, la información es procesada en el servidor por otros servicios totalmente distintos. A estos dos mundos se les denomina *Frontend* y *Backend*. Mientras que el *Frontend* es la parte visible para el cliente con la que podrá interactuar, el *Backend* es el lado del servidor que queda desapercibido en la mayoría de los proyectos, pues esas "Apis" no dejan de ser unas "cajas negras" de *inputs* y *outputs*.

Precisamente, en el presente trabajo, el *Backend* tiene una gran importancia por lo que a lo largo de esta memoria vamos a ir entrando en cada una de esas "cajas negras" para explicar no sólo su comportamiento, sino también la necesidad de usarlos al solucionar una serie de problemas que pueden tener las páginas web.

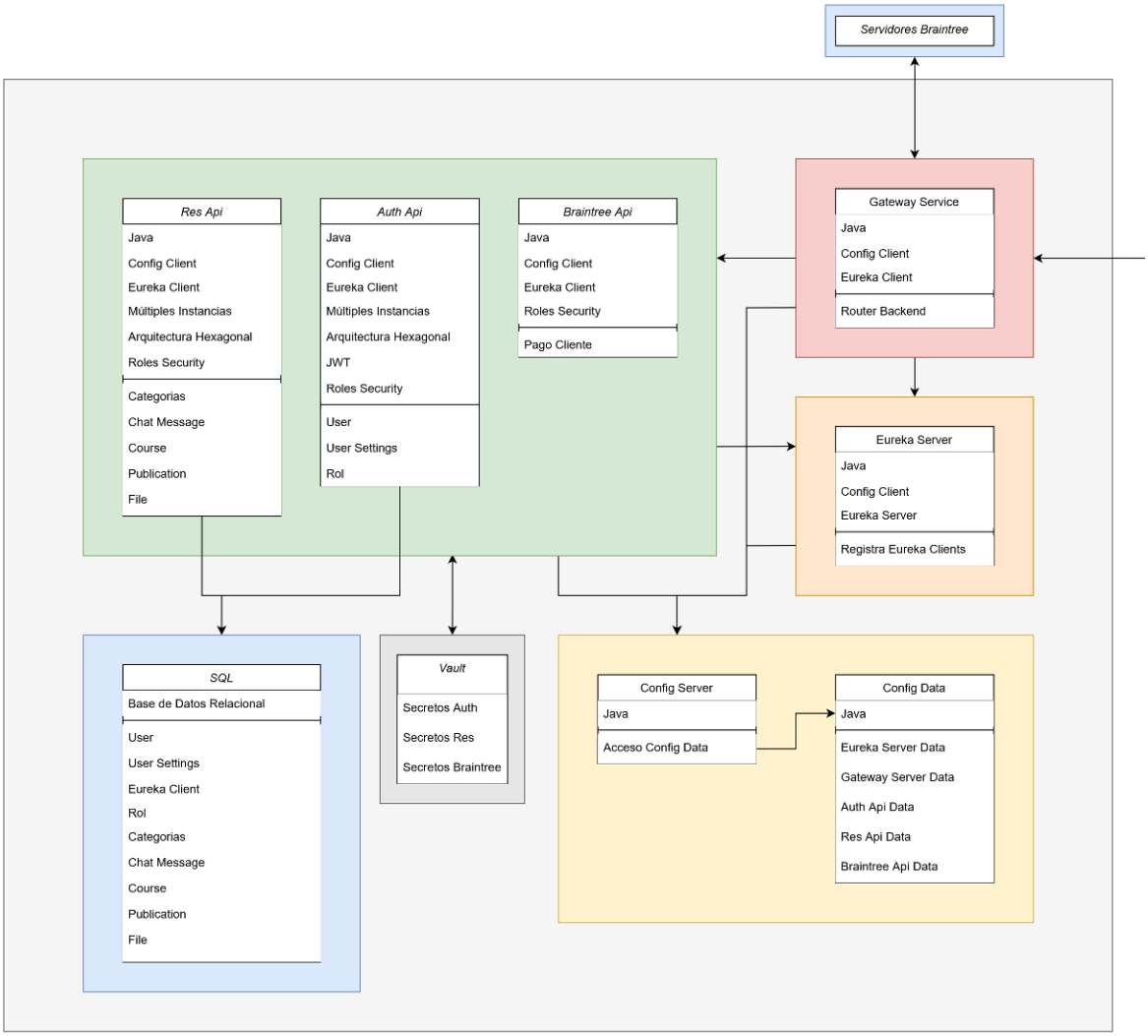

Ilustración 2: Representación final de la estructura del Backend. Esta figura es la sección derecha de la ilustración27 presente en el Anexo1.

Tal como se observa en la ilustración anterior este es el diagrama final del *Backend*. A continuación, explicaremos en detalle cada uno de los módulos y su funcionamiento. No obstante, hay que mencionar que existen dos tipos de flechas, las que van por **módulo** (cada uno de los **recuadros blancos**) y las que van por **contenedores** (cada uno de los **cuadrados de colores** que simboliza que la flecha afecta a todos los módulos dentro de ese contenedor).

A modo de resumen, los colores significan lo siguiente: el contenedor rojo representa al *Gateway* que actuará como *router* o *firewall*, pues será el único punto de entrada y salida para el *Frontend*. El de color naranja simbolizará el registro y descubrimiento de microservicios, mientras que el contenedor amarillo significará el tema relacionado con la configuración cuyo apartado es el que explicaremos a continuación. De igual manera, en el de color verde están los tres microservicios que se ocuparán de la gestión y el manejo de los datos por lado del servidor. Además, el contenedor gris hace referencia al *Vault* que incluirá los secretos de la configuración necesaria no incluida en el contenedor amarillo para los módulos del contenedor verde. Finalmente, los contenedores azules significan una conexión con algunos recursos, mientras que abajo encontramos el módulo de la base de datos, arriba podemos observar una conexión con los servidores propios del *Braintree*.

Tal como mencionábamos, a continuación, vamos a introducir el contenedor amarillo encargado de proporcionar la configuración al resto de componentes.

Nuestra séptima parada y la primera en este nuevo mundo que es el *Backend*, consistirá en aprender la mecánica actual para poder configurar los distintos microservicios.

b. Spring Cloud Config

Cualquiera que desarrolle un *Backend* basado en microservicios se dará cuenta que para cada uno de los microservicios deberá configurar su archivo de propiedades correspondiente.

Esta tarea se puede tornar algo tediosa y lenta si la hacemos manualmente para cada uno de los distintos microservicios. Sin embargo, podemos utilizar *Spring Cloud Config* que nos facilita esa tarea de configuración.

Para ser exactos, usando *Config Server* tenemos un lugar central para administrar las propiedades externas de las aplicaciones en todos sus entornos. Es decir, en lugar de tener que acudir a cada uno de los microservicios, si queremos modificar una propiedad de configuración, todas estas propiedades estarían almacenadas de forma centralizada en un lugar externo a los microservicios. De esta forma, obtendremos que todos los archivos se encuentran en un repositorio y el *Config Server* acudirá a dicho repositorio para proporcionarle la configuración necesaria a cada uno de los *Config Clients* (nuestros microservicios).

Los sistemas serán capaces, en el proceso de arranque, de buscar sus configuraciones en el servicio remoto y trabajar con ellas. No obstante, podemos apreciar que existe un problema en caso de actualizar el fichero de propiedades, pues habrá que notificar a los clientes para que refresquen ese cambio o, en caso contrario, será necesario levantar de nuevo cada microservicio (que es precisamente lo que estamos evitando al implementar este patrón de configuración). Es por ello por lo que, en nuestro caso particular de *Spring Cloud Config*, este problema de inyectar las configuraciones en caliente se resuelve utilizando la combinación de *Spring Cloud Bus* con un sistema de cola de mensajes

MQ como podría ser *Rabbit* o *Active*. De esta forma, estaríamos enviando una notificación a los clientes para que ellos mismos hagan un auto refrescamiento de las propiedades al detectarse un cambio en dicho fichero.

i. Implementación del Patrón (*Server*)

Para implementar este patrón de configuración centralizada por el lado del Servidor deberemos crear un nuevo microservicio y añadir la etiqueta de "@EnableConfigServer" en su archivo main además de añadir las dependencias correspondientes al *pom* presentes en el Anexo2.

Por último, y aquí es donde reside lo más importante, habrá que definir tres aspectos en su archivo de configuración. La primera será el puerto donde se está ejecutando, la segunda será el nombre la de aplicación. Y la tercera, y más importante, será la localización del *git* donde estarán ubicados los archivos de configuración para cada uno de los microservicios.

ii. Implementación del Patrón (*Client*)

En el lado del Cliente la configuración es muy similar. En la clase principal no se deberá añadir ninguna etiqueta, sino que será en el archivo de configuración del propio microservicio donde deberemos añadir las propiedades del *Config*. Concretamente, la siguiente ilustración corresponde al microservicio de "auth-api" que se encarga de toda la lógica de los usuarios (más adelante en este propio documento lo explicaremos). Sin embargo, de todas las propiedades tendremos que añadir las que corresponden a la etiqueta: "spring.cloud.config". En la que podemos ver que al añadir "enable: true" estamos indicando que es un cliente y por tanto deberá buscar el Servidor que está ubicado mediante la etiqueta: "uri".

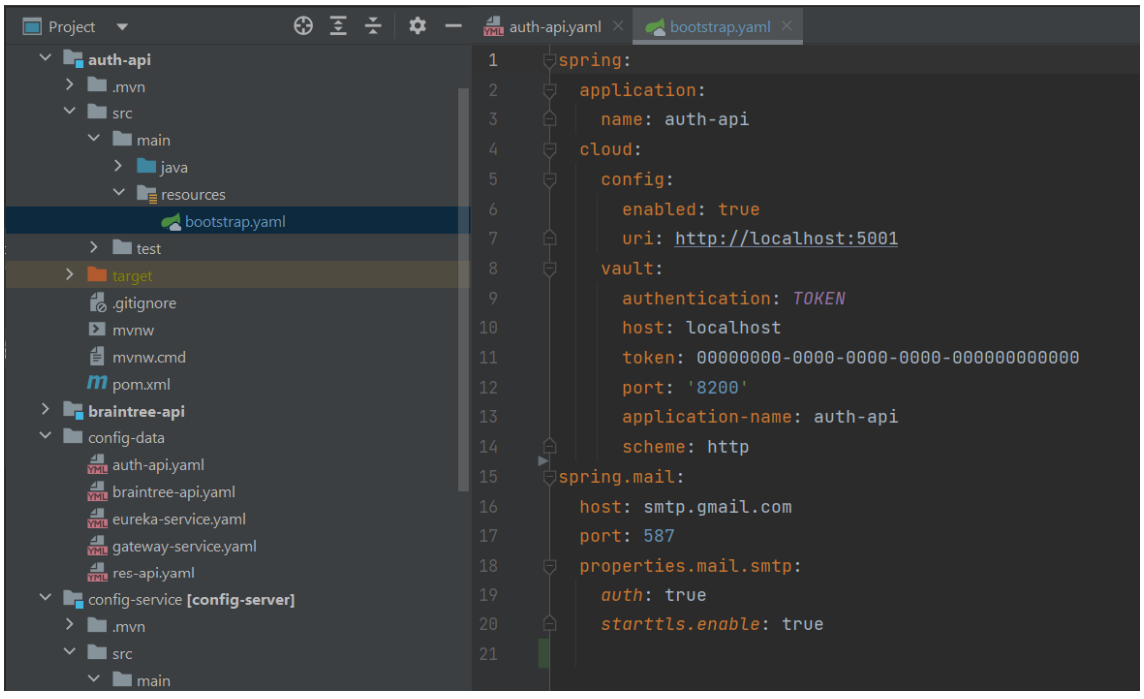

Ilustración 3: Representación del archivo de propiedades "bootstrap.yaml" del microservicio "auth-api".

Para acabar, el fichero “aplication.properties” lo renombraremos como: “bootstrap.yaml” además de que la configuración del fichero *pom* para que actúe como un *Config Client* podemos encontrarla en el Anexo3.

iii. Monorepo vs multirepo y repositorio privado vs público

Llegados a este punto es fundamental mencionar dos detalles. El primero es la diferencia entre una estructura monorepo en lugar de la multirepo.

La estructura monorepo se caracteriza porque todo está bajo una carpeta principal que actúa como contenedor principal y es la que tiene el “.git”; en otras palabras, todo va a un único repositorio. Mientras que la estructura multirepo, se caracteriza porque cada microservicio tiene su propio repositorio. Que sea monorepo o multirepo no significa que la aplicación sea monolítica o esté basada en microservicios, tan sólo significa cómo se organizan los distintos módulos bajo un solo repositorio o en varios. En este proyecto se ha adoptado una estructura monorepo, debido a que de esta forma es más fácil poder entregar el trabajo del TFG pues todo está en un único lugar (repositorio), en lugar de tener que entregar varios archivos cada uno de un repositorio distinto en caso de haber implementado una estructura multirepo.

Es por este motivo por el que el repositorio aparte que mencionábamos que era el encargado de almacenar todos los ficheros de configuración para cada uno de los microservicios, aunque en la realidad lo ideal es que fuera un repositorio completamente aislado y distinto, a fines prácticos para este proyecto no deja de ser una carpeta (concretamente la carpeta “config-data”) dentro de esta organización monorepo pero fuera e independiente a cualquier otro módulo (micoservicio).

El otro detalle que es necesario destacar es la diferencia entre un repositorio Privado y uno Público.

Esta diferencia, aunque evidente, es fundamental a niveles prácticos. En un repositorio público, únicamente con tener su url (la propiedad “uri”) ya sería suficiente pues cualquiera puede acceder a esa dirección. Sin embargo, en un repositorio privado no se puede acceder únicamente con la url, pues necesitaremos unas credenciales para poder visualizar el repositorio. Este pequeño detalle es fundamental cuando se está implementando este patrón de configuración centralizada, pues deberá poder acceder a esa url para buscar el contenedor (denominado por el campo “search-paths”) que albergue los distintos ficheros de configuración.

En caso de buscar información o documentación sobre cómo implementar un *Spring Cloud Config Server* con un repositorio privado (pues actualmente mi repositorio de *Github* del TFG es privado), lamento informar que todos y cada uno de los tutoriales son para repositorios públicos. Sin embargo, no está todo perdido, pues la solución que se me ocurrió fue la de generar un token de únicamente lectura y añadir las credenciales de dicho token en la propia configuración con las etiquetas: “username” y “password”. El “username” es mi propio nombre de usuario de *Github*, mientras que el “password” es el propio token.

Tras varios ciclos de prueba y error (pues no hay información sobre este tema) en la configuración del token, puedo garantizar que funciona perfectamente, aplicando la siguiente configuración (aunque no descarto que sea posible que alguna característica activada, realmente no sea necesaria a niveles prácticos):

Ilustración 4: Representación esquemática de la configuración del token utilizado para proporcionar las credenciales al microservicio "config-service". La configuración expandida de este token podemos encontrarla en el Anexo4.

De todas formas, cabe la posibilidad de tener que modificar el repositorio y hacerlo público para la corrección de este propio trabajo. Sin embargo, como este problema lo he visto en multitud de escenarios y comentarios, aunque a nivel teórico algunos sí han aportado alguna respuesta, a niveles prácticos nadie ha mostrado cómo hacerlo concretamente. A continuación, muestro la configuración de este microservicio "config-service" aportando la solución a conectar con los repositorios privados.

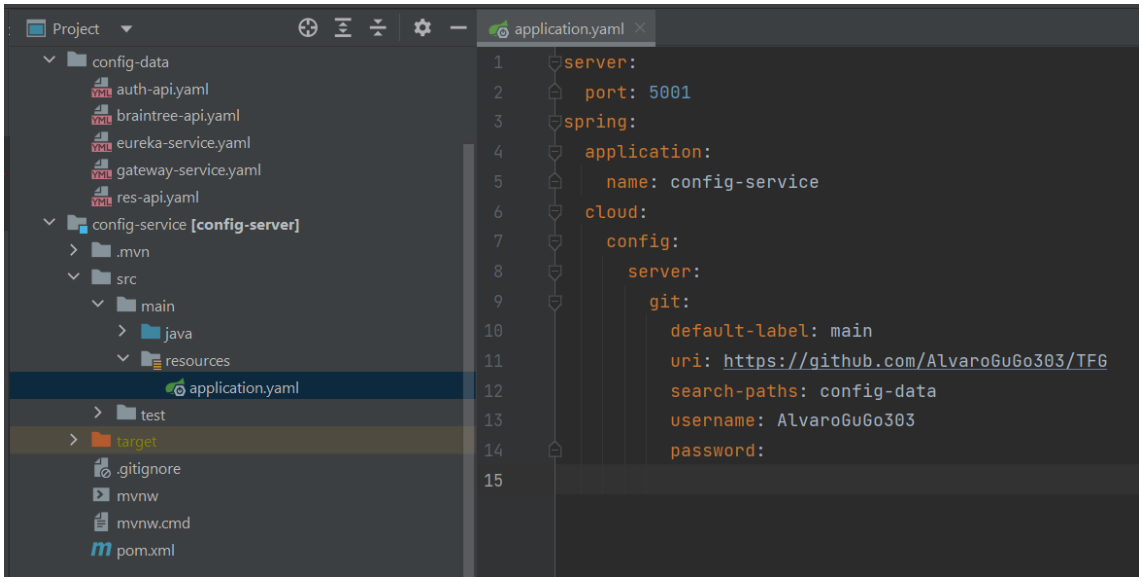

Ilustración 5: Representación del fichero de propiedades "application.yaml" del microservicio "config-server" en donde puede obervarse las credenciales para la conexión a la ubicación "config-data" en el repositorio de Github.

Por último, la etiqueta "default-label" corresponde a la rama del repositorio a la que acudirá a buscar los archivos de configuración.

iv. Config Data

Por tanto, los archivos de configuración que necesitará cada uno de los "Config Clients" indicado por la etiqueta "search-paths" podrán ser, por ejemplo, del siguiente estilo:

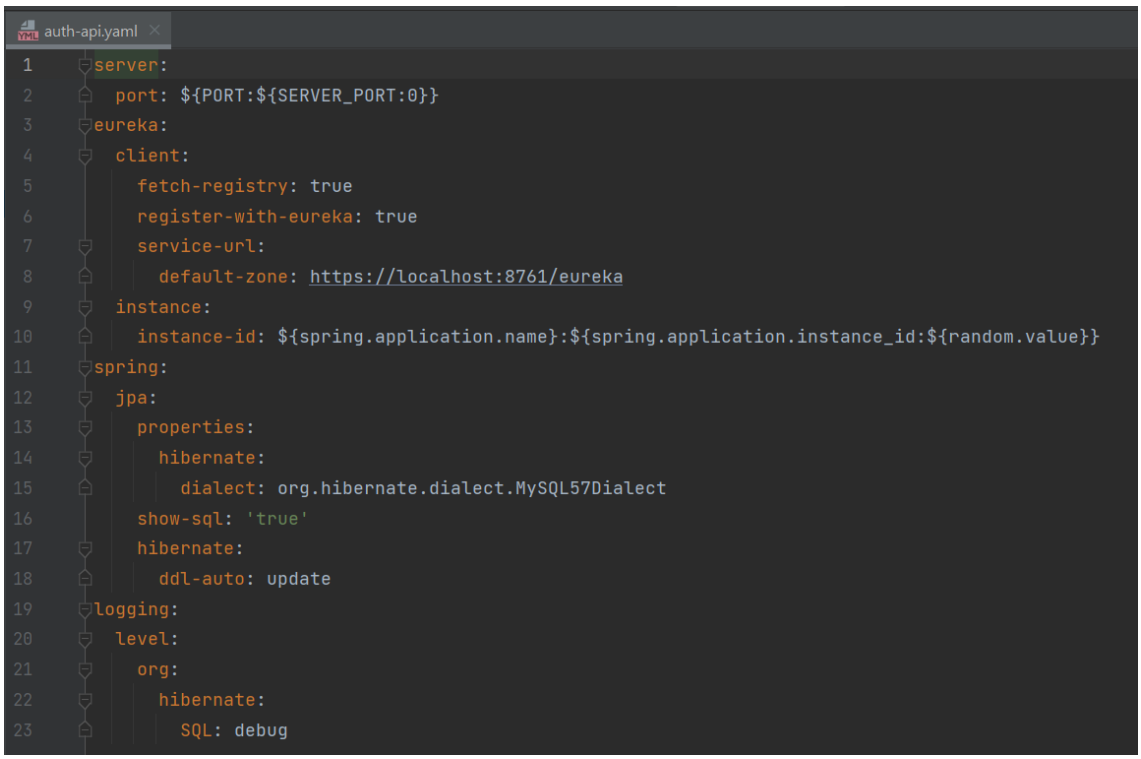

Ilustración 6: Representación del fichero de propiedades "auth-api.yaml" ubicado dentro de la carpeta "config-data". Este fichero contiene las propiedades que necesitará el microservicio "auth-api".

Ahora no nos detendremos en explicar cada uno de los puntos de este fichero de configuración, sino que lo haremos progresivamente a lo largo de este documento a medida que vayamos explicando cada una de las tecnologías y patrones utilizados.

Dejando atrás el microservicio de la configuración, ahora debemos dirigirnos a nuestra siguiente estación: nuestra novena parada. A partir de este momento, nos adentraremos en el territorio de los secretos.

c. Integración de Vault para los secretos

Llegados a este punto y, tal como se puede observar, añadir las credenciales en los archivos de configuración es un problema de seguridad y grabe; pues cualquiera con acceso al fichero podrá obtener las credenciales y hacer lo que quiera con ellas (es por este motivo por el que en la ilustración anterior he borrado el propio token).

Recordemos que uno de los propósitos fundamentales de este TFG, es poder ser utilizado por una empresa real. Por tanto, si tenemos en cuenta este enfoque, si una empresa añade las credenciales en la propia configuración se expone a que cualquier desarrollador o cualquiera que logre vulnerar el sistema y llegue a obtener dicho fichero; pueda, no sólo atacar a la base de datos (pues es la típica configuración en la que se debe añadir la localización de la base de datos, el usuario y la contraseña), sino también venderlas o proporcionarlas a un tercero.

De todas formas, este problema tiene una solución. Podemos utilizar la herramienta de control de información *Vault* para almacenar estas propiedades de la configuración que sean "críticas" como precisamente las credenciales.

i. Implementación del Patrón

Para implementar este patrón de diseño de modelado de datos, únicamente debemos añadir en la propia configuración que acceda a *Vault*, tal como se puede ver a continuación:

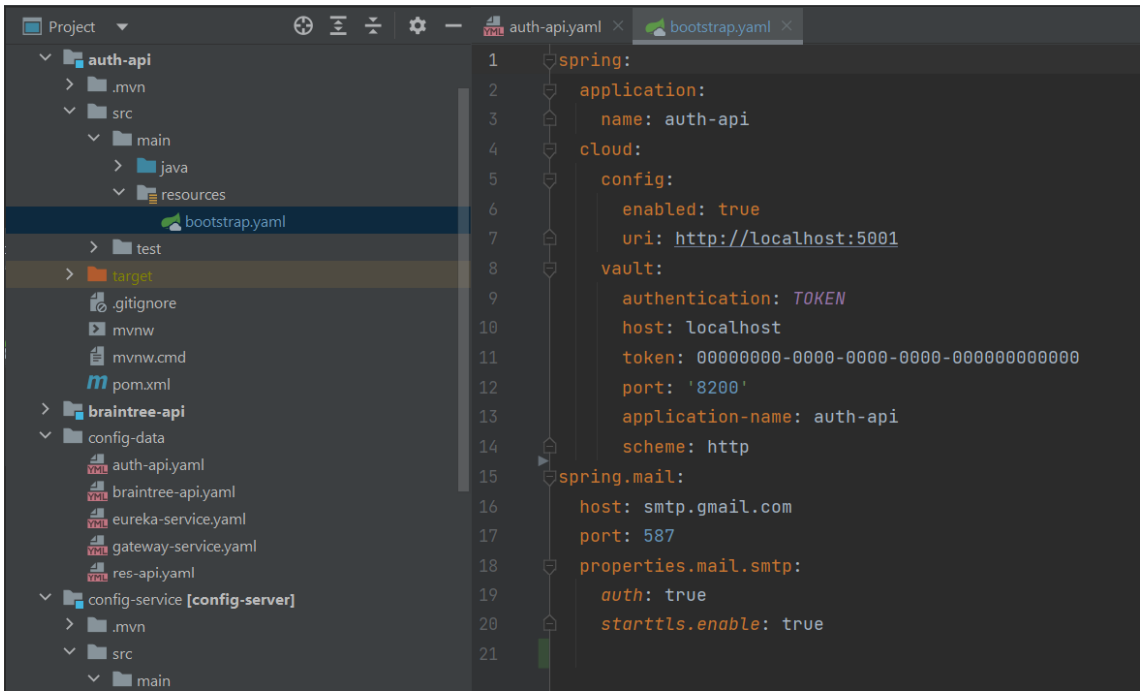

Ilustración 7: Representación del fichero de propiedades "bootstrap.yaml" del microservicio "auth-api". Esta figura es equivalente a la ilustración3. En el Anexo5 se proporciona el significado de cada campo en la configuración del Vault.

A fin de conseguir que acuda a *Vault* para recoger los secretos que le corresponden a dicho microservicio, deberemos añadir la siguiente etiqueta: "spring.cloud.vault" con sus respectivos campos.

Con toda esta información le estamos indicando al microservicio que tiene configuración en *Vault*, pero en ningún momento hemos mencionado qué secretos hemos añadido a *Vault*. Estos secretos los veremos más adelante cuándo expliquemos cada uno de los microservicios. Sin embargo, es necesario mencionar la existencia del fichero "vaultJson.txt" ubicado dentro de la carpeta principal del proyecto TFG.

En este fichero se encuentra toda la configuración de *Vault* o, lo que es en otras palabras, los secretos completamente expuestos. Este fichero evidentemente no debería existir. No obstante, lo he dejado a propósito a modo de "template" o guía por si alguien quisiera ejecutarlo de forma local. Esto se debe a que los microservicios que utilizan una base de datos, por defecto van a pedir la url de dicha base de datos bajo la etiqueta "spring.datasource.url" y, en caso de no proporcionarla, el microservicio no podrá ejecutarse y dará error.

Por tanto, si por ejemplo nuestro microservicio (en este caso estamos mostrando el *Vault* correspondiente al microservicio "res-api") utilizase una base de datos *SQL*, deberemos añadir a *Vault* lo siguiente:

```
res-api
{
  "spring.datasource.password": "root",
  "spring.datasource.url": "jdbc:mysql://localhost:3306/TFG",
  "spring.datasource.username": "root"
}
```

Ilustración 8: Representación del formato JSON para añadir secretos a Vault. Código extraído del fichero "vaultJson.txt" ubicado dentro de la carpeta principal del proyecto. Acuda a Anexo6 para obtener más información sobre como añadir secretos a Vault.

La siguiente meta en nuestro trayecto será el descubrimiento y registro de los microservicios. Esta parada es muy necesaria porque debemos saber con qué recursos contamos en todo momento.

d. Discovery, Eureka Server

Como toda aplicación web de una empresa que tenga un tráfico medio o alto de conexiones, la saturación de los nodos y el tráfico de red es un problema en caso de sobresaturarse. Para ello, una de las soluciones más adoptadas es la de usar un servicio *Discovery*.

El servicio *Discovery* no deja de ser un componente que se encarga de recuperar el registro de servicios de todas las instancias de los servicios disponibles. Y, de esta forma, realizar un balance de carga en caso de sobresaturarse el microservicio (recurso) de peticiones. Este descubrimiento de servicios se puede dar tanto del lado del cliente como del lado del servidor.

Existen distintos servicios *Discovery*, el utilizado en este trabajo será el *Eureka*.

i. Implementación del Patrón (Server)

Para implementar este patrón de descubrimiento de servicios por el lado del Servidor, tendremos que añadir la etiqueta: "@EnableEurekaServer" y añadir en archivo *pom* las dependencias necesarias que muestra el Anexo7.

Además, en su archivo de configuración tan sólo hemos de especificar el nombre de la aplicación y que es un *Config Client*, tal como se muestra a continuación:

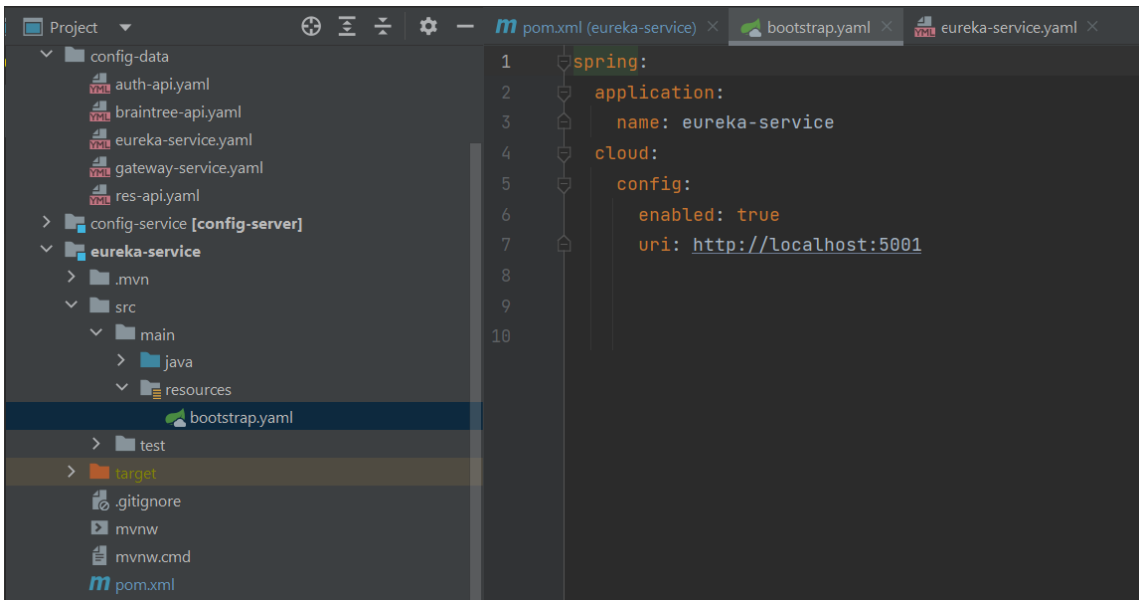

Ilustración 9: Representación del fichero de propiedades "bootstrap.yaml" del microservicio "eureka-service".

Con todo ello, en la configuración principal (que la obtendremos a través del *Config Server*) será donde configuremos realmente lo que necesitamos para implementar este patrón de descubrimiento de servicios por parte del Servidor.

Con este fin, deberemos fijar el puerto del microservicio que actuará como *Eureka Server*. Adicionalmente no queremos que se registre a sí mismo como un *Eureka Client* por lo que deberemos dejarlo a false. Pero sí será necesario, por parte del *Eureka Client*, especificar dónde se encuentra ("eureka.client.service-url.default-zone") y el nombre de la instancia ("eureka.instance.hostname").

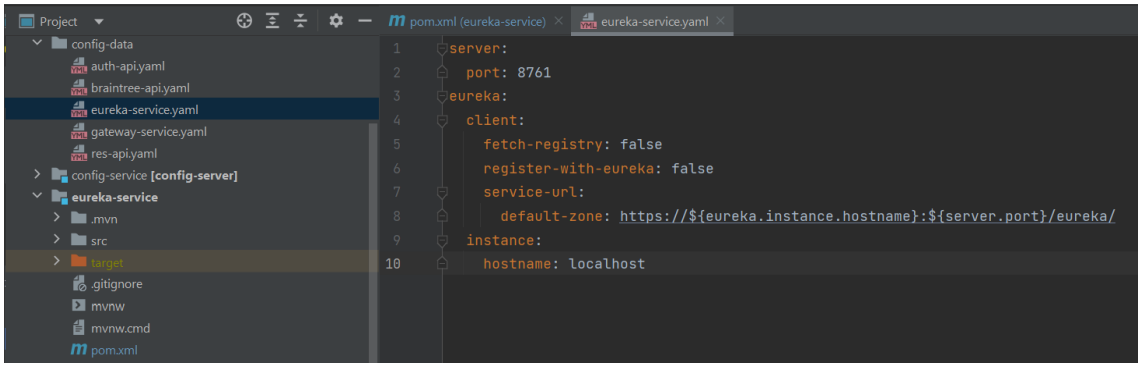

Ilustración 10: Representación del fichero "eureka-service.yaml" ubicado dentro de la carpeta que actúa como repositorio externo "config-data".

Como podemos observar en la ilustración, esta sería la configuración final del *Eureka Server* una vez especificados todos sus campos.

El puerto escogido "server.port: 8761" es el clásico que se le asigna por defecto a *Eureka Server*. Por ello lo he fijado a dicho puerto, pero a niveles prácticos se podría especificar cualquier otro.

ii. Implementación del Patrón (Client)

En el lado del Cliente hay algunas variaciones respecto a la del Servidor. La primera y gran diferencia la podemos encontrar en la etiqueta que deberemos añadir a la clase principal, pues en lugar de ser “@EnableEurekaClient” usaremos “@EnableDiscoveryClient”.

Es preferible utilizar la genérica debido a que, de esta forma, consigues abstraer a este microservicio de tipo *Discovery Server* que estamos usando. Recordemos que existen varios tipos de implementación del patrón de descubrimiento de servicios, por lo que, en caso de poder tener más de un tipo distinto de *Discovery Client*, al dejarlo de forma genérica únicamente nos tendremos que preocupar de añadir las distintas dependencias en el *pom* y el único fichero que se deberá modificar para añadir un tipo u otro de *Discovery Client* será el fichero de configuración que está apartado en el repositorio (en nuestro caso, una carpeta) “config-data” gracias al *Config Server*. Lo que permite no tener que parar la ejecución ni alterarla en caso de cambiar de tipo de *Discovery Client*.

En nuestro caso, únicamente está previsto utilizar el tipo *Eureka*, pero ya lo dejamos configurado en caso de ser necesario a posteriori. Con todo ello, también deberemos añadir en el *pom* la misma dependencia que para el *Eureka Server* que puede visualizarse en el Anexo8.

En el fichero de configuración propio del microservicio no deberemos añadir nada respecto al *Discovery Eureka Client*, por lo que tendremos un fichero como, por ejemplo:

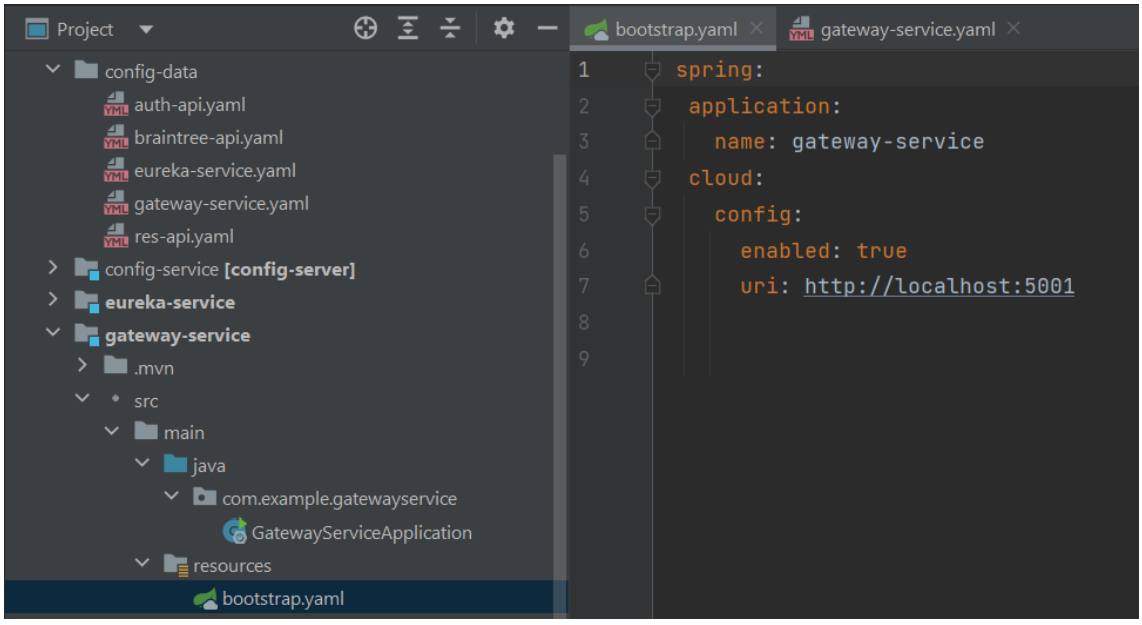

Ilustración 11: Representación del fichero “bootstrap.yaml” del microservicio “gateway-service” que ilustra que no es necesario ningún tipo de configuración extra para el Discovery Eureka Client en ese fichero de propiedades.

En el fichero podemos observar cómo únicamente está configurado el propio nombre del microservicio, y dónde está el *Config Server*, pues este microservicio de *Gateway* (cuyo nombre es “gateway-service”) es un microservicio que actuará como *Config Client*, además de ser un *Discovery Eureka Client*.

Donde realmente reside la diferencia entre si el microservicio actúa como Servidor o como Cliente *Eureka*, es en su archivo de configuración externo, en la carpeta “config-data”:

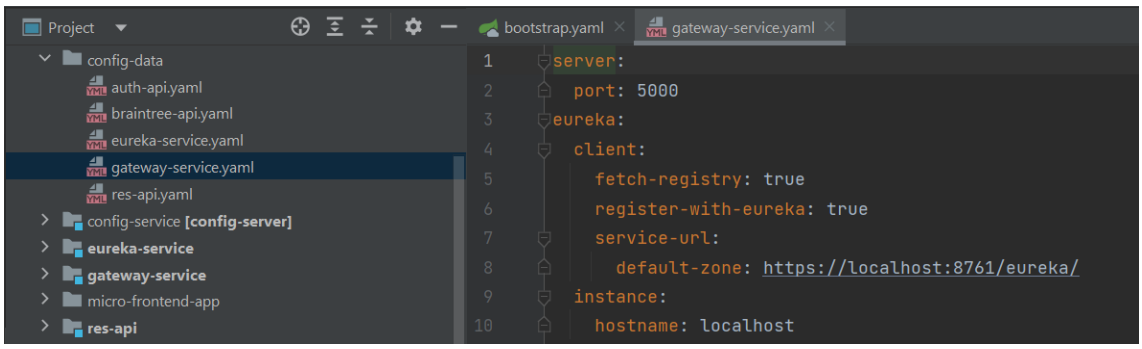

Ilustración 12: Representación del fichero "gateway-service.yaml" ubicado en la carpeta "config-data" que muestra la configuración para el Discovery Eureka Client.

Dejando de un lado el tema del puerto y otro tipo de configuraciones pertenecientes a este microservicio concreto ("gateway-service"), nos hemos de fijar en dos detalles fundamentales:

- Las variables booleanas del Cliente tendrán que estar activas (recordemos que en el caso del servidor estaban a desactivadas, es decir, con su valor a "false")
- Especificar (tal como hacíamos en el caso de los *Config Clients* que añadíamos donde estaba ubicado el Config *Server*) dónde se encuentra el *Discovery Eureka Server* mediante la variable: "eureka.client.service-url.default-zone".

iii. Implementación de múltiples instancias

Llegados a este punto, habíamos presentado un problema que sufre cualquier web, que es el balanceo de carga cuando hay demasiado tráfico y se saturan los microservicios. Es por ello por lo que hemos introducido este nuevo patrón de descubrimiento de servicios. Aun así, por el momento no hemos visto cómo solucionar dicho problema.

La solución por tanto es poder crear múltiples instancias (duplicados) de los microservicios que nosotros deseemos y (para que funcione) necesitamos dos cosas:

- Poder mantener un registro de estas nuevas instancias. De esto se ocupará precisamente el patrón *Discovery*.
- Poder dirigirnos a cada una de estas instancias sin depender de su puerto (pues cada duplicado se caracterizará por tener un puerto distinto); pero todos ellos deben poder ser accesibles desde un único punto que será nuestro *Gateway*, el cual será introducido al finalizar este apartado.

Para poder crear múltiples instancias y de esa forma solucionar el problema del balance en la carga de peticiones web, debemos realizar tres pasos básicos:

- El primero consiste en, una vez seleccionado el microservicio que queremos que pueda tener múltiples instancias (pues no es necesario que todos lo permitan), deberemos añadir un "Bean" de configuración (marcado con la etiqueta "@Configuration") en el que añadiremos la etiqueta "@LoadBalanced") tal como se muestra en la siguiente ilustración:

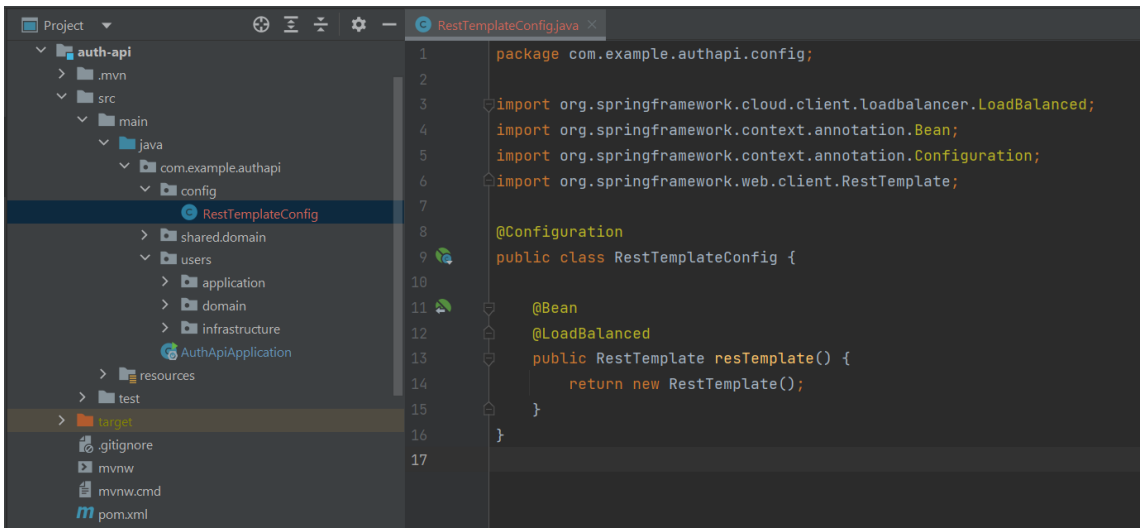

Ilustración 13: Representación del Bean de configuración "RestTemplate" del fichero "RestTemplateConfig.java" en el microservicio "auth-api" para permitir que el microservicio pueda tener múltiples instancias.

- El segundo paso será modificar las propiedades: "server.port" y "eureka.instance" de los archivos de configuración externos ubicados en el "config-data". Concretamente deberemos dejarlos de la siguiente forma:

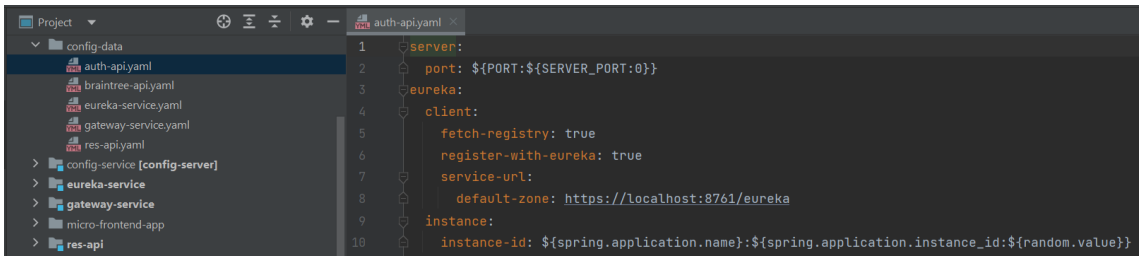

Ilustración 14: Representación del archivo de configuración "auth-api.yaml" ubicado en la carpeta "config-data" en donde se visualizan las propiedades para permitir las múltiples instancias.

- El tercer y último paso, será propio del entorno del *IDLE* de programación. En nuestro caso se ha utilizado la aplicación "IntelliJ IDEA" en su versión premium (con licencia). Para poder activar esta característica, deberemos ir a la configuración y activar la opción de "Allow multiple instances".

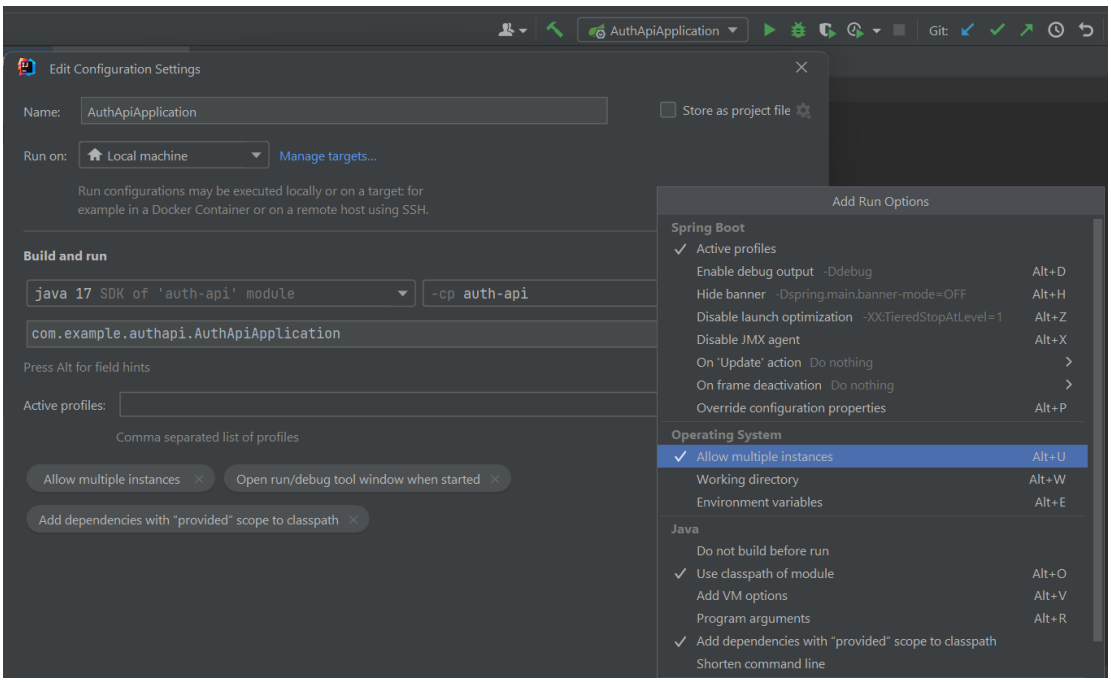

Ilustración 15: Representación de la configuración en el IDLE "IntelliJ IDEA" para habilitar que un microservicio pueda tener múltiples instancias.

Realizaremos estos tres pasos para cada uno de los microservicios en los que queramos permitir las instancias múltiples.

Realizando esto, habremos solucionado el problema original que teníamos de la saturación de peticiones web a un microservicio provocando una demora en el número de solicitudes procesadas al ser mayor que el número de peticiones que puede procesar.

Actualmente nos vemos en la obligación de parar en la estación del *Gateway*, pues será absolutamente necesario si queremos proseguir con nuestro viaje.

e. Gateway

Como hemos visto anteriormente, debemos ser capaces de poder acceder a cada uno de nuestros microservicios, y a cada una de las instancias que tengamos desde un único punto en común. Es por ello por lo que es necesario implementar un patrón *API Gateway*. Aplicaremos este patrón de despliegue de aplicaciones (microservicios en nuestro caso), dado que desde el *Frontend* no sería posible intentar acceder directamente a una instancia de un microservicio que tenga balanceo de carga, pues el puerto se asigna dinámicamente y de forma aleatoria; por lo que observamos que no tenemos otra alternativa que implementar dicho patrón de *Gateway*.

De esta forma, todas las solicitudes del *Frontend* se dirigirán al *Gateway* y será éste quien vaya redireccionando a cada uno de los distintos microservicios e instancias.

El objetivo del patrón *API Gateway* es la de brindar una solución al problema anterior, centralizando y enrutando todas las peticiones del exterior hacia cada uno de los microservicios. Es decir, actuaría como un *proxy* o punto único de entrada que después enruta la petición hacia cada "endpoint" distinto.

Además, al utilizar este patrón estamos brindando de una serie de mejoras transversales a cada uno de los microservicios; como, por ejemplo:

- Seguridad en los datos pues al actuar como proxy también puede actuar como firewall haciendo que peticiones no deseadas no puedan llegar a los microservicios.
- Nos ayuda también en el monitoreo y las métricas para observar precisamente que microservicio es más demandado y por tanto poder aplicar un balanceo de carga sobre éste.
- Además de muchos otros como la resiliencia pues se pueden definir de entrada múltiples enrutamientos.

i. Configuración del CORS

Existe un gran problema inicial al implementar un *Gateway*: se trata del *CORS*.

CORS (en su denominación inglesa: “Cross-Origin Resource Sharing”) es un mecanismo de seguridad que aplican los navegadores cuando estamos haciendo una petición a un recurso que está alojado en otro origen. Por tanto, protege a las *APIs* de las llamadas de orígenes cruzados, pues el navegador automáticamente comprobará las cabeceras *HTTP* buscando la autorización expresa por parte del servidor.

La configuración del *CORS* realmente es bastante sencilla pues hay una gran cantidad de documentación sobre ello. No obstante, cuando se implementa un *Frontend* basado en microfrontends con *SingleSpa* y un *Backend* basado en microservicios con los patrones anteriormente vistos; a simple vista puedes no estar seguro de donde debes añadir la configuración necesaria del *CORS* para solucionar dicho error. Esto se debe principalmente a que el error no te indica en qué parte del microservicio o del microfrontend exacto se produce. Por este motivo puede ser una labor muy difícil encontrar la solución si se desconoce su origen.

Dentro de cada uno de los microservicios de este TFG (“auth-api”, “braintree-api” y “res-api”) hay implementado una validación del *CORS*. Sin embargo, el origen del problema venía porque la petición se estaba rechazando desde el propio *Gateway*. Para solucionarlo, hay que añadir el siguiente código:

```
17      default-filters:
18        - DedupeResponseHeader=Access-Control-Allow-Credentials Access-Control-Allow-Origin
19      globalcors:
20        corsConfigurations:
21          '[/**]':
22            allowedOrigins: "*"
23            allowedMethods: "*"
24            allowedHeaders: "*"
```

Ilustración 16: Representación de la configuración necesaria del CORS que hay que añadir al Gateway en el fichero “gateway-service.yaml” en la carpeta “config-data”.

Dar con esa solución me llevó cinco días enteros de búsqueda, documentación, prueba y error. Esto se debe a que, en ese momento, no era consciente de qué era y dónde se estaba produciendo el fallo de *CORS* (si la parte del *Backend* o la del *Frontend* era quien originaba este error).

Como se puede apreciar en la ilustración anterior, realmente a nivel de seguridad se está dejando pasar cualquier tipo de petición. En otras palabras, sería posible aplicar un filtro de seguridad allí mismo. Pero como internamente, para cada microservicio ya se está gestionando ese filtro del *CORS*, dejé abierta la configuración en el *Gateway*.

ii. Implementación del Patrón

Para implementar este patrón *Gateway* únicamente debemos modificar dos ficheros. En este caso no será necesario modificar ni añadir ninguna etiqueta en la clase principal del microservicio "gateway-service". No obstante, hay que tener presente que el *Gateway* también será un *Discovery Eureka Client* y en consecuencia necesitará la etiqueta "@EnableDiscoveryClient" y la configuración pertinente del *Eureka Client* en su archivo externo de configuración ubicado en el repositorio (carpeta) "config-data".

La dependencia necesaria para el Gateway en el fichero pom podemos encontrarla en el Anexo9. El otro fichero que es necesario modificar es precisamente el de la configuración proporcionada por el *Config Client*; es decir, el fichero "gateway-service.yaml" ubicado en la carpeta "config-data".

En este fichero podemos observar tres bloques. El primero corresponde a las propiedades para que actúe como un *Eureka Client*. El segundo bloque corresponde a la configuración del *CORS* que mencionamos en el apartado anterior más en detalle. Y, por último, el tercer bloque corresponde a implementación de este patrón *Gateway* al definir el enrutamiento de cada una de las distintas rutas.

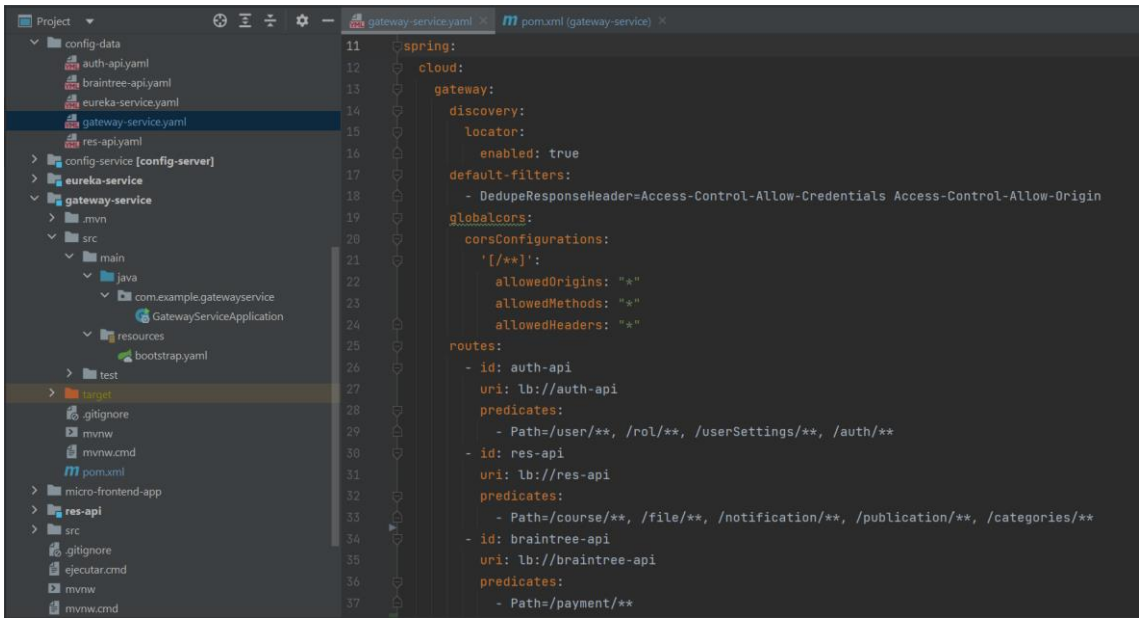

Ilustración 17: Representación del fichero "gateway-service-yaml" de configuración externa ubicado en la carpeta "config-data" para el microservicio "gateway-service".

Como podemos observar lo que nos interesa está a continuación de la variable "spring.cloud.gateway.routes" pues allí deberemos especificar el enrutamiento para cada uno de nuestros microservicios y sus distintos "endpoints".

Tras salir de los dominios del *Gateway*, que cobró cinco días atravesar, ahora (en lo que es nuestra doceava etapa) nos adentraremos dentro de los tres microservicios básicos.

Siendo la próxima estación muy importante pues aprenderemos muchos conceptos nuevos al profundizar en el microservicio “auth-api”.

f. Auth Api

Dejando al margen todos los otros microservicios en el *Backend* para aplicar distintas estrategias y patrones; en este TFG existen tres microservicios *API-REST* (“auth-api”, “braintree-api” y “res-api”) para la comunicación con nuestro *Frontend.*

El primero de ellos y el que veremos a continuación será el “auth-api”.

i. Funcionalidad y Objetivo

Como su propio nombre indica, este microservicio se encarga de todo lo relacionado con los usuarios. Es decir, no sólo se encarga de albergar y gestionar la información de los usuarios en la base de datos sino también la interacción que pueda tener este respectiva al propio usuario.

Por tanto, su objetivo será encargarse de: la gestión del inicio de sesión, la creación de la cuenta, la gestión de los múltiples roles y la autentificación. Siendo la autentificación por la validación de un correo electrónico, y el manejo del JWT (“JSON Web Token”). Ambos los detallaremos en profundidad más adelante.

ii. Patrones, Características y tecnologías utilizadas

Con el fin de no repetir lo anteriormente mencionado, en este microservicio se ha implementado:

- La integración con múltiples instancias (“Load Balanced”).
- El uso de *Vault*.
- Una validación extra para el *CORS*, añadiendo la etiqueta “@CrossOrigin” en cada uno de los controladores del microservicio.

Además de eso, también se han implementado varios conceptos, patrones, características y tecnologías nuevas que explicaremos a continuación.

iii. DDD + Patrón Arquitectura Hexagonal

Para el desarrollo de software existe una gran variedad de arquitecturas de software, es decir, ese conjunto de patrones que proporcionan un marco de referencia necesario para guiar la construcción de ese código y que todo el conjunto de desarrolladores debe compartir para poder aplicarla. Este conjunto de patrones establece el diseño de la arquitectura de un sistema debido a que establecen no sólo la estructura sino también el propio funcionamiento e interacción entre las partes del software.

Existen varios ejemplos, entre ellos el más conocido sería el modelo *MVC* (Modelo-Vista-Controlador). Sin embargo, debido a que la complejidad de los sistemas de software va en aumento, cada vez se va haciendo imprescindible el uso de “arquitecturas limpias” que nos permitan separar las distintas responsabilidades mediante capas y así poder definir las reglas de dependencia entre ellas.

Es por ello por lo que también en este microservicio “auth-api” se ha implementado uno de los nuevos tipos de arquitecturas limpias, la denominada *Arquitectura Hexagonal*, que se está imponiendo en los proyectos empresariales. De esta forma obtendríamos una serie de ventajas como, por ejemplo:

- Serían independientes del framework.
- Se facilita a que el código sea más testeable.
- Son independientes no sólo de la *UI* (interfaz de usuario) sino también de la base de datos, así como de agentes externos diversos como dependencias o módulos añadidos.
- Además, serían más tolerantes al cambio pues serían más reutilizables al igual que sostenibles en el tiempo.

Por todo ello, la solución la tenemos al implementar *Arquitectura Hexagonal* (también conocida como arquitectura de puertos y adaptadores) pues su principal motivación será separar nuestra aplicación en distintas capas (*Dominio*, *Aplicación* e *Infraestructura*) en la que cada capa tendrá su propia responsabilidad y podrán evolucionar de forma completamente aislada de las otras.

Esta arquitectura se suele representar como un hexágono, donde cada uno de los lados representa un puerto tanto de entrada como de salida de la aplicación. Ahora bien, para ir de una capa más externa a una más interna, se deberá comunicar capa a capa y desde fuera hacia adentro (sin saltarse ninguna capa intermedia). Es decir, la capa más externa, la capa de *Infraestructura* únicamente puede ver y comunicarse con la capa intermedia de *Aplicación*; y la capa intermedia de *Aplicación* podrá ver y comunicarse con la capa más interna de *Dominio*.

La comunicación entre las distintas capas estará definida por puntos de entrada (puertos) e interfaces (adaptadores) que otros módulos (*UI*, *BBDD*, *Test*) puedan implementar y comunicarse con la capa de negocio sin que dicha capa deba conocer cuál es el origen de dicha conexión. Dicho de otra forma, estamos aislando completamente cada una de las capas.

A continuación, se muestra el clásico esquema de la Arquitectura Hexagonal.

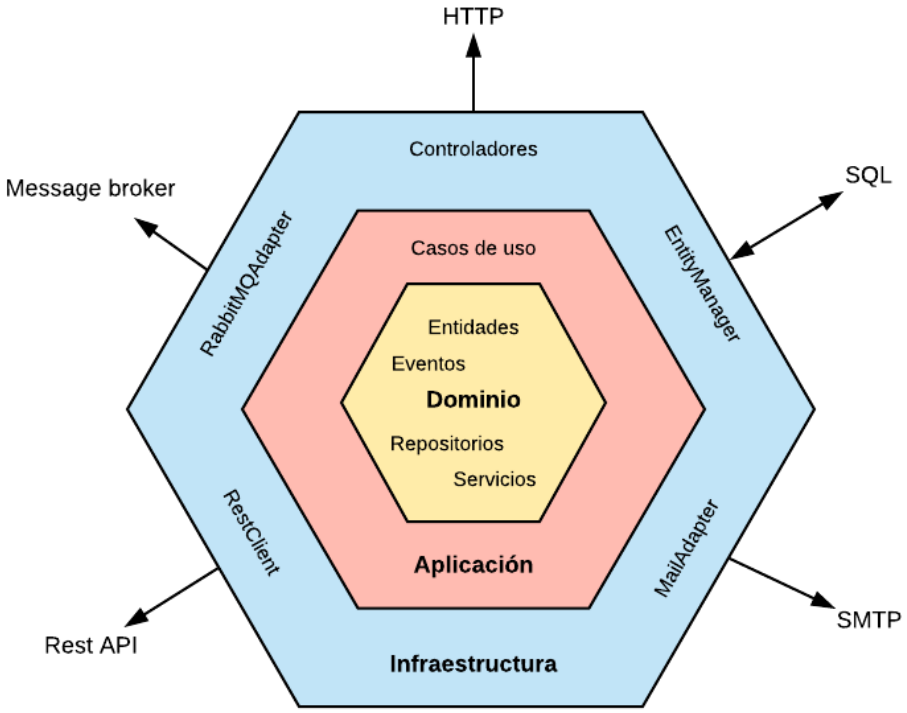

Ilustración 18: Representación clásica del esquema de la Arquitectura Hexagonal

Como podemos apreciar en la ilustración anterior, este tipo de arquitectura se complementa perfectamente con *DDD* (“Domain Driven Desing”). Para entender esta arquitectura debemos tener en cuenta los siguientes conceptos:

- Capa de *Infraestructura*: son los elementos externos con los que se comunica nuestra aplicación, ya sea de entrada como de salida. Los típicos puntos de salida en esta capa es la comunicación con distintas bases de datos; mientras que los puntos de entrada sería la comunicación con una *API* utilizando *REST,* por ejemplo.
- *Puertos*: son los distintos puntos de entrada.
- *Adaptadores*: son los distintos puntos de salida.
- Capa de *Aplicación*: son los servicios que definen la *API* del dominio.
- Capa de *Dominio*: contiene la lógica del negocio de la aplicación la cual puede ser implementada mediante *DDD*.

Con todo ello, puede parecer bastante sencillo a nivel teórico; pero, a nivel práctico es más complejo y más teniendo en cuenta que es la primera vez que incorporo la *Arquitectura Hexagonal*. Aun así, se ha conseguido implementar satisfactoriamente respetando este patrón arquitectónico.

Si bien es cierto que la *Arquitectura Hexagonal* está pensada para grandes códigos, recordemos que uno de los principales objetivos era poder integrar las distintas funcionalidades que podría requerir una empresa real. Aunque el código utilizado no es excesivamente grande, no podría considerarse tampoco pequeño, por lo que el uso de esta arquitectura facilita mucho su lectura.

iv. Patrón Spring Security + JWT, seguridad transversal

Este microservicio “auth-api” es especial pues podríamos decir que contiene 2 grandes bloques. Por un lado, está el bloque que utiliza la *Arquitectura Hexagonal* y será toda la lógica del microservicio y el que contenga las entidades, *DTOs* y validaciones para las tablas y campos presentes en la base de datos que son relativos al usuario.

Por otro lado, tenemos otro bloque que se ocupará de la seguridad y de aplicar el patrón *Spring Security* al incorporar *JWT* (“JSON Web Token”). Esto es necesario pues en caso de no implementar este patrón, nos encontramos ante el problema de seguridad de cómo verificar que el usuario es correcto y de cómo garantizar el cierre de la sesión en caso de quedar abierta.

Por ello, tal como hemos mencionado vamos a utilizar el patrón *Spring Security* que va a resolver este dilema pues vamos a poder tener un control completo sobre el usuario y la sesión al implementar la validación y verificación por *JWT*. Con este fin, deberemos crear un paquete de seguridad que sea el que se ocupe de verificar que los datos recibidos en el inicio de sesión sean correctos y, por consiguiente, pueda generar un token válido. Para llevar a cabo este propósito, se ha creado un paquete transversal de seguridad que es el que se ocupará de toda la gestión del token. En consecuencia, este paquete transversal de seguridad es la única excepción a la *Arquitectura Hexagonal* que existe en este microservicio. Este paquete se llama “security” y está ubicado dentro del microservicio “auth-api”. El bloque “auth-service” estará contenido dentro del paquete “security”.

Por tanto, para lograr explicar la funcionalidad de este paquete haremos uso de las siguientes ilustraciones que explicarían los diferentes escenarios posibles:

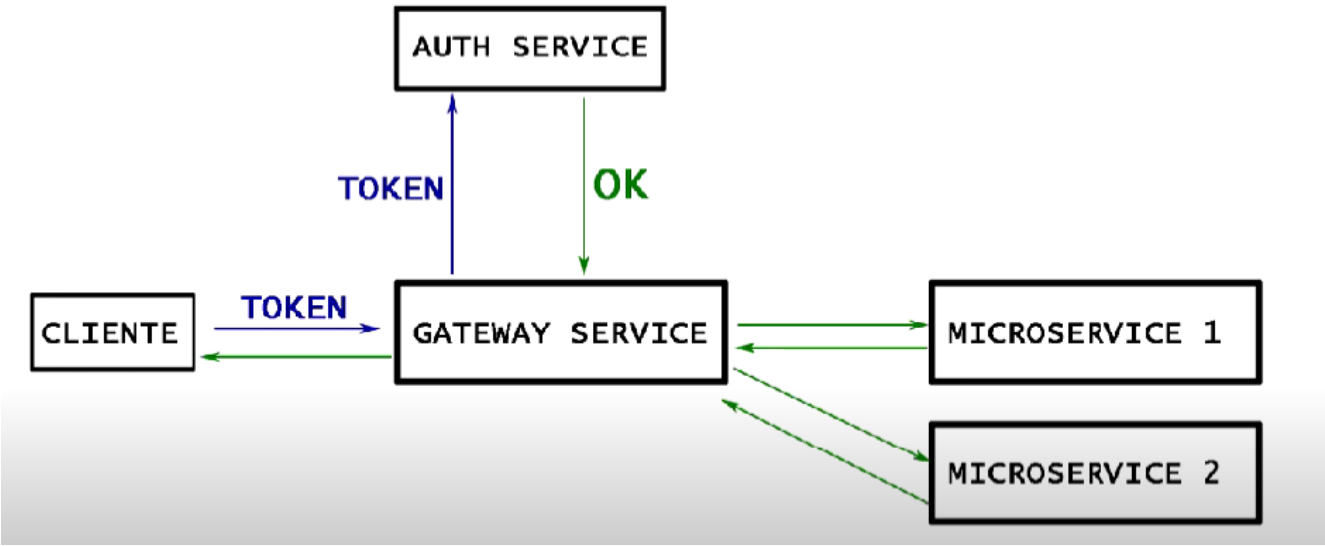

Ilustración 19: Ilustración que muestra el flujo de la validación del token en el escenario positivo permitiendo que la petición del usuario finalmente llegue a los microservicios.

En esta ilustración 19 se puede observar el escenario positivo, en el que el usuario en el momento de loguearse envía el token y éste es finalmente recibido por el bloque del “auth-service” que validará si el token es correcto. En caso de ser correcto, permitirá que el flujo de la petición continúe y, en consecuencia, pueda acceder al microservicio correspondienten a la petición origial del usuario.

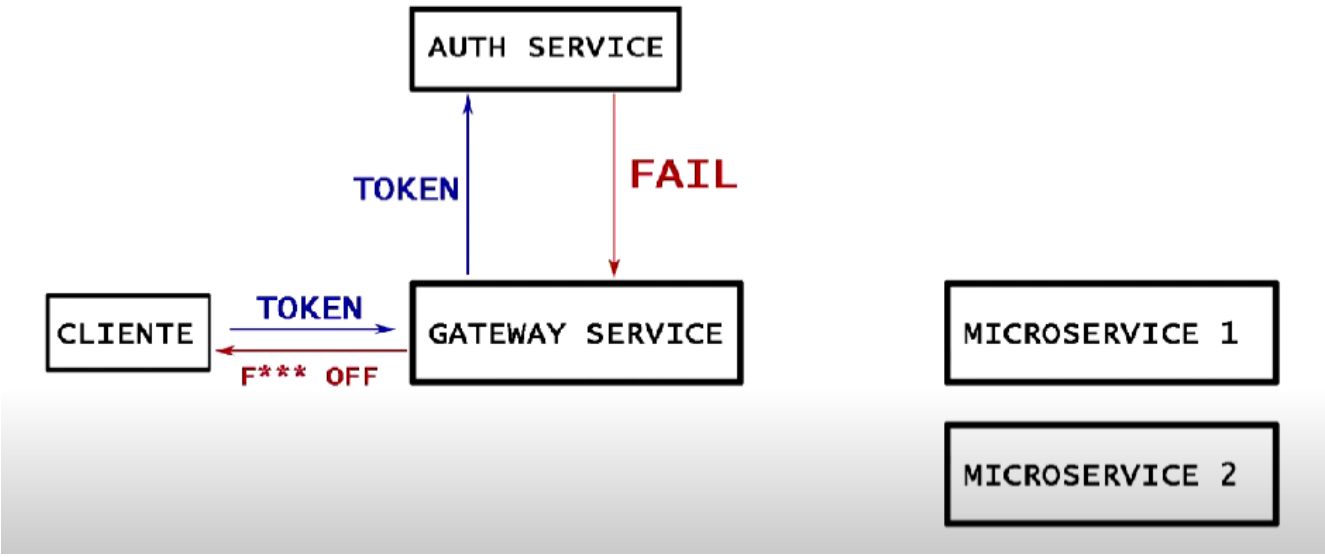

Ilustración 20: Ilustración que muestra el flujo de la validación del token en el escenario negativo impidiendo que la petición del usuario pueda acceder a los microservicios.

En la ilustración 20 se puede observar el escenario negativo. En este caso, el token es inválido o ha caducado y, por tanto, le devolverá el error impidiendo que el usuario pueda acceder al microservicio solicitado.

Tal como se muestra en el Anexo10, este microservicio cuenta con cuatro paquetes generales (“config”, “security”, “shared” y “users”). Mientras que el paquete “config” se ocupa de la configuración del microservicio permitiendo las múltiples instancias de este mismo, el paquete “shared” contiene una serie de entidades que son transversales para varios microservicios.

El paquete “users” será el núcleo de este microservicio ya que se ocupará de toda la gestión de los usuarios (los propios usuarios y sus preferencias de configuración y ajustes) además de los roles. Este paquete es el desarrollado con el patrón de Arquitectura *Hexagonal* (que hemos explicado en la sección anterior), en contraposición al paquete “security”.

Dicho paquete (el “security”), contiene la gestión del JWT, y será más pequeño y transversal por su finalidad, por lo que carece de sentido aplicar el patrón de *Arquitectura Hexagonal*.

Posteriormente, en esta memoria desarrollaremos en profundidad el *Frontend*. Pero por el momento, únicamente lo mencionaremos superficialmente para explicar el token. El token, caracteriza inequívocamente al usuario, y será el único campo relativo a quien es el usuario que vayamos a almacenar en el *LocalStorage* del navegador. De esta forma, podrá ser accesible por cada uno de los componentes de cada microfrontend de nuestro *Frontend*.

Debemos almacenar el token y no el propio usuario debido a que, en caso de que la sesión caduque (el tiempo del token haya expirado), cuando se acuda al *Backend* preguntando por ese token para devolver el usuario, el *Backend* notificará al *Frontend* que habrá caducado haciendo cerrar la sesión en el *Frontend*. Este es el caso negativo que podíamos visualizar en la ilustración 20. De esta forma, lo que conseguimos es prevenir

fallos de seguridad por sesiones activas o abiertas y también prevenimos que alguien pueda modificar voluntariamente el *LocalStorage* para acceder a datos de otros usuarios.

Por último, es fundamental mencionar que para la correcta verificación del token va a ser imprescindible que nuestra clase implemente la clase “UserDetails” tal como puede verse en el Anexo11. Esto se debe a que, a la hora de hacer la autentificación del token, la clase que está esperando es esa pues contiene una serie de argumentos internos sobre los que él mismo trabaja para validar el token.

v. Verificación con Email

Como en cualquier aplicación o sitio web que requiera de un registro, éste puede hacerse de varias formas y etapas. Puede ser un registro de forma local en el que la información de la cuenta queda almacenada en la propia base de datos de esa aplicación o web; o, en contraposición, también se puede utilizar el clásico registro con una cuenta por ejemplo la de Google.

En este caso se ha optado por un registro de forma local con el que la información queda almacenada en la propia base de datos. Con todo, esto supone un desafío pues si bien es cierto que hay forma de verificar que los datos introducidos sean correctos, nada te asegura que dichos datos existan de forma real. Es decir, que la cuenta de correo electrónico proporcionada (aun cumpliendo con el formato de una cuenta de correo) exista y pertenezca al usuario.

Para solventar este desafío, se ha incorporado la verificación por correo electrónico. En otras palabras, en el momento del registro se guardará toda la información en la base de datos, pero habrá un campo de verificación que se guardará a falso. No será hasta que se haya verificado la cuenta de correo, que ese campo se pondrá a verdadero. Por tanto, en el momento del inicio de sesión debemos únicamente permitir iniciar sesión a aquellos usuarios que tengan ese campo a verdadero (pues significará que han verificado la dirección de correo proporcionada).

Con tal propósito se ha modificado el controlador “AuthController” encargado de la validación del token en la creación del usuario, (ubicado dentro del bloque “Auth-Service” del paquete “security” dentro del microservicio “auth-api”) para enviar un correo electrónico a la dirección de correo electrónico proporcionada por el usuario.

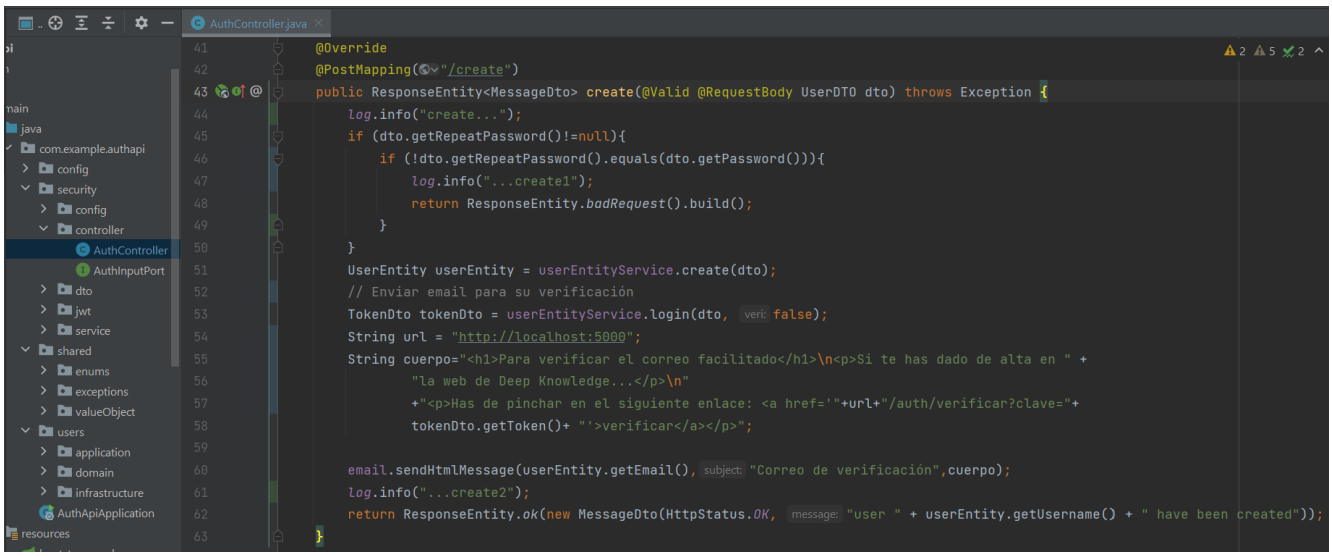

Ilustración 21: Representación del archivo "AuthController.java" ubicado dentro del paquete "security" en el microservicio "auth-api" donde se puede visualizar el código que permite enviar un correo electrónico.

Tal como podemos observar, éste es el código encargado de mandar el correo de verificación. Ahora bien, para poder mandar el correo electrónico será necesario activar una serie de características en nuestra propia cuenta de correo presentes en el Anexo12.

vi. Contenido, Casos de uso y Base de datos

Tal como hemos mencionado, este microservicio "auth-api" se va a ocupar de toda la lógica relacionada con el usuario. Para ser exactos, partiendo del diagrama de casos de uso añadido en el Anexo13, la principal función de este microservicio será pasar del usuario que no ha iniciado sesión y por tanto no está registrado, al usuario que ha iniciado sesión y por tanto está registrado.

Respectivo a la base de datos, este microservicio utiliza una base de datos SQL. De hecho, ambos microservicios ("auth-api" y "res-api") estarán utilizando la misma base de datos. A pesar de que estén utilizando la misma no significa que no sean completamente independientes o, de así quererlo, se pueda separar ambas bases de datos en dos conexiones distintas. El motivo por el que lo he unido es porque en mi caso carecía de sentido tener dos instancias separadas. Se podría decir por tanto que de nuevo hemos aplicado la estrategia de monorepo (que todo dependa bajo una misma carpeta, en este caso base de datos) en lugar de la de multirepo.

Sin embargo, en caso de que una empresa quiera utilizar dos bases de datos distintas pues están en ubicaciones diferentes, no habría ningún problema pues el código ya está preparado para ello.

Además de modificar la configuración del archivo de propiedades externo "auth-api.yaml" ubicado en la carpeta "config-data" en caso de querer cambiar alguna propiedad que no sea critica como el dialecto; lo que sería absolutamente necesario cambiar sería la configuración crítica añadida en el *Vault* donde está la ubicación de la base de datos así como las credenciales para acceder a ella.

Como podemos observar, en ningún momento tenemos la necesidad de modificar el propio microservicio teniendo que pararlo y volver a recompilar, pues lo modificaremos de forma externa sin afectar su ejecución.

vii. Roles, Secured y Granthed Authorities

Tal como mencionábamos en el apartado anterior, dependiendo del tipo de usuario (si ha iniciado sesión o no, o si tiene un rol u otro) las funcionalidades que podrá hacer serán distintas. Para poder diferenciar entre el tipo de permisos y acceso que tiene cada usuario, se le añade el rol correspondiente. Es necesario tener en cuenta que un usuario no tiene un único rol, sino que puede tener una lista de ellos. Por tanto, todo usuario tendrá el rol básico una vez haya creado la cuenta y, dependiendo de las acciones, se le añadirá el rol de estudiante (en caso de comprar un curso) o el rol de profesor (en caso de publicar un curso). Existe un cuarto rol que es el de administrador, el cual no puede ser otorgado con ninguna acción en la propia web, sino que será asignado modificando el valor en la propia base de datos. De esta forma nos aseguramos de que nadie pueda aprovecharse de algún exploit para escalar en privilegios.

No obstante, para proteger los “endpoints” y que un usuario que no tenga permisos (el rol necesario) no pueda ejecutarlo, se debe hacer de forma distinta. Con tal finalidad se ha añadido unas condiciones respecto a los permisos para cada uno de los “endpoints”. Si bien es cierto que se podría haber añadido la condición para cada uno de ellos, se ha optado por delegar dicha responsabilidad en la comprobación del token.

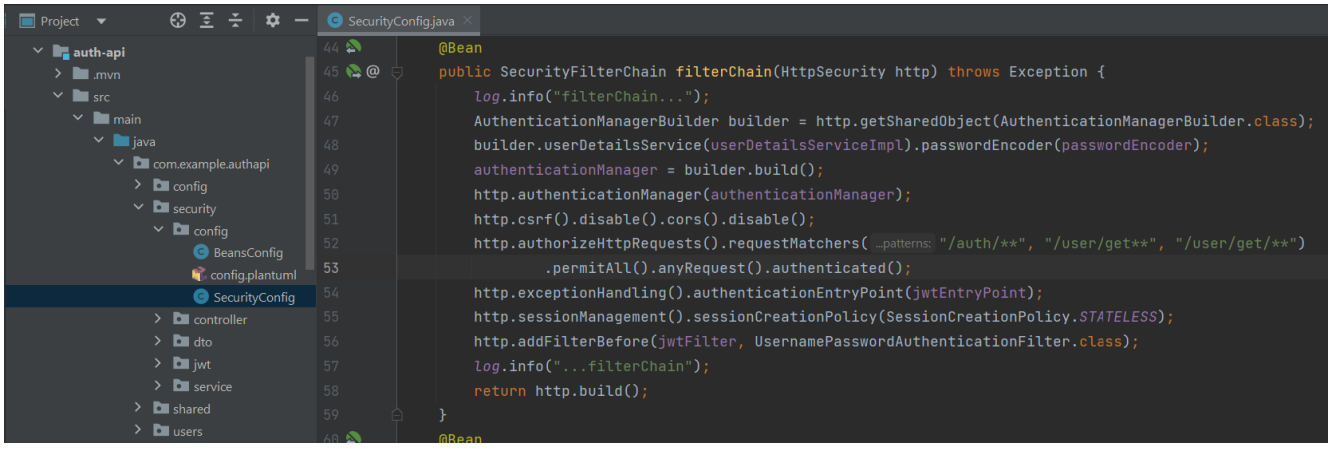

Ilustración 22: Representación del archivo “SecurityConfig.java” dentro del paquete “security” ubicado en el microservicio “auth-api”. En el código se visualiza la configuración para los distintos “endpoints” permitiendo o no su acceso.

Tal como puede observarse en la ilustración anterior, estamos dejando abierto los “endpoints” respectivos a la obtención de los usuarios además de algunas llamadas del “AuthControler” (el controlador encargado de lo relativo al token). Estos “endpoints” serán los responsables de la creación de la cuenta además del registro en la propia web.

Sin embargo, con lo anterior no estamos filtrando por roles; por lo que si lo que queremos es que únicamente un determinado rol pueda acceder a un determinado “endpoint”, será necesario añadir a cada “endpoint” una etiqueta “@PreAuthorize” especificando ese rol concreto. Es importante mencionar que existe la etiqueta “@PreAuthorize” y la “@PostAuthorize”, pero en nuestro caso nos interesa validar que tiene los permisos antes de que pueda obtener los datos o acceder al “endpoint”.

Ilustración 23: Representación del fichero "RolController.java" dentro de la Arquitectura Hexagonal del microservicio "auth-api" donde se está utilizando la etiqueta "@PreAuthorize" para filtrar por un determinado rol.

Como podemos observar, para obetener la lista de todos los roles únicamente lo pueden hacer quellos usuarios que tengan el rol administrador. De todas formas, es una verdad a medias pues el *Secured* de *Spring Boot* no va a buscar el atributo que nosotros hemos definido, sino uno conreto que es el *Granthed Authorities* y que internamente cada rol añadido deberá ser un *Granthed Authorities y* empezar por "ROLE_". Es por este motivo por el que, para cada usuario, además de la lista de roles se le ha añadido un campo que hace referencia a una colección de *Granthed Authorities*, como puede verse en la definición de la clase en el Anexo11.

Por tanto, en caso de modificar nuestra variable con los roles, deberemos modificar esta otra colección para asegurar que el "@PreAuthorize" funciona correctamente.

Ilustración 24: Representación de la conversión de roles a la colección de Granted Authorities en el fichero "UserDetailsService.java" dentro del paquete "security" en el microservicio "auth-api".

Aunque quizás se podría pensar que puede resultar ser algo ineficiente, la realidad es que este paso es absolutamente necesario y el coste que tendría buscar por los elementos existentes de la colección *Granthed Authorities* (eliminando la primera parte del String) podría llegar a ser mayor que el coste de recorrer una lista de muy pocos elementos. En cambio, si se diera el caso que una empresa real implementase esta metodología, sin duda alguna este podría ser un punto para mejorar en caso de trabajar con un gran número de roles posibles.

viii. Password cifrada

Como último método de seguridad en los datos dentro de este microservicio, se ha añadido un cifrado de la contraseña en el momento de guardarla en la base de datos. De esta forma, en caso de que alguien pudiera vulnerar el sistema y acceder a los datos, no podría entrar en la cuenta.

Si tenemos en cuenta el hecho de que uno de los objetivos del TFG es poder ser utilizado por una empresa real, podríamos entrar en un debate interno si quisiéramos implementar el cifrado en la base de datos. Debido a que los otros campos (todos a excepción de la contraseña del usuario) no están cifrados y, por tanto, son legibles desde la propia base de datos. A pesar de que no existe ningún impedimento para cifrar todos y cada uno de los datos guardados en la base de datos, mi propia experiencia tras trabajar para una empresa me ha demostrado que, por suerte o por desgracia, a niveles prácticos no siempre se suelen cifrar todos los datos.

Es por este motivo que, al tratase de un TFG, he decidido dejarlo así tal como se hacía en la empresa donde yo trabajé. Con todo, en caso de querer ser implementado por una empresa que quiera cifrarlos para así aumentar la seguridad, lo único que se deberá hacer es llamar al encoder (ubicado en el módulo transversal “security” dentro de este microservicio “auth-api”) para cada uno de los campos.

Después de un largo periodo viajando por los territorios del microservicio “auth-api”, a continuación podremos observar en el horizonte lo que será nuestra decimotercera estación: las pasarelas de pago del microservicio “braintree-api”.

g. Braintree Api

Dejado al margen el tema de la autentificación ahora hablaremos del microservicio “braintree-api” dentro del contenedor de color verde que hemos mostrado al inicio de todo y está presente en el Anexo1.

i. Funcionalidad y Objetivo

En la inmensa mayoría de aplicaciones y páginas web, llega un cierto momento en el que se debe realizar una transacción para poder comprar o adquirir un elemento. Para poder realizar esta transacción podríamos optar por dos alternativas:

Una opción podría ser la de solicitar al usuario los campos necesarios para después poderlos guardar en nuestra base de datos y más adelante realizar la transacción con los datos guardados. Ahora bien, esto implica un gran problema de seguridad, pues significa que estás dando la información del método de pago seleccionado para ser almacenada. Es decir, poniendo el ejemplo de una tarjeta bancaria, lo que el usuario estaría haciendo sería permitir que la información sobre su cuenta bancaria, (código de seguridad, fecha de expiración y nombre de tarjeta) quedasen, estos campos, registrados y guardados en la base de datos, por lo que el administrador podría tener acceso a esos datos y hacer con ellos lo que considere. Es por ello por lo que se implementó el servicio de la pasarela de pago.

Una pasarela de pago no deja de ser un proveedor de servicios de aplicación para el comercio electrónico con el que se pueden autorizar pagos a negocios en línea. Existen

muchos tipos distintos de pasarelas de pago pues el tema de las transacciones es extenso y se puede llegar a profundizar hasta niveles insospechados. Dado que, ese no es el objetivo de este proyecto, únicamente nos centraremos en el ámbito que nos afecta como desarrolladores que implementamos el uso de las pasarelas de pago.

Consecuentemente, el objetivo del microservicio "braintree-api" será precisamente la gestión de todo lo relacionado con la transacción y la pasarela de pago.

Como podemos observar, este microservicio se encuentra aislado pues realmente no necesita nada de ningún otro, a excepción de los recursos y el usuario que realiza dicha transacción. Si bien, esta tarea no la realizará el microservicio "braintree-api", sino que la realizará el microservicio "res-api" después de adquirir toda la información necesaria a través del *Frontend*.

ii. Patrones, Características y tecnologías utilizadas

A diferencia del microservicio anterior de "auth-api", en este ("braintree-api") se ha implementado:

- La integración con múltiples instancias ("Load Balanced").
- El uso de *Vault*.
- Una validación extra para el *CORS*, añadiendo la etiqueta "@CrossOrigin" en cada uno de los controladores del microservicio.

Por tanto, este microservicio carece de *Arquitectura Hexagonal* pues es demasiado pequeño ya que únicamente tiene una funcionalidad concreta y específica.

iii. Pasarela de Pago Braintree

Existen diversas pasarelas de pago como, por ejemplo, *Redsys*, *Stripe*, *Braintree*… La elección de una u otra dependerá del equipo empresarial y del equipo de desarrolladores. En este caso, la pasarela de pago que he decidido integrar es la de *Braintree*, pero el funcionamiento sería equivalente a cualquier otra.

Llegados a este punto es necesario destacar dos cuestiones pues la integración de una pasarela de pago implica que las operaciones económicas tienen la arquitectura para poder ser reales.

- La primera es que al utilizar ese servicio de pasarela de pago realmente no estamos obteniendo la integridad total de la transacción, sino que existe una pequeña comisión que son las ganancias de quienes ofrecen dichos servicios de transacciones.
- La segunda es que existen dos modos: el modo "real" y el modo "sandbox". Mientras que en el primero las transacciones son completamente reales, en el segundo son ficticias y están pensadas para un entorno de desarrollo. En la página web de este trabajo, las credenciales de la pasarela de pago son para el modo "sandbox" por lo que ninguna transacción realizada en la web tendrá un cargo económico de forma real. Evidentemente, en caso de querer utilizar esta mecánica por una empresa real, el único cambio que se debería hacer sería en las credenciales proporcionadas, que están ubicadas en el *Vault*.

```
braintree-api
{
  "braintree.merchant-id": "XXXXXXXXXXXXX",
  "braintree.public-key": "YYYYYYYYYYYYY",
  "braintree.private-key":  "ZZZZZZZZZZZZZZZZZZZZZZZZZ"
}
```

Ilustración 25: Representación de la configuración necesaria que hay que añadir en Vault para el microservicio "braintree-api". Acuda al Anexo6 donde se muestra como añadir un nuevo secreto en Vault.

En esta imagen podemos observar las credenciales (están modificadas por cuestiones de seguridad) que se deben proporcionar al código para que pueda utilizar este servicio de pasarela de pago.

Como desarrolladores para poner en funcionamiento este servicio de pasarela de pago no debemos implementar todo el flujo de información, sino unicamente la sección relativa al Servidor y el Cliente pero sin la necesidad de ocuparnos de la gestión interna que tiene los propios servidores de *Braintree*.

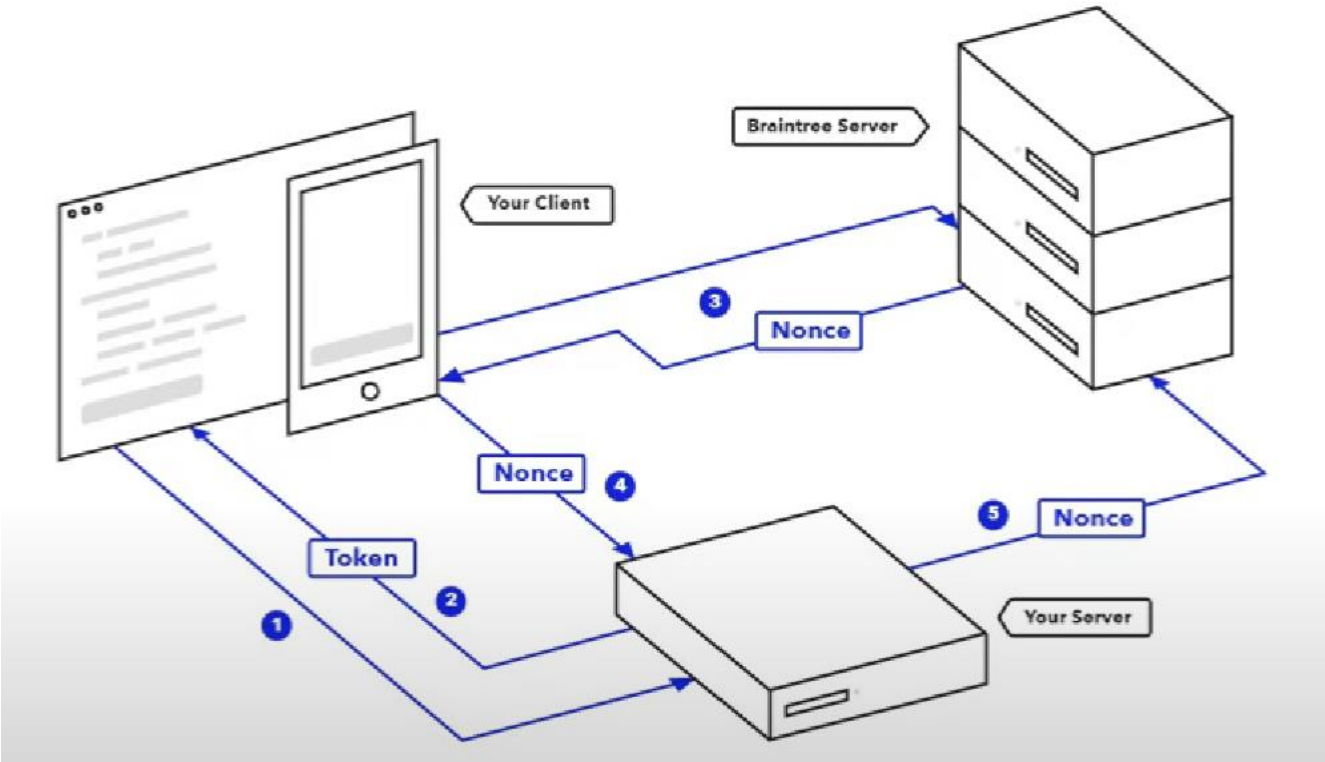

Ilustración 26: Representación del flujo interno de llamadas y pasos que se requiere para realizar la integración con la pasarela de pago Braintree.

El funcionamiento que debemos realizar es bastante sencillo. Primero de todo el Cliente a través del *Frontend* solicitará el token (paso1) y el Servidor se lo otorgará (paso2). A continuación, con este token el Cliente podrá introducir los datos de su forma de pago y se mandará una petición a los servidores de *Braintree* (paso3). Más adelante los servidores devolverán una variable fantasma denominada "nonce". Nosotros no tenemos control sobre ella ya que nos es completamante invisible, pero será una variable de un sólo uso pues será

la que autorice que la transacción se realice. Una vez el *Frontend* ha recibido la variable automáticamente y sin ninguna interacción por parte del usuario, se mandará dicha varibale automaticamente al Servidor (paso4), el cual la enviará a los servidores de *Braintree* como si fuera un checkout a modo de comprobación (paso5). En caso de salir todo correctamente, la transacción quedará reflejada cuando el "nonce" se verifique en los servidores extrernos de *Braintree* pues será cuando se haga efectiva la transacción.

iv. Contenido, Casos de Uso y Base de Datos

Este microservicio "braintree-api" no necesita conectarse a ninguna base de datos, sino que únicamente necesitará la configuración que le proporcionará el *Config Server* y aquella almacenada en *Vault* que contendrá las claves para poder acceder a la cuenta creada en *Braintree*.

El contenido de este microservicio tal como habíamos mencionado no es muy extenso, pues únicamente realiza una función concreta, la de permitir al usuario registrado poder comprar cursos. Esta funcionalidad será representada en los casos de uso presentes en el Anexo13.

Después de "pagar las tasas" asociadas a nuestro viaje, abandonamos el sector de las pasarelas de pago y nos embarcamos en lo que será nuestra última parada en el mundo del *Backend*.

h. Res Api

Para acabar, el último microservicio que nos falta por ver en el *Backend* es el que corresponde a "res-api" (ubicado junto con el "auth-api" y el "braintree-api" en el contenedor de color verde del esquema principal de esta sección y en el Anexo1). Este microservicio englobará el resto de las funcionalidades que contendrá nuestra página web.

i. Funcionalidad y Objetivo

Aun cuando el concepto de microservicio reside en ser un pequeño servicio independiente del resto, eso no significa ni mucho menos que para cada una de las entidades o tablas deba crearse un microservicio aparte. El tamaño o, mejor dicho, las funcionalidades que englobe cada microservicio dependerán en gran medida de los criterios del equipo empresarial y de desarrollo. En este caso, decidí englobar aquellas entidades que están relacionadas unas con otras en cada uno de los dos microservicios principales. Por un lado, el microservicio "auth-api" con la gestión de los usuarios (como ya hemos visto anteriormente), y por este otro lado la gestión de los recursos que engloba a cada usuario presentes en este microservicio "res-api".

Por tanto, en este microservicio nos centraremos en otorgar al usuario las siguientes funcionalidades:

- La gestión de los cursos, de las publicaciones asociadas a cada curso y de los ficheros asociados a cada publicación.
- Además, también controlaremos los mensajes de chat pertenecientes a cada uno de los cursos, así como las categorías que serán añadidas a los cursos.

Tal como podemos observar el objetivo de este microservicio será gestionar toda la información que tiene relación con los cursos, pues son estos el eje principal de nuestra aplicación web.

ii. Patrones, Características y tecnologías utilizadas

De forma idéntica al microservicio “auth-api” este ofrece las mismas características, patrones y tecnologías utilizadas:

- La integración con múltiples instancias (“Load Balanced”).
- El uso de *Vault*.
- Una validación extra para el *CORS*, añadiendo la etiqueta “@CrossOrigin” en cada uno de los Controladores del microservicio.
- Integración con *Arquitectura Hexagonal* y *DDD.*
- Implementación del *Secured* para los “endpoints”.

Sin embargo, existen dos grandes diferencias con el microservicio “auth-api”. Mientras que la primera es que no es necesario el uso de *JWT* debido a que esa funcionalidad pertenece en exclusiva a ese microservicio. La segunda consiste en la integración con *Firebase*.

iii. Base de Datos relacional vs no relacional. MongoDB vs Firebase.

A la hora de querer almacenar tus datos y, por tanto, escoger un tipo de base de datos; primero hay que pensar en cuál de los dos tipos de base de datos necesitará tu aplicación. De todas formas, hay que tener en cuenta que no es el objetivo de este TFG profundizar sobre las bases de datos, pues es un mundo muy extenso. Por ello, únicamente mencionaremos los aspectos más relevantes para el desarrollo de este trabajo.

Por un lado, tenemos las bases de datos relacionales en donde se especifica cada tabla y cada atributo y tipo de dato que contiene dicha tabla. Además, en este tipo de base de datos también deberemos especificar las relaciones entre los distintos campos de las diversas tablas, formando así unas llaves primarias y otras foráneas.

Mientras que las llaves primarias, que normalmente son los identificadores (“ids”), lo que harán será identificar inequívocamente cada dato de entre todos los otros datos presentes en la tabla. Las llaves foráneas, serán campos en las tablas que apunten a esos identificadores, por lo que podremos hacer referencia entre varias tablas.

Por el otro lado, tenemos las bases de datos no relaciones. En esta no se especifica los atributos para cada tabla y dato que queramos guardar en nuestra base de datos. Sino que, por el contrario, cada dato guardado (entrada en nuestra base de datos) podrá tener campos distintos. De forma equivalente también se puede generar un identificador único para cada entrada de nuestra base de datos.

Ahora bien, teniendo en cuenta que anteriormente habíamos mencionado que el microservicio “auth-api” y este microservicio “res-api” utilizan la misma base de datos *SQL*, es importante mencionar que ese tipo de base de datos es de tipo relacional. Esto significa que la definición de las entidades (tablas) y de sus campos estará definida en el propio código en el apartado de *Infraestructura* de la *Arquitectura Hexagonal*.

No obstante, hay que tener presente que, si alguien quisiera probar el código, primero debería crear un “schema” vacío con el nombre de “TFG”, pues será el nombre de la base de datos a la que se intente conectar; y, en caso de no existir, dará error.

Adicionalmente, para guardar los ficheros estamos utilizando una segunda base de datos. Pero, en este caso, será no relacional. Concretamente estamos utilizando el *Storage* de *Firebase* que está pensado específicamente para tal propósito (guardar archivos).

Si bien es cierto que originalmente se pensaba guardarlos utilizando una base de datos *MongoDB*, después de documentarme sobre cómo se tratan internamente los ficheros, pude comprobar que, aunque a fines prácticos sea equivalente, *Firebase Storage* está pensado para ello. No obstante, en caso de querer cambiar la base de datos a una *MongoDB* no habría ninguna dificultad pues nicamente sería cambiar las propiedades que están almacenadas como secretos en *Vaul.*

Cabe destacar que la única interacción que tiene este microservicio “res-api” con la gestión de ficheros en *Firebase* es proporcional al *Frontend* las credenciales que están almacenadas como secretos. De esta forma, evitamos tener que añadir en el propio microservicio del *Frontend* “mf-resources-angular” las credenciales de una forma visible y accesible.

Si bien es cierto que se podría recibir el fichero desde el microservicio del *Frontend* y enviarlo hasta el *Backend*, haciendo así que fuera el propio “res-api” en el *Backend* quien guardase el fichero; se ha optado por delegar la responsabilidad al microfrontend “mf-resources-angular” debido a lo aprendido en las prácticas de empresa donde la gestión de ficheros era por parte del *Frontend*. No obstante, dependiendo de lo que se desease podría implementarse tanto en uno lado como en el otro.

iv. Contenido y Casos de Uso

Tal como hemos mencionado este microservicio “res-api” se conecta a la misma base de datos SQL que el microservicio “auth-api”, además de proporcionar las credenciales al microfrontend sobre la conexión a *Firebase Storage*.

El contenido de este microservicio realmente es muy extenso, pues va a realizar todas las otras funcionalidades presentes en el diagrama de casos de uso del Anexo13 que no hagan los otros dos microservicios anteriores (“auth-api” y “braintree-api”). Por tanto, se ocupará de toda la gestión de los cursos y sus derivados (publicaciones y ficheros).

A continuación, una vez acabado todo el recorrido por las distintas paradas en el *Backend*, nos dirigiremos a cruzar el túnel que nos llevará hacia el mundo del *Frontend*, donde una tierra completamente desconocida y nueva nos aguarda. En nuestra travesía por el conocimiento, nuestra decimoquinta parada estará en la misma frontera del *Frontend*.

9. Implementación del Frontend

a. Resumen Global

En la sección anterior, cuando introdujimos el mundo web presentamos los conceptos de *Backend* y *Frontend*; dado que ambos serán necesarios para la creación de la página web. En esta sección, nos centraremos exclusivamente en el *Frontend* dejando totalmente al margen el *Backend* (ya explicado anteriormente). Respecto a la construcción del *Frontend*, observaremos que sigue exactamente la misma estructura conceptual, salvo algunas diferencias:

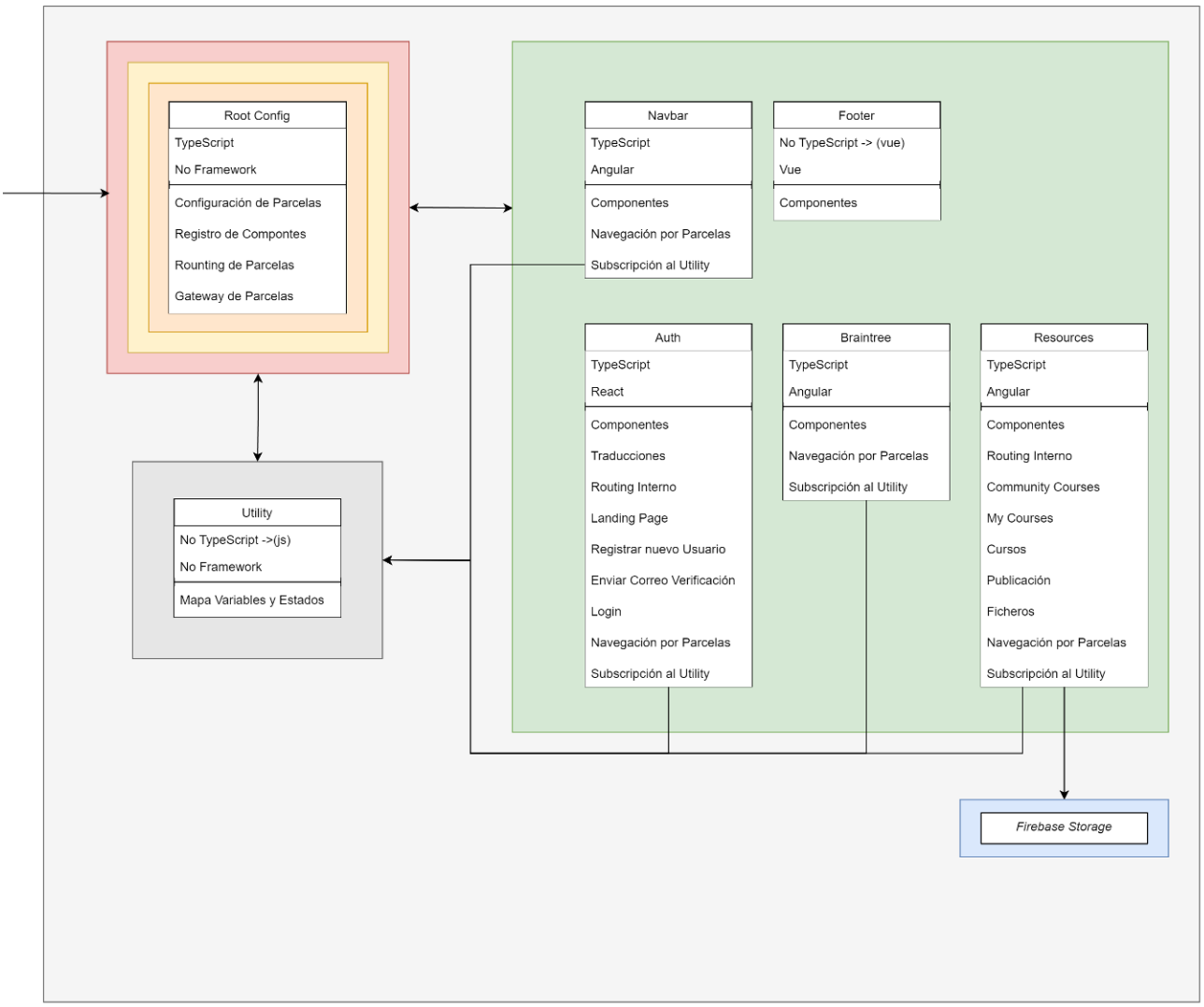

Ilustración 27: Representación final de la estructura del Frontend. Esta figura es la sección izquierda de la ilustración2 presente en el Anexo1.

Tal como podemos apreciar, esta es la mitad derecha que complementa el diagrama presentado en la sección del *Backend*. El significado de las flechas y de los colores de cada uno de los contenedores es el mismo. En este caso también tenemos **módulos** (los recuadros blancos) y los **contenedores** (los recuadros de colores) que engloban a uno o diversos módulos.

Como podemos apreciar el módulo *Root Config* está contenido por tres contenedores (naranja, amarillo y rojo). Esto se debe a que las funcionalidades de cada uno de los contendores las engloba este único módulo. Es decir, el módulo *Root Config* hará el equivalente a las funcionalidades que en el Backend necesitábamos tres módulos distintos:

- El equivalente en *Backend* al *Config Server* cuyo nombre del microservicio era "config-service" (ubicado dentro del contenedor amarillo).
- El equivalente además al *Discovery Eureka Server* cuyo nombre de microservicio en el *Backend* era "eureka-service" (ubicado dentro del contenedor naranja).
- Por último, el equivalente al microservicio del *Gateway Service* cuyo nombre era "gateway-service" (ubicado dentro del contenedor rojo).

De forma equivalente en el contenedor verde tenemos los mismos tres módulos que teníamos antes con los microservicios ("auth-api", "braintree-api" y "res-api") pero ahora serán microfrontends ("mf-auth-react", "mf-braintree-angular" y "mf-resources-angular" respectivamente). De igual manera, se incorporan dos nuevos microfrontends a este contenedor que serán: la *Navbar* ("mf-navbar-angular") y el *Footer* ("mf-footer-vue").

Además, para el microfrontend de *Resources* ("mf-resources-angular") tenemos una conexión con los servicios de *Firebase* que, si recordamos, fue introducida cuando explicamos (en la sección del *Backend*) el microservicio "res-api" y que explicaremos más en detalle (en la sección del *Frontend*) en el apartado correspondiente a este microfrontend.

Por último, podemos apreciar un último módulo en el contenedor gris que hará una función parecida al *Vault* del *Backend*. Pero en este caso no almacenará secretos, sino que actuará como una librería compartida entre la mayoría de los módulos (o, en otras palabras, microfrontends) del contenedor verde. De esta forma, y a través de suscripciones y observadores, podremos notificar a los microfrontends cuando en uno de ellos se produzca un cambio concreto (el inicio de sesión y el cierre de sesión por "logout" o por caducidad del token) por ejemplo.

Si continuamos nuestro viaje, en la próxima estación tendremos que decidir por cuál de las dos rutas continuar, cado que cada una de ellas nos ofrece un camino distinto.

b. Microfrotend (Module Federation y SingleSpa)

De forma totalmente equivalente a la estructuración del *Backend* entre un código monolítico y un código basado en microservicios; en el *Frontend* podemos aplicar el mismo concepto.

A pesar de que en su inmensa mayoría todos los *Frontends* son monolíticos, los microfrontends (el equivalente del *Backend* a los microservicios, pero en el *Frontend*) son el futuro. Es por este motivo que cada vez más se está incorporando este tipo de arquitectura en la creación de aplicaciones web.

El microfrontend es un tipo de arquitectura donde la página web se divide en distintos módulos individuales. Estos serán implementados de forma autónoma e independiente entre cada uno de ellos. Haciendo posible que cada equipo de *Frontend* tenga la máxima flexibilidad y velocidad posible pues únicamente debe centrarse y conocer ese microfrontend. Además, al ser completamente independiente entre ellos, permite poder utilizar no sólo diferentes *frameworks* o lenguajes, sino también distintas versiones de éstos.

Para trabajar con microfrontends existen dos opciones: utilizar *Module Federation*, o usar *SingleSpa*. Aunque, también existe la alternativa de utilizar ambos a la vez y, en consecuencia, el control y manejo que puedes llegar a tener es absoluto.

i. SPA vs MPA

Toda página web debe decidir cómo actualiza su propia información o navegación interna. Por ello, primero es necesario esclarecer el significado de los siguientes términos:

- Por un lado, *SPA* o "Single Page Application" significará que en el momento que el servidor envíe la información que el Frontend necesita, a cada click, el navegador podrá renderizar la información nueva sin la necesidad de recargar la página de nuevo.
- Por el otro lado, *MPA* o "Multi Page Application" significará que, a cada nueva petición de datos, con cada nuevo cambio, se renderizará en el navegador una nueva página desde cada petición del servidor.

Teniendo en cuenta estos dos funcionamientos distintos sobre cómo visualizar la información que recibe el Frontend desde el Bakcned, ahora podremos discernir sobre cuál de las dos opciones (*Module Federation* o *SingleSpa*) usar para crear nuestros microfrontends.

ii. Module Federation (MF)

"Webpack Module Federation" (*MF*) es una característica de webpack que permite la carga dinámica de múltiples versiones de un módulo desde múltiples sistemas de compilación independientes.

Esto permite la creación de aplicaciones de estilo microfrontend, donde varios sistemas pueden compartir código y actualizarse dinámicamente sin tener que reconstruir toda la aplicación.

También permite que equipos y aplicaciones distribuidos con diferentes ciclos de lanzamiento compartan código sin necesidad de esperar a que todos los sistemas acepten e implementen una única versión compartida de un módulo.

Además, permite dividir el código según las rutas y otros criterios, lo que puede mejorar el rendimiento.

Para utilizar *MF* lo que haremos será construir dos aplicaciones separadas de tipo *SPA* que usarán *MF* para compartir componentes durante el tiempo de ejecución.

La *Aplication-A* contendrá un "SayHelloFromA" que será consumido por la *Aplication-B*. Mientras que la *Aplication-B* tendrá un componente "SayHelloFromB" que será consumido por la *Aplication-A* tal como se muestra en la siguiente imagen:

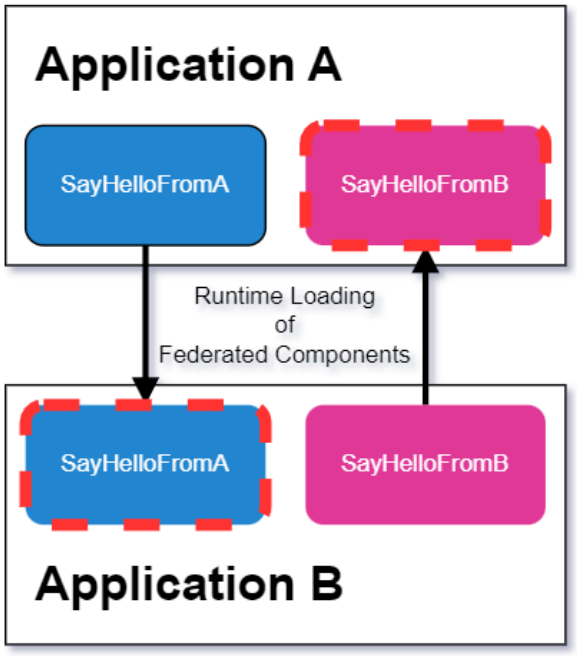

Ilustración 28: Representación del funcionamiento de Module Federation. En la figura se puede observar cómo ambas aplicaciones están exportando su propia instancia, permitiendo que la otra aplicación pueda verla y comunicarse.

Esta arquitectura permitirá que cada *SPA* se desarrolle e implemente de forma independiente junto con la recepción instantánea de actualizaciones de otras *aplicaciones federadas* (microfrontends utilizando *MF*) sin implementaciones.

iii. SingleSpa

SingleSpa es un *framework* que permite reunir múltiples microfrontends *JavaScript* en una sola aplicación *Frontend*. De esta forma, estaremos creando una arquitectura para nuestro *Frontend* utilizando un sólo *SPA*. Al hacer esto, obtendremos una serie de beneficios, tales como:

- Utilizar múltiples *frameworks* (*React*, *AngularJS*, *Angular*, *Ember*, *Vue*…) en una misma página, sin necesidad de recargar la página.
- *Además, permitirá deployar* (lanzar a producción) cada microfrontend de manera independiente.
- Así mismo facilitará escribir código usando un nuevo *framework* sin la necesidad de reescribir tu aplicación existente.
- También mejorará la optimización del código al tener precargadas algunas páginas.

Debemos tener en cuenta que *SingleSpa* es una nueva arquitectura avanzada. Debido a que cambia el paradigma de cómo se realizan normalmente las aplicaciones *Frontend*, se necesitará la comprensión de las herramientas subyacentes necesarias para construir un *SingleSpa* con microfrontends.

En este TFG precisamente se ha utilizado *SingleSpa* en lugar de *Modules Federation* para la creación de los microfrontends. El motivo de esta elección fueron principalmente dos aspectos: un mayor conocimiento sobre *SingleSpa* y el hecho de que la arquitectura de *SingleSpa* integra varias configuraciones por defecto en su creación, que pasan a ser más o menos transparentes para el usuario.

Ante todo, para comprender cómo funciona *SingleSpa* debemos definir tres conceptos: *Root Config*, *Parcela* y el módulo *Utility*.

Por un lado, tal como habíamos mencionado anteriormente, *SingleSpa* es una *SPA* y, por tanto, todo se visualiza en una única página que no hace falta recargar. Esta única página que lo engloba todo es el *Root Config*. El *Root Config*, por tanto, lo podemos entender como un contenedor que está vacío, en el que cada una de las aplicaciones microfrontend se visualizarán dentro de este contenedor madre.

Por otro lado, cada una de nuestras aplicaciones microfrontend se denominarán *Parcelas*. Cada una de estas *Parcelas* podemos entenderlas como un contenedor independiente que se ejecuta y se visualiza dentro del contenedor raíz (el *Root Config*). Aunque existe una sutil diferencia, a lo largo de este documento cuando nos refiramos a microfrontend será equivalente a las *Parcelas* que no sean ni el *Root Config* ni el módulo *Utility*. Es importante señalar que estos dos (*Root Config* y el módulo *Utility*) no son *Parcelas* como hemos definido, pues ambos son microfrontends especiales y distintos al resto de microfrontends. Es por ello que todas las *Parcelas* serán microfronteds pero no todos los microfrontends serán de tipo *Parcela*.

En la siguiente imagen se representa el contenido del *Root Config* siendo éste el fondo gris, mientras que cada una de las *Parcelas* (microfrontends) está por encima, en blanco.

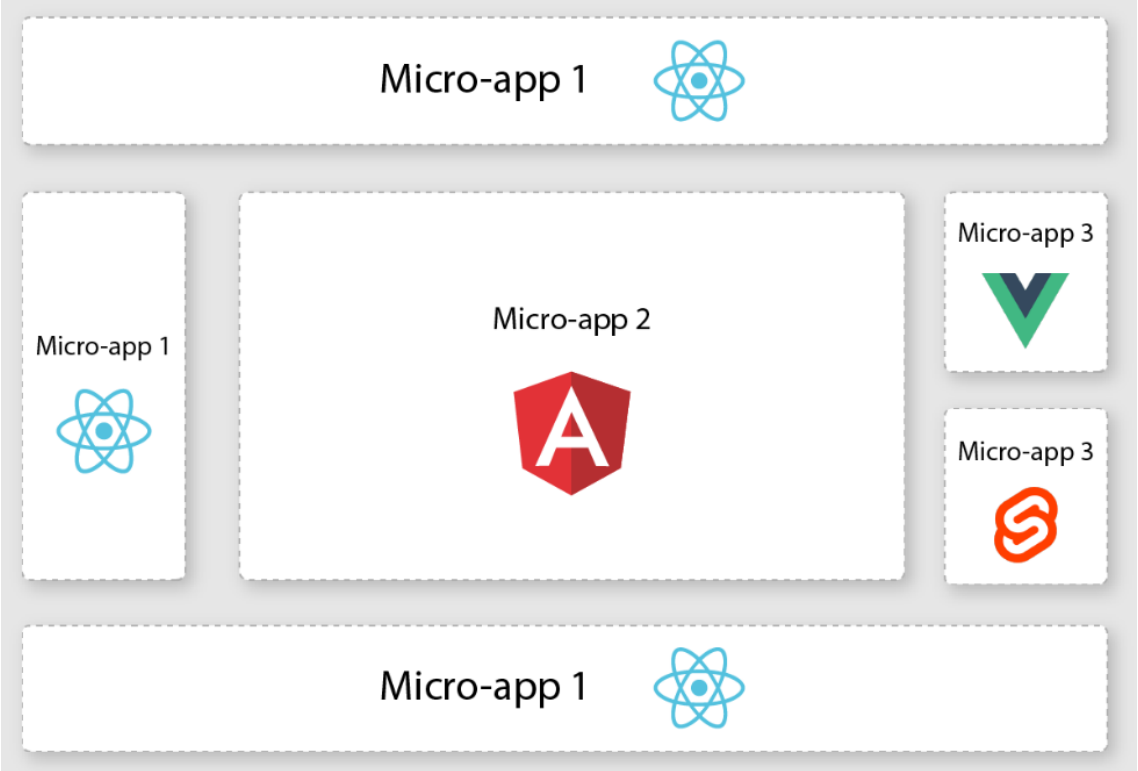

Ilustración 29: Representación de los microfrontends utilizando SingleSpa. En la figura observamos el Root Config como el fondo gris mientras que cada una de las Parcelas está representada como una micro-app.

Por último, y tal como habíamos previamente mencionado, tenemos un contenedor de utilidades, el módulo *Utility*. Este módulo no se visualizá, sino que actúa como una biblioteca transversal para todos los otros microfrontends para poder compartir datos a través de ellos.

Después de haber escogido la vía de *SingleSpa,* nos dirigimos a la que es nuestra decimoséptima parada en esta gran travesía. En ella veremos el microfrontend principal de *SingleSpa*.

c. SingleSpa Root Config (micro-frontend-app)

Como hemos mencionado anteriormente, el *Root Config* (tal como su nombre indica) se encarga principalmente de toda la gestión, organización y configuración; pues será el contendor raíz donde se visualizarán cada una de las *Parcelas*.

Dado que no existe una estructuración concreta de carpetas para los distintos módulos, en mi caso lo he estructurado de la siguiente forma (tal como puede observarse en el Anexo14).

La carpeta principal (“micro-frontend-app”) al mismo tiempo que encapsula a todas las *Parcelas* y al módulo *Utility*, también es el *Root Config*. Es fundamental entender que el hecho de que las *Parcelas* estén dentro del *Root Config* no significa que no sean completamente independientes; sino que únicamente es a nivel de estructuración interna de ficheros. Similar a como habíamos visto en el *Backend* con una estructura de monorepo y multirepo; podríamos decir que el *Frontend* también es un monorepo de microfrontends donde la carpeta principal es el propio *Root Config*.

SingleSpa proporciona una arquitectura y jerarquía de ficheros para este contenedor, por lo que únicamente deberemos modificar tres ficheros concretos.

i. Configuración de fichero ejs

El fichero más importante y el que será el núcleo de absolutamente todo este microfrontend, será el fichero “index.ejs”.

En este fichero no sólo se van a definir cada una de las *Parcelas* y cuáles son sus ubicaciones, sino que también se va a necesitar importar todas y cada una de las librerías necesarias que usen dichas *Parcelas*.

Tal como se puede ver en las imágenes, existen dos zonas para definir nuestras *Parcelas*. Por un lado, en la primera imagen se importan aquellos componentes que se requerirán transversalmente o que estén en producción. Mientras que en la imagen inferior se importaran las parcelas que se ejecuten en local.

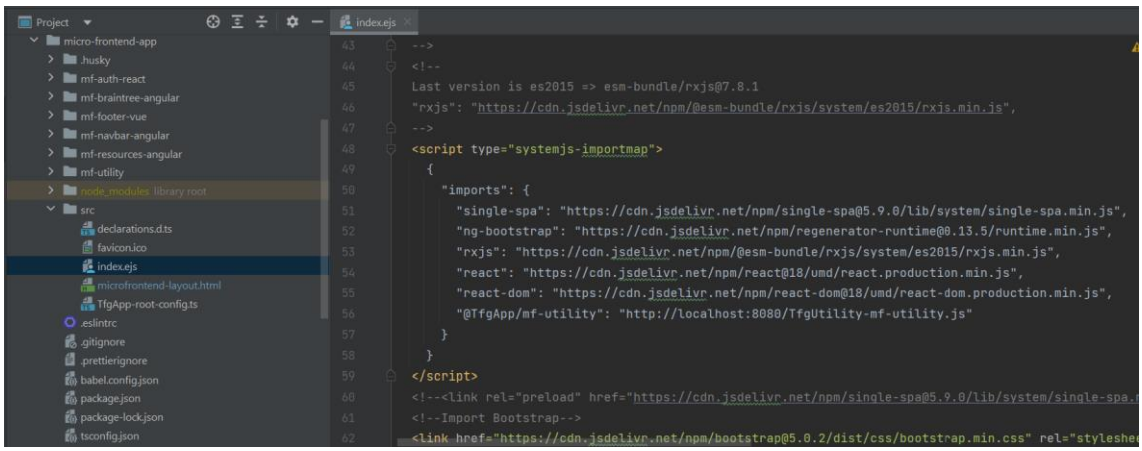

Ilustración 30: Representación del fichero "index.ejs" del microfrontend "micro-frontend-app" que actúa como Root Config en SingleSpa. En la figura puede observarse los módulos e imports a nivel de producción.

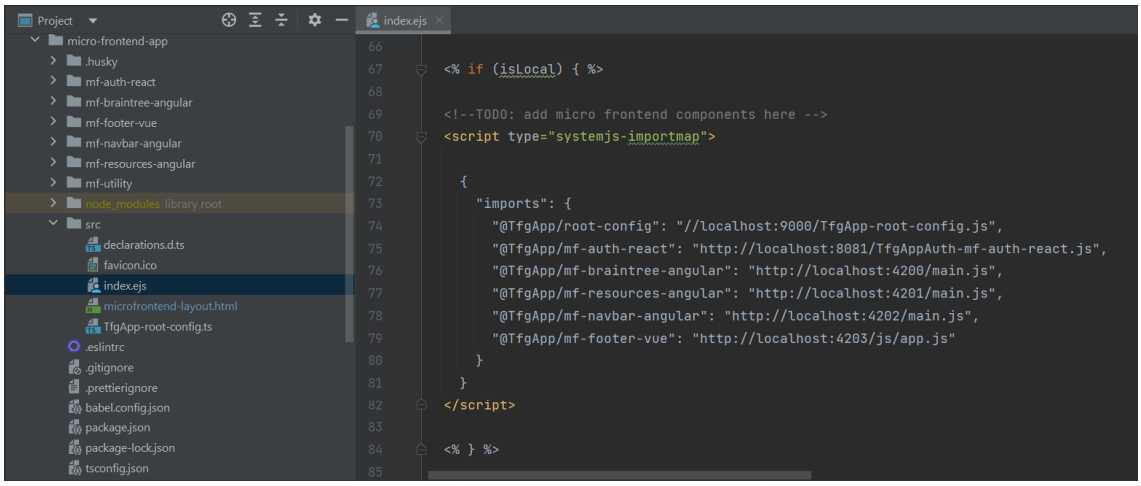

Ilustración 31: Representación del fichero "index.ejs" del microfrontend "micro-frontend-app" que actúa como Root Config en SingleSpa. Esta ilustración es complementaría con la ilustriación30. A diferencia de la anterior, en esta figura puede observarse los módulos e imports a nivel local.

Por otro lado, en lo relativo a las librerías de terceros como por ejemplo *Material Icons* o el *Bootstrap*, será necesario que, las importemos en este fichero (tal como se muestra en el Anexo15) en lugar de importarlas dentro de nuestras *Parcelas* que sería el procedimiento normal en caso de usar un *Frontend* monolítico.

ii. Configuración de fichero html (Routing)

En el segundo fichero nos encontramos con el fichero "microfrontend-layout.html". En este fichero se va a definir no sólo las rutas, sino también qué es lo que se va a visualizar en cada una de ellas.

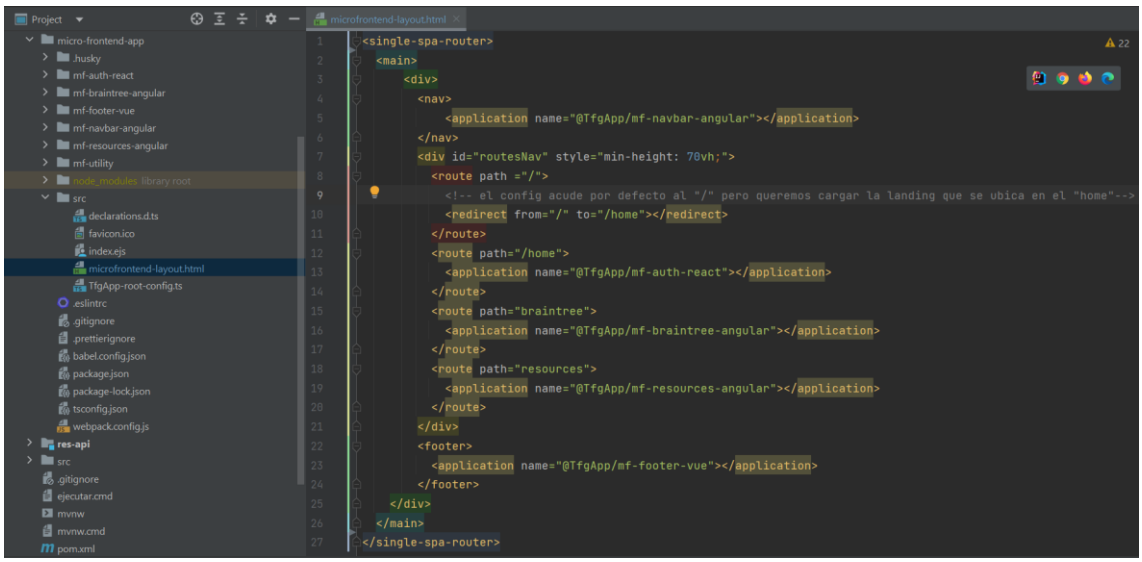

Ilustración 32: Representación del fichero "microfrontend-layout.html" ubicado en el microfrontend "micro-frontend-app" que actúa como nuestro Root Config en SingleSpa. En la ilustración podemos ver como se define el router y qué parcelas se renderizarán para cada ruta.

El *Router* (manejo de las rutas) en *SingleSpa* no es nada trivial cuando las *Parcelas* son aplicaciones enteras y tienen su propia navegación interna. Es decir, en la inmensa mayoría, por no decir en todos los tutoriales y documentación; la *Parcela* es un único componente, es un único documento *html* sin navegación interna.

El desafío propuesto en el presente TFG es mucho mayor pues hay un doble *routing*, el primero es la propia visualización de las P*arcelas*, y el *subrouting* es la propia navegación para cada una de las *Parcelas*.

La solución una vez se tiene, parece evidente, ahora bien, cabe destacar que este fue el segundo mayor problema que me encontré a la hora de realizar el TFG (siendo el primero el problema del *CORS* en el *Gateway*). Para dar con la solución al problema del doble *routing* necesité dos días enteros de literalmente búsqueda, lectura de documentación y, ensayo y error para lograr dar con la solución.

iii. Configuración de fichero ts

Por último, el fichero "TfgApp-root-config.ts" es quien realmente (entre las sombras) va a ser quien haga la configuración interna para que todo funcione.

En caso de implementar microfrontends con *Module Federation*, precisamente sería este fichero el que habría que modificar para registrar nuevos componentes.

Sin embargo, al utilizar *SingleSpa* nos podemos ahorrar esta parte pues él mismo ya registrará cada una de las aplicaciones definidas en nuestro fichero "index.ejs". Concretamente, esta sería su visualización:

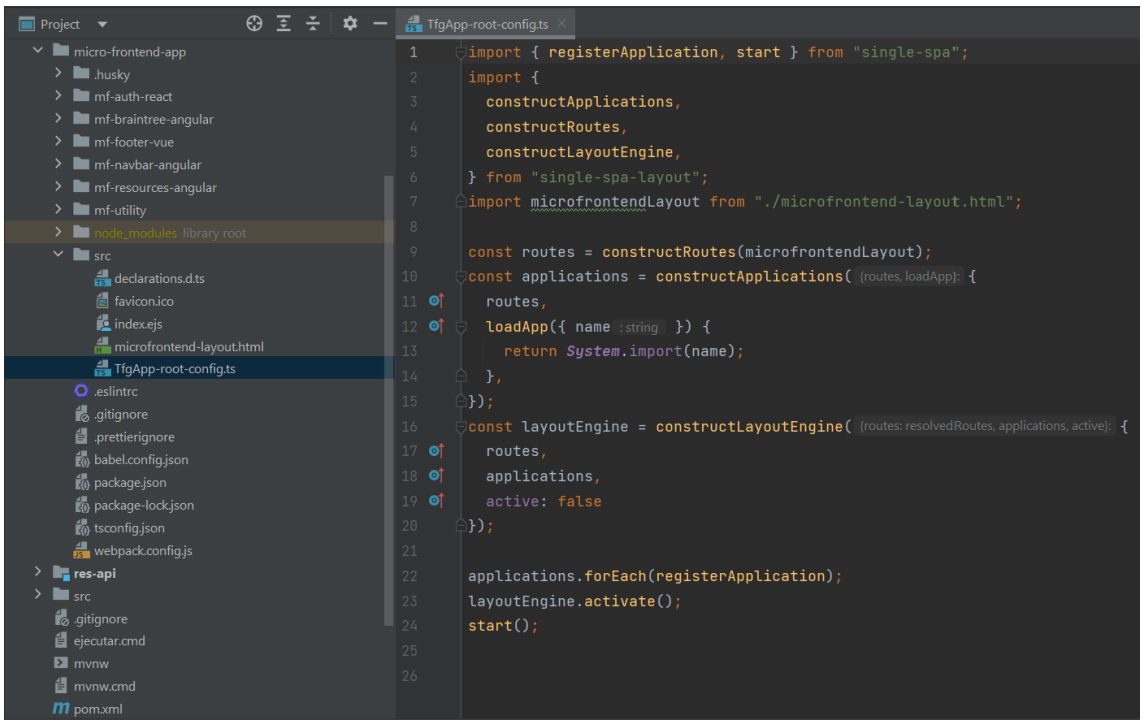

Ilustración 33: Representación del fichero "microfrontend-layout.html" ubicado en el microfrontend "micro-frontend-app" que actúa como nuestro Root Config en SingleSpa. En la ilustración podemos

Es importante mencionar que es fundamental importar aquí nuestro fichero "microfrontend-layout.html", pues será aquí donde se haga efectivo el *router* que habíamos definido allí.

iv. Funcionalidad y Objetivo

Teniendo en cuenta todo lo anteriormente expuesto, podemos concluir que la funcionalidad del *Root Config* es bastante compleja pues su objetivo será la de no sólo configurar cada una de las *Parcelas* y su disposición en la visualización de la página web, sino también la de tener un registro de todas y cada una de los microfrontends que se deben cargar (incluido el módulo *Utility*), además de actuar como *Gateway* de cara al exterior; pues todas y cada una de las peticiones recibidas en el *Backend* no vendrán de ningúna *Parcela* sino que todas partirán del *Root Config* ya que actúa como *Gateway* para el resto de *Parcelas* del *Frontend*.

v. Patrones, Características y tecnologías utilizadas

Finalmente, es importante tener en cuenta que para este tipo de microfrontend se suele utilizar un *framework* similar a *React* pues ambos se basan en la gestión y manejo a través de ficheros *js* (o *tsx* en este caso, pues lo hemos creado para que sea *TypeScript*).

En la próxima parada de nuestro tren, nos enfrentaremos ante una decisión muy importante, pues aprenderemos y deberemos decidir cuál de las diversas técnicas sobre cómo compartir la información en *SingleSpa* vamos a utilizar.

d. Compartir Datos en Utility Js

Anteriormente habíamos mencionado que en *SingleSpa* existen tres tipos de microfrontends: El *Root Config* (que es el que acabamos de explicar en la sección superior), las *Parcelas* (que eran nuestras propias aplicaciones de microfrontends), y un tercero que es el módulo *Utility*, que actuaba como librería trasversal.

En este apartado explicaremos este módulo de utilidad transversal (*Utility*) para solucionar el problema de la comunicación entre las distintas *Parcelas*.

i. Utility vs Cookies vs LocalStorage

Realmente existen diversas metodologías que podemos aplicar a *SingleSpa* para compartir información entre los distintos microfrontends. Cada una de ellas tiene sus ventajas y sus desventajas.

Por un lado, tenemos la alternativa clásica de utilizar el *LocalStorage* del propio navegador para poder compartir información a través de los diversos componentes de la página web. Esta alternativa es muy potente debido a que no sólo estamos compartiendo la información para cada microfrontend, sino también para cada uno de los componentes dentro de cada microfrontend pues puede ser accesible desde cualquier punto del *Frontend*. Pero, tenemos el problema de la persistencia y de la seguridad pues, a no ser que se cifren los datos, estos son visibles.

Por otro lado, tenemos la opción de almacenar los datos que queramos compartir en formato de *cookie* para el navegador, en lugar de utilizar el *LocalStorage*. Si bien es cierto que en una de las etapas del TFG se desarrolló completamente esta alternativa, la integración de compartir la información a través de *cookies* fue eliminada. El principal motivo de esto es que, dependiendo del navegador y de su propia configuración, la *cookie* podía actuar de una u otra forma puesto que era el propio navegador el que gestionaba su acceso. En otras palabras, dependiendo del navegador, algunos usuarios podrían utilizar la página web y otros no. Personalmente, aunque no descarto que se pueda implementar utilizando esta tecnología, por el momento, no poseo el conocimiento ni la experiencia suficiente para garantizar que a través de *cookies* funcione en la totalidad de las ocasiones.

Como último recurso tenemos el módulo *Utility* propio de *SingleSpa*, que precisamente está enfocado en este propósito pero que fue inicialmente descartado debido a que en realidad no era necesario. Esto es debido a que la única información que queríamos compartir era dentro de un mismo microservicio, no a través de varios microservicios; y, por tanto, únicamente con el almacenamiento en *LocalStorage* ya lográbamos tal propósito.

No obstante, tal como comentaré a continuación, me di cuenta de que en realidad sí debo notificar sobre el inicio de sesión a los demás microfrontends, por lo que el módulo *Utility* pasó a ser una necesidad y finalmente se ocupa de mantener dos observadores acerca del usuario actual activo.

ii. Funcionalidad y Objetivo

Supongamos que un microfrontend *Parcela-A* debe comunicarse o pasar un dato a la *Parcela-B* para que así ésta pueda actualizar su valor. El ejemplo práctico lo podemos encontrar en este TFG pues después de hacer el inicio de sesión en la *Parcela-A* (“mf-auth-

react") se debe ir a la *Parcela-R* ("mf-resources-angular"). Sin embargo, existe una tercera parcela que es la *Parcela-N* ("mf-navbar-angular").

Un problema que me he encontrado cuando desarrollaba mi TFG es que la *Navbar* puede mostrar o bien el botón de iniciar sesión y crear cuenta, o bien el botón de cerrar sesión, en caso de que la sesión ya esté iniciada. No obstante, donde se inicia la sesión es el la *Parcela-A*, que justo después de hacer el inicio de sesión se redirige a la *Parcela-N*. Por tanto, el desafío reside en cómo notificar a la *Parcela-R* que no tiene interacción ninguna en el momento de pasar de navegar desde la *Parcela-*A a la *Parcela-N.*

Si nos acordamos de la definición del archivo *html* del *Root Config*, la *Navbar* (*Parcela-N* o lo que es lo mismo "mf-navbar-angular") y el *Footer* ("mf-footer-angular") siempre se están ejecutando independientemente de la navegación del *rooter*.

La solución es implementar unos observadores, concretamente para la *Navbar* pues en mi caso el *Footer* no lo necesita. El valor de la variable observada será lo que actualice la *Parcela-A* antes de hacer efectivo el inicio de sesión y, por tanto, pasar a ejecutarse la *Parcela-R*. De esta forma, la *Parcela-N* se actualizará sola pues detectará el cambio de la variable en el observador y pasará de visualizar los botones de iniciar sesión y registrarse, a mostrar el botón de la cuenta del usuario y el respectivo de cerrar sesión.

Por tanto, y como podemos observar, el objetivo de este módulo *Utility* es desempeñar la función de una biblioteca transversal, como si de un gestor de mensajes se tratase, como por ejemplo el *RabbitMQ* que mencionábamos en la sección del *Backend*, cuando explicamos el funcionamiento del *Config Server*.

iii. Patrones, Características y tecnologías utilizadas

Es importante tener en cuenta que este tipo de microfrontend no puede utilizar ningún tipo de *framework* (*Angular*, *React*, *Vue*…) sino que deben ser ficheros *Javascript* (*js*) pues han de ser compatibles con cada uno de los distintos *frameworks* de las *Parcelas*. Es por este motivo por el que no se ha implementado *TypeScript* en este microfrontend.

iv. ApiCache vs State vs Custom Events

De entre todos los ficheros presentes en este microservicio de utilidad "mf-utility", realmente sólo nos debemos centrar en el fichero principal de este microservicio, pues solamente debemos modificar un único fichero que es el ubicado en la carpeta "src".

En este fichero podemos optar por implementar un formato de *ApiCache* que se caracteriza por añadir un mapa (estructura en formato llave, valor) y una función que modifique los valores que tiene ese mapa y sea capaz de añadir nuevas entradas a dicho mapa.

En su lugar, también podemos definir una variable que actúe como observador, por lo que únicamente cada una de las *Parcelas* deberá suscribirse a esa variable (observador) y así recibirán una notificación automáticamente ante cualquier cambio.

Otra opción para compartir información en el módulo *Utility*, consistiría en implementar una estrategia muy similar a *Module Federation* y crear y configurar eventos personalizados que se activen después de cumplirse una determinada condición bajo unos parámetros concretos.

v. Configuración Interna del Utility

Tal como podremos observar en el siguiente apartado cuando introduzcamos el microfrontend de la *Navbar*, la metodología escogida para implementar en este módulo *Utility* es la segunda opción. La creación de una variable que actúa como observador, a la que cada componente de cada *Parcela* debe suscribirse para poder ser notificado automáticamente de cualquier cambio que otro componente, de este mismo o de distinto microfrontend pueda hacer.

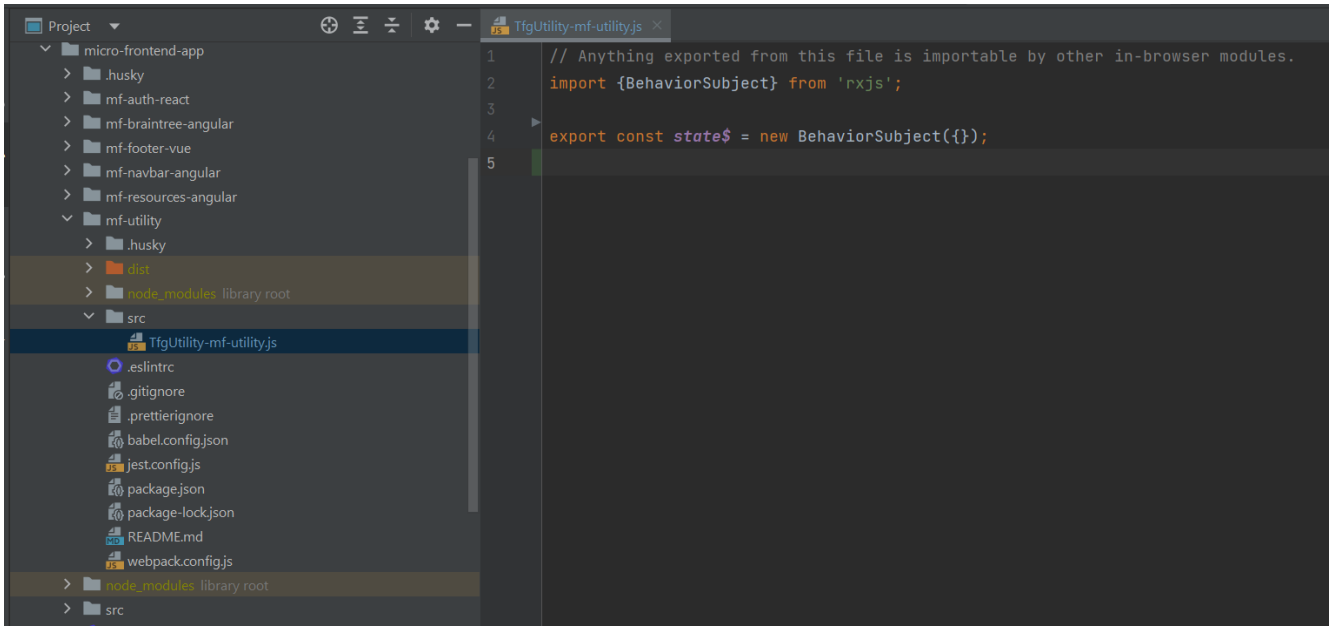

Ilustración 34: Representación del fichero "TfgUtility-mf-utility.js" en el microfrontend "mf-utility" donde puede visualizarse como estamos exportando y definiendo la variable compartida que actuará como observador entre diversos microfrontends.

Tal como podemos observar, tan sólo debemos exportar una variable que actúe de observador al ser de la clase *BehaviourSubject* (que pertenece a la biblioteca "rxjs", es decir *ReactiveExtensionsForJavaScript*). Por lo que el siguiente paso será exportar las librerías necesarias en este microfrontend, e importarlas en aquellos que queramos utilizarlo.

En realidad, a fines prácticos se están exportando dos observadores distintos. No obstante, para aligerar la complejidad del contenido explicaremos el concepto con uno solo. Ambos observadores tienen la misma funcionalidad, pero se ha decidido separarlos pues tienen ámbitos distintos al utilizarse para finalidades distintas.

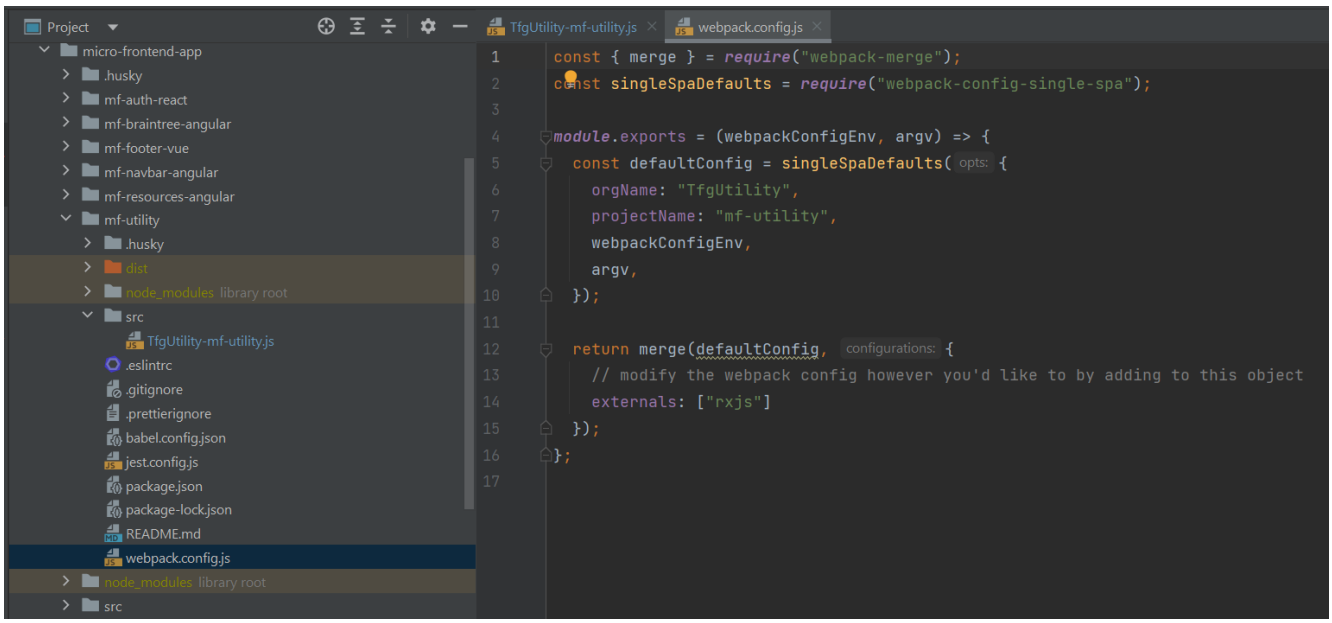

Ilustración 35: Representación del fichero "webpack.config.js" del microfrontend "mf-utility" donde se visualiza que en el apartado "externals" estamos exportando los elementos (librerías) necesarias para utilizar nuestra variable.

Como se visualiza en la imagen, el otro fichero que debemos modificar en el propio microservicio *Utility* es el "webpack.config.js" pues debemos añadir el campo de *externals* haciendo referencia al *import* que habíamos hecho anteriormente del paquete "rxjs".

vi. Configuración Externa en una Parcela React

Por tanto, ahora únicamente deberemos importar estos paquetes y variables en los microfrontends que deseemos utilizarlos. No obstante, hay que tener en cuenta que dependiendo del *framework* utilizado en cada una de las *Parcelas*, si bien el procedimiento es el mismo y los pasos son exactamente los mismos, la forma en la que se realizará variará únicamente por el formato a la hora de tocar el fichero de configuración del *webpack*.

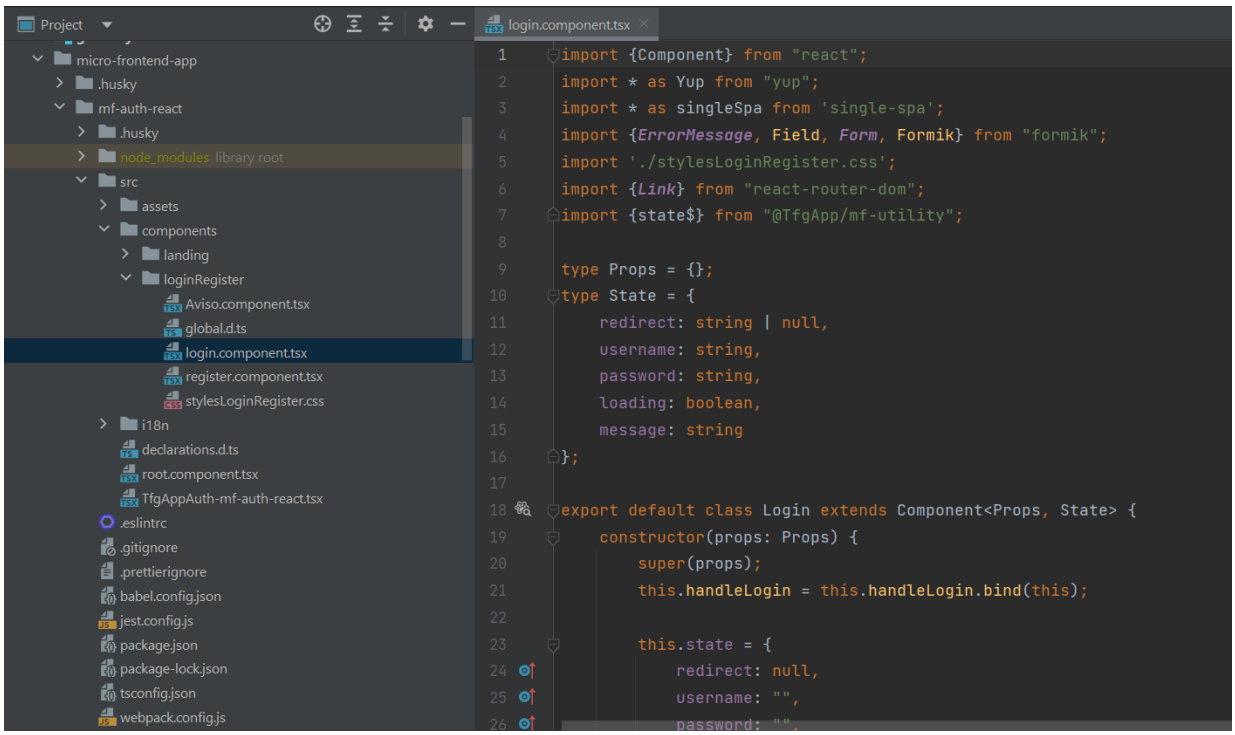

Ilustración 36: Representación del fichero "login.config.tsx" del microservicio "mf-auth-react" donde se visualiza que estamos importando nuestra variable observadora del microfrontend "mf-utility".

La imagen anterior podemos observar que hemos importado el microfrontend "mf-utility" y la variable "state$" que habiamos definido en él.

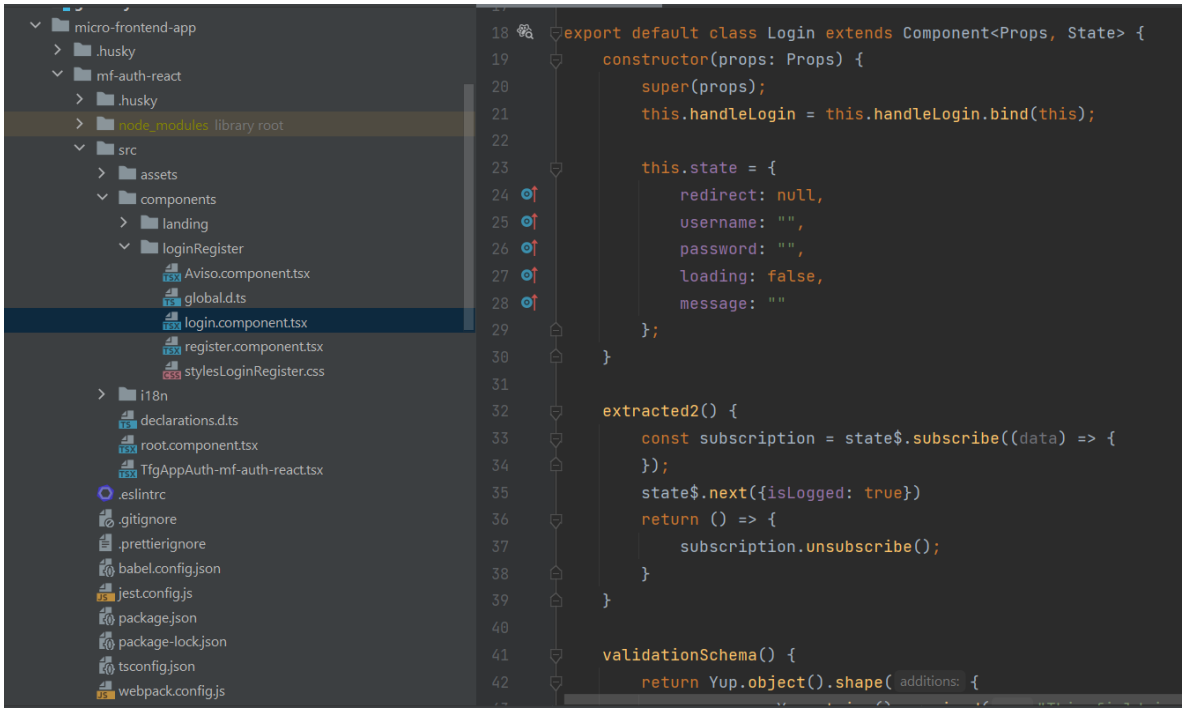

Ilustración 37: Representación del mismo fichero "login.config.tsx" del microservicio "mf-auth-react" donde se visualiza cómo estamos utilizandola variable observadora del microfrontend "mf-utility".

Y tal como mencionábamos, en este caso el componente del login lo que hará será suscrbirse para actualizar el valor de esta variable observador y a continuación desuscribirse, pues al finalizar esta funcion se llamará a la pagina principal del microfrontend de resources (“mf-resources-angular”).

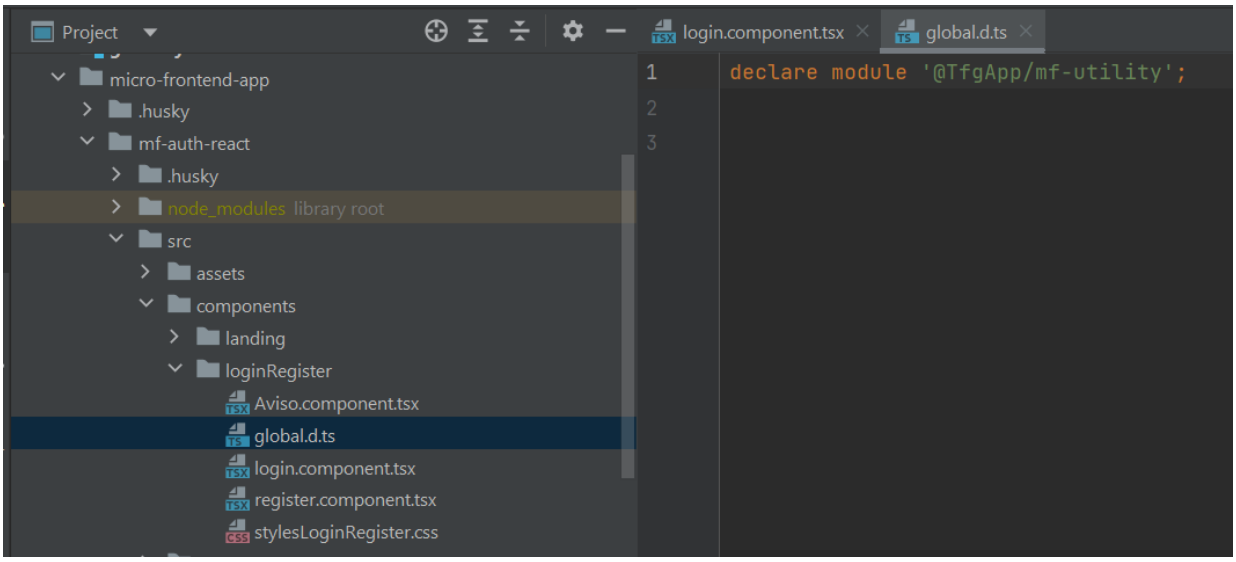

Ilustración 38: Representación del fichero “global.d.ts” del microservicio “mf-auth-react” donde se visualiza cómo estamos declarando el módulo que equivale a nuestro microfrontend “mf-utility”.

Así mismo, si se intenta implementar el código de arriba nos encontraremos ante el problema de que el *import* ni reconoce ni encuentra este módulo *Utility*. Por tanto, es necesario definir la declaración del módulo de la forma que puede observarse en la imagen anterior.

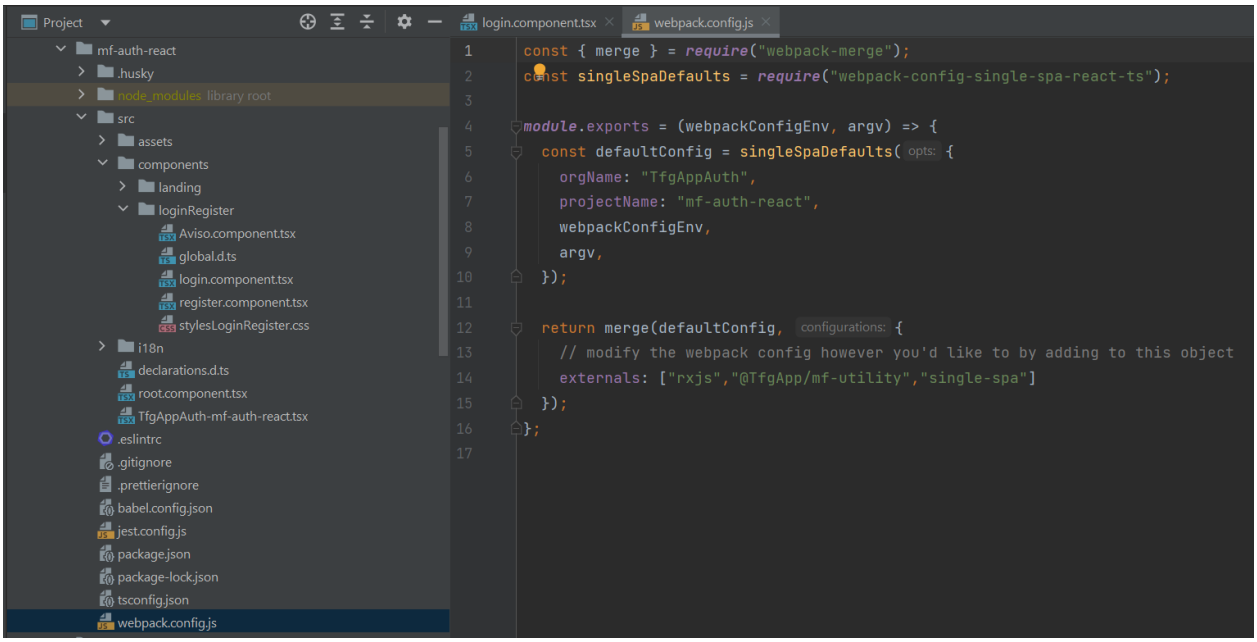

Ilustración 39: Representación del fichero “webpack.config.js” del microservicio “mf-auth-react” donde se visualiza cómo estamos declarando en el apartado de “expyternals” los distintos módulos que necesitamos para poder utilizar la variable observadora de nuestro microfrontend “mf-utility”.

Finalmente, ya sólo deberemos modificar el fichero de “webpack.config.js” en donde deberemos añadir la sección de *externals*, añadiendo evidentemente el microfrontend “mf-utility”.

vii. Configuración Externa en una Parcela Angular

Por otro lado, si en lugar de querer añadir el módulo *Utility* en una *Parcela* que use el *framework React*, quisiéramos añadirlo en una *Parcela* en *Angular* deberemos modificar un par de ficheros.

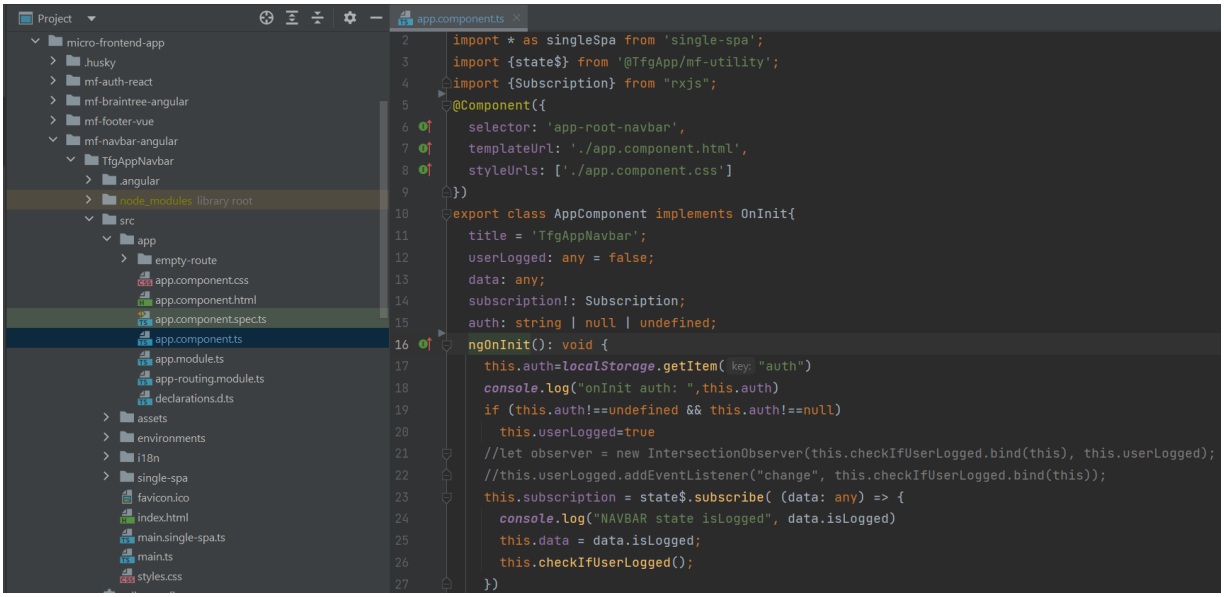

Ilustración 40: Ilustración equivalente a las ilustraciones 36 y 37 pero en este caso en Angular. Representación de cómo importar y utilizar la variable observadora del microfrontend Utility en el microfrontend “mf-navbar-angular” en su fichero “app.component.ts”.

La primera diferencia la encontramos en el fichero donde queremos utilizar esta varibale, pues además deberemos importar y crear una variable auxiliar del tipo *Subscription*. En esta variable será donde nos guardaremos la susbcripción de la variable que estamos observando.

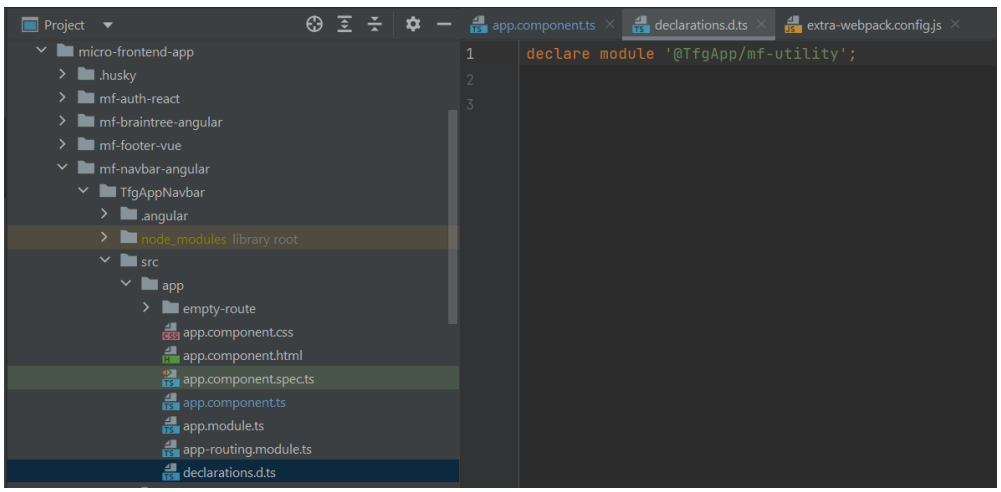

Ilustración 41: Ilustración equivalente a la ilustración 38 pero en este caso en Angular. Representación de cómo declarar el microfronetnd “mf-utility” como un módulo dentro del microfrontend “mf-navbar-angular” en el fichero “declarations.d.ts”.

De igual forma a como lo haciamos en *React*, para que nos reconozca el microfrontend “mf-utility” también será necesario crear un fichero donde lo declaremos. De esta manera y exclusivamente tras efectuar esto, el *import* que hemos realizado anteriormente nos funcionará.

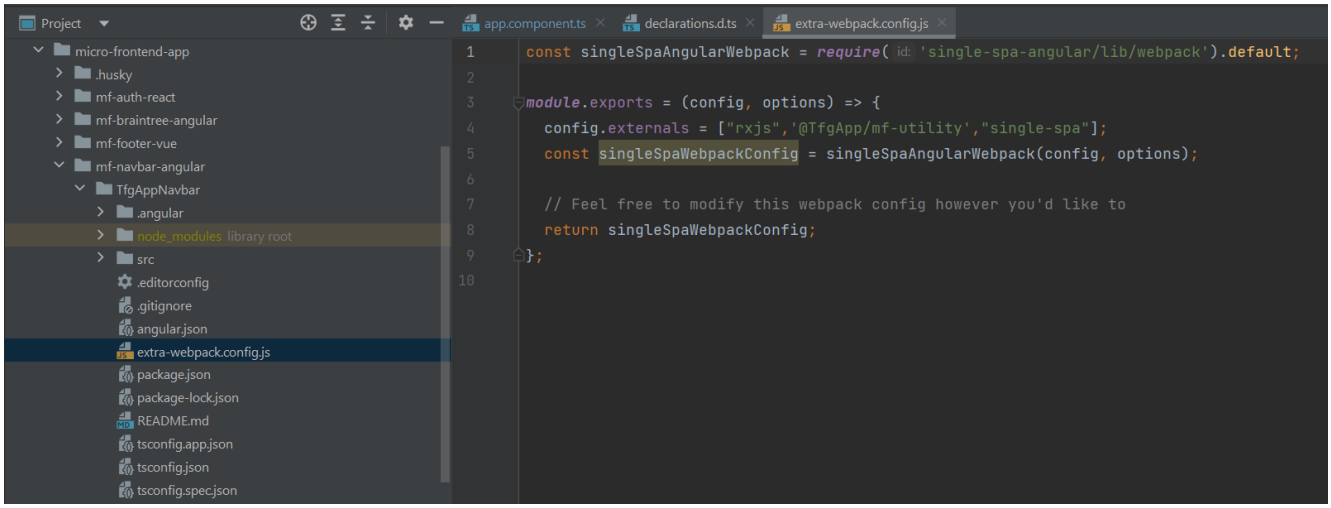

Ilustración 42: Representación equivalente a la ilustración 39 pero en este caso en Angular; donde podemos visualizar como se está definiendo en el archivo “extra-webpack.config.js” en el campo “externals” para poder utilizar dichos módulos dentro del microfrontend “mf-navbar-angular”.

Finalmente, también debemos añadir la sección de *externals* en el “extra-webpack.config.js” de la *Parcela* en *Angular*. De forma equivalente no sólo deberemos añadir el microfrontend *Utility*, sino también aquellos otros que necesitemos como es el caso de “rxjs” y “single-spa”.

Sin lugar a dudas hemos recorrido un largo viaje parando en cada una de las estaciones para aprender elementos nuevos que han sido incorporados satisfactoriamente en este trabajo final de grado. En nuestra decimonovena parada, dejaremos a un lado la configuración y nos adentraremos en las *Parcelas*. Concretamente, empezaremos por visitar el microfrontend “mf-navbar-angular”.

e. Navbar Angular

Uno de los microfrontends más importantes y al mismo tiempo más infravalorados precisamente es la *Navbar*, ya que en esta se gestiona todo el acceso a las diferentes secciones de nuestra página web y funcionalidades que ofrecemos.

En un *Frontend* monolítico únicamente deberíamos tener en cuenta la disposición de rutas y componentes definidos en el propio *router* y cargar aquella ruta que nos interese. Pero, al utilizar *SingleSpa* esta tarea es bastante diferente, pues no basta con conocer las rutas propias de ese microfrontend sino de las diferentes *Parcelas*.

i. Funcionalidad y Objetivo

Es por tanto que el objetivo de este microservicio realmente es la de redirigir hacia el componente correspondiente de cada una de las otras *Parcelas*. Para ello, no es posible

utilizar un *router link*, ni tampoco es nada recomendable utilizar un *link* o un *href* convencionales. Esto se debe precisamente a la diferencia entre *SPA* vs *MPA* que comentamos al inicio de este capítulo donde explicamos el *Frontend* además de ser microfrontends.

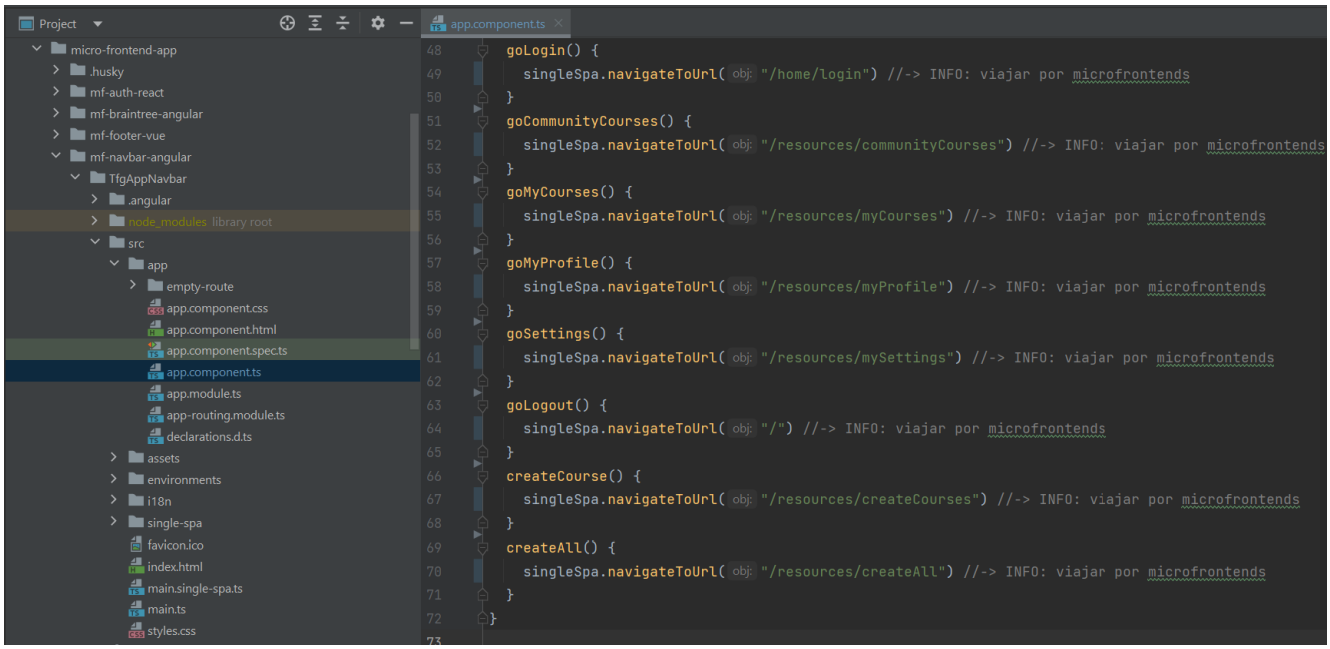

Ilustración 43: Representación del fichero "app.component.ts" del microfrontend "mf-navbar-angular" en donde queda reflejada la comunicación y redireccionamiento SPA entre diversos microfrontends.

Si nos fijamos en la imagen, para la navegación por la url estamos utilizando la función *navigateToUrl* que ya viene integrada dentro de *SingleSpa*. Esta es la metodología a seguir si queremos cambiar de url, pues únicamente deberemos cambiar el *path* correspondiente al *router* de *SingleSpa* (definido en el *Root Config*). El primer elemento del *path* diferenciará entre un microfrontend y otro, mientras que el segundo elemento corresponderá al *routing* interno de cada uno de los *microfrontends*.

ii. Patrones, Características y tecnologías utilizadas

Dejando de lado el evidente patrón de diseño de navegación al implementar una *Navbar*; la característica y tecnología utilizadas que cabe resaltar es la navegación entre microfrontends. Pues esta debe hacerse utilizando la función que nos proporciona internamente *SingleSpa* para garantizar una correcta navegación *SPA* entre los diversos microfrontends.

Seguidamente, haremos una breve parada para analizar el sencillo microfrontend encargado del *Foooter*.

f. Footer Vue

En contraposición a la *Navbar* tenemos el microfrontend que se ocupa del *Footer*, el cual usará el *framework* de *Vue* y será el "mf-footer-angular".

i. Funcionalidad y Objetivo

Este microfrontend en realidad es bastante sencillo y simple, pues tan sólo tiene como finalidad la de proveer a la página web de un *Footer*. Por tanto, ese será su objetivo y no será necesaria ningún tipo de interacción con el módulo *Utility* dado que el comportamiento de este microservicio no se va a ver afectado dependiendo de si el usuario ha o no iniciado sesión. De la misma forma a como lo hemos hecho en los *frameworks* de *React* y *Angular*, si quisiéramos añadir alguna diferencia entre ambos estados (si el usuario no ha iniciado sesión o, sí que lo ha hecho) entonces sí deberíamos añadirlo de forma equivalente a como lo hacen los otros dos microservicios “mf-auth-react” y “mf-navbar-angular”.

iii. Patrones, Características y tecnologías utilizadas

Dejando de lado el evidente patrón de diseño de navegación al implementar un *Footer*; de forma equivalente a la *Navbar*, en el *Footer* estaremos trabajando con un único componente “Footer.vue” y el *framework* que utilizaremos será el de *Vue*, tal como ya hemos mencionado anteriormente.

Llegando a nuestra vigésima primera estación, podemos atisbar en el horizonte que entramos en los últimos tres microfrontends esenciales: *Auth*, *Braintree* y *Resources*. Concretamente, en esta etapa nos detendremos a analizar el microfrontend “mf-auth-react”.

g. Auth-React

En este apartado vamos a explicar el microfrontend encargado de la gestión de la *landing page*, el registro de nuevos usuarios y el propio *login*. Este microfrontend “mf-auth-react” complementa al microservicio “auth-api” en el Backend encargado de la gestión de los usuarios.

i. Funcionalidad y Objetivo

La funcionalidad y objetivo de este microfrontend es darle la bienvenida al usuario con la *landing-page* e introducirlo sobre de qué trata esta página web. Igualmente, se ocupará de ofrecerle la interfaz para que pueda crear una nueva cuenta con sus respectivas comprobaciones y de notificarle que debe acudir a su correo electrónico para la verificación de su cuenta recién creada.

Por último, este microfrontend también tendrá como objetivo la gestión del acceso de los usuarios al incorporar la pantalla del *login*. Pues será quien llame a los procesos asociados a la gestión y verificación del token que explicamos en la sección del *Backend*.

ii. Patrones, Características y tecnologías utilizadas

Para este microfrontend se ha creado una *Parcela* utilizando *TypeScript* además del *framework* de *React*. Se ha añadido un *routing* interno para poder navegar por los distintos componentes dentro de este microfrontend.

Además, se ha incorporado e integrado y está totalmente operativo el sistema de traducciones en este microservicio. Pero se ha dejado comentado y sin utilizarse pues para poder implementar esta mecánica debía implementarse en todos los microfrontends y *frameworks* utilizados.

Como último aspecto a destacar, tal como hemos mencionado cuando hemos introducido el microfrontend *Utility*, hemos comentado que el módulo *Utility* está integrado en esta *Parcela* encargada de los usuarios, debido a que debe notificar del cambio a la *Navbar* para que se actualice y permita ver los elementos por el que el usuario puede navegar.

Continuando con nuestro itinerario, llegaremos a la próxima parada. En ella podremos acabar de observar cómo funcionan las pasarelas de pago, pues nos adentramos en los dominios del microfrontend “mf-braintree-angular”.

h. Braintree-Angular

A continuación, explicaremos el microservicio que se encarga de la pasarela de pago en el *Frontend*. De forma equivalente al microservicio en el *Backend*, el microfrontend es muy pequeño pues únicamente tiene como objetivo realizar la transacción de un importe que corresponderá a los cursos que haya comprado el usuario.

i. Funcionalidad y Objetivo

Aunque su objetivo es bien sencillo (pues debe realizar la transacción y llamar al correspondiente microfrontend y de esta forma actualizar los valores correspondientes en el microservicio “res-api” del *Backend*); su funcionalidad y su implementación no es tan trivial como pudiera parecer.

Esto se debe a que estamos comunicando datos entre distintos microfrontends. Para ser exactos, esta *Parcela* deberá recibir una lista con los cursos que desea comprar el usuario, al mismo tiempo que recibe qué usuario es el que quiere realizar dicha transacción. Es por este motivo por el que es vital que este microfrontend se suscriba a otra variable de la *Parcela Utility* pues será por allí por donde reciba el disparador en el observador que le informe que los datos están listos en el *LocalStorage*.

ii. Patrones, Características y tecnologías utilizadas

Para este microfrontend se ha creado una *Parcela* utilizando *TypeScript* además del *framework* de *Angular*. Se ha añadido un *routing* interno para poder navegar por los distintos componentes dentro de este microfrontend (el correspondiente a realizar el pago, y el correspondiente a la verificación y *feedback* una vez se ha realizado correctamente la transacción).

Además, se ha incorporado e integrado el sistema de traducciones en este microservicio, pero se ha dejado comentado y sin utilizarse pues no se ha logrado implementar de una forma limpia el acceso a los ficheros locales *jsons* que almacenan el contenido de las traducciones. A pesar de que se puede implementar la traducción, hacerlo actualmente supondría una gran cantidad de condicionales para saber cuál de las traducciones mostrar. Es por ello por lo que se ha preferido dejar esta mecánica más apartada para centrarse en desarrollar y finalizar completamente otras que aportan más valor al proyecto.

Como último detalle destacable, decir que para la pasarela de pago se ha usado “ngx-braintree” y que para poder añadir este módulo es necesario añadir el parámetro “--legacy-per-deps” cuando se instalan las dependencias de los “node modules”.

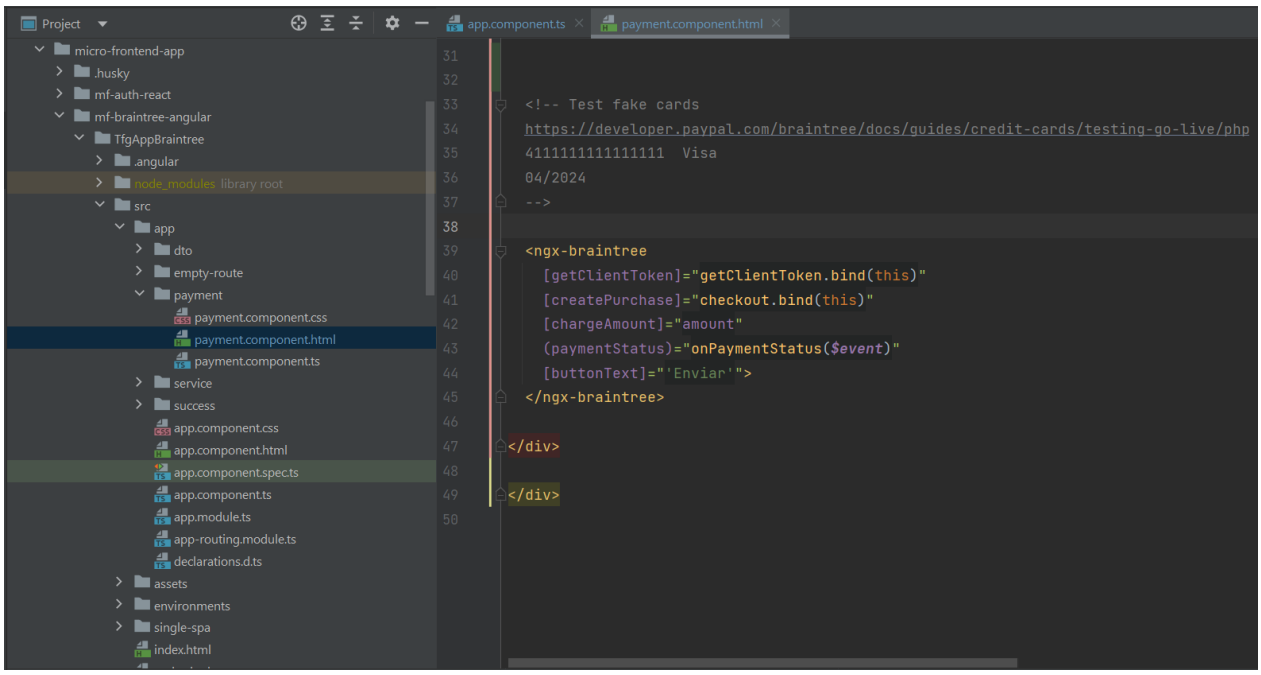

Ilustración 44: Representación del fichero "payment.component.html" del microfrontend "mf-braintree-angular" en el que se puede observar el uso de la pasarela de pago al utilizar las etiquetas "<ngx-braintree>".

Tal como podemos observar en la imagen anterior, la pasarela de pago a niveles practicos no deja de ser la gestión del flujo de información de ese bloque de etiquetas.

El camino de nuestro viaje, sin duda alguna, ha sido largo. Parada tras parada, etapa tras etapa hemos ido adquiriendo nuevos conocimientos y metodologías que aplicar para cumplir el objetivo final de nuestra travesía: que la página web tenga los elementos necesarios para poder ser implementada por una empresa real. En esta última etapa del mundo del *Frontend*, llegamos al último microservicio: la *Parcela* "mf-resources-angular".

i. Resources-Angular

Finalmente, explicaremos el último de los microfrontends, el que tendrá relación con el microservicio "res-api" encargado de la gestión de toda la información relativa a los cursos. Este microfrontend, "mf-resources-angular", será el encargado de ofrecer al usuario la interacción principal con toda la página web. Pese a que se podría haber disgregado este microfrontend en otros más pequeños, se ha optado por seguir la misma metodología que en el *Backend* al agrupar todo lo relacionado con los cursos.

i. Funcionalidad y Objetivo

En consecuencia, la funcionalidad de esta *Parcela* será la mayor y principal de todas, pues es la que internamente debe mostrar la absoluta mayoría de la información que se gestiona en la base de datos. Por tanto, su objetivo principal será proporcionar al usuario el acceso a la información solicitada mediante operaciones *CRUD* que recogerá el microservicio correspondiente.

No es de extrañar que este otro microfrontend también deba incorporar el módulo *Utility*, pues de la misma forma los anteriores también deberán suscribirse a esa variable

para poder estar al corriente de los cambios y actualizaciones que otros microfrontends puedan hacer.

ii. Patrones, Características y tecnologías utilizadas

Para este microfrontend se ha creado una *Parcela* utilizando *TypeScript* además del *framework* de *Angular*. Se ha añadido un *routing* interno para poder navegar por los distintos componentes dentro de este microfrontend (el correspondiente a todo lo relacionado con los cursos) y será la *Parcela* que sea llamada principalmente desde la *Navbar*.

Finalmente, saldremos del mundo del *Frontend, para* adentrándonos en el mundo del despliegue web (*deploy*).

10. Deploy en AWS

En este apartado explicaremos las distintas configuraciones de entorno de trabajo local y el de producción.

i. Localhost vs Producción

Tal como hemos mencionado anteriormente en la configuración del *Root Config*, existen dos contextos en los que cualquier web puede funcionar.

El primero es el contexto local (*Localhost*), en el que cualquier página web empieza a ser desarrollada. No obstante, en ese contexto la web únicamente existe en tu propio sistema, por lo que es necesario lanzarla al exterior, hacerla accesible desde internet para que otros usuarios puedan interactuar con ella.

El segundo contexto precisamente trata sobre ello. En el segundo modo, el modo de P*roducción*, precisamente estás haciendo que tu página web sea accesible desde internet.

Para hacer visible tu aplicación web en internet existen diferentes plataformas que realizan estas funciones. Concretamente nos centraremos en las tres que fueron pensadas en la realización de este proyecto final de grado.

ii. Heroku vs Hostinger vs AWS

La primera alternativa siempre fue *Heroku*. Con esta plataforma gratuita me encontraba muy cómodo pues la conocía muy bien ya que había hecho el *deploy* de mas de cinco aplicaciones en ella por lo que sin duda alguna era la primera y la única opción nada más comenzar este proyecto.

Sin embargo, lamentablemente *Heroku* anunció hace aproximadamente un año que ya no iba a seguir ofreciendo sus recursos de forma gratuita y que por tanto iba a cerrar todas aquellas aplicaciones web que estuvieran desplegadas en esa plataforma.

Es por ese motivo por el que me vi obligado a cambiar el plan inicial y emprendí una búsqueda para encontrar alternativas.

En mi búsqueda me encontré con decenas de posibles plataformas para hacer el despliegue de la web en internet. Sin embargo, por sus características me centré únicamente en dos: *Hostinger* y “Amazon Web Services” (*AWS)*.

Cabe destacar que no es finalidad de este trabajo enumerar y describir en qué consiste cada una de ellas pues son herramientas sumamente complejas y exageradamente amplias y profundas que difieren del objetivo de este trabajo. Por ello, únicamente nos centraremos en lo que a este proyecto les afecta.

La opción más segura sin duda alguna era la de incorporar *AWS*. Sin embargo, como me gusta complicarme la vida y afrontar los retos por muy difíciles que sean, escogeremos la opción de hacerlo en *Hostinger*.

La idea de hacer el *deploy* utilizando *Hostinger* se centraba en utilizar los *Kubernetes* de *Docker*. A modo breve de resumen pues no vamos a entrar en detalles como anteriormente hemos mencionado, *Docker* sirve para desplegar aplicaciones dentro de contenedores de *Software*. Y, precisamente los *Kubernetes* sirven para automatizar ese despliegue además de facilitar y ayudar en su escala y el propio manejo de esos contenedores. En otras palabras, en *Docker* podemos desplegar nuestra aplicación y los *Kubernetes* facilitan el uso de *Docker* para ello.

No obstante, después de todo un día configurando, informándome y viendo tutoriales, solucionando problemas y errores me di cuenta de que la alternativa de utilizar *Docker* con *Kubernetes* en *Hostinger* era literalmente imposible independientemente de cuanto pudiera esforzarme yo o los errores que pudiera corregir por mi parte. Esto se debe a que después de hablar con soporte técnico y tras varias pruebas y debate con ellos, llegamos a la conclusión de que *Hostinger* no permite (por su propia construcción y configuración interna de la plataforma) el uso de *Kubernetes* en *Docker.*

Por ello, finalmente decidí hacer el *deploy* en AWS, pues además contaba con un plan gratuito de un año que me permitía poder desplegar esta página web.

Como en la absoluta mayoría de los elementos introducidos e integrados en este trabajo final de grado, *AWS* no fue una excepción. Pues partía de un conocimiento previo que era nulo, por lo que debía informarme y documentarme sobre cómo realizar el despliegue web utilizando esa plataforma. Si bien es cierto que cuando realicé prácticas de empresa, mi empresa utilizaba *AWS*, toda su gestión e integración para mi fueron completamente transparentes por lo que carecía de cualquier experiencia previa.

Después de lo que equivaldrían a más de doce horas buscando información y documentándome sobre *AWS*, muy a mi pesar, no he podido desplegar la página web en internet para que sea accesible a cualquiera.

Esto se debe a mi propio desconocimiento sobre la burocracia en la adquisición de un *dominio web*. Un *dominio web* no es más que tu localización en internet. Es decir, la url por la que se accede a tu página web, eliminando la primera parte que caracteriza al protocolo. “http://” y “https://” son ejemplos de esa parte del protocolo presentes en la url. Por tanto, poniendo el ejemplo de Google: “https://google.com” por ejemplo, el dominio web de Google sería “google.com”.

Este dominio web debe comprarse y puede tardar hasta tres días en ser otorgado. Además, hemos de tener en cuenta que tras la creación de tu cuenta en *AWS* debe pasar un día completo para poder ser activada y así utilizar sus recursos.

Sumando ese tiempo, y teniendo en cuenta que eran cinco los días que tenía reservados para realizar el *deploy* en este gran viaje que ha sido realizar este trabajo final de

grado; podemos darnos cuenta de que esos días fueron consumidos independientemente de mi interacción, pues fue cuestión de burocracia.

Algunos podrían pensar llegamos a este puto que quizás haber destinado esos días antes y, por tanto, haberme encontrado el problema antes, me hubiera otorgado de un mayor tiempo de reacción y de respuesta. En caso de yo haber sido consciente entonces sin lugar a duda habría optado por esa alternativa. Sin embargo, nada más lejos de la realidad, pues desconocía absolutamente el problema y no era consciente ni tampoco me lo podía imaginar. Esto se debe a que estoy acostumbrado a que el *dominio web* te lo proporcione la propia plataforma (como es el caso de *Heroku* en el que el propio dominio lo crea el automáticamente y en cuestión de segundos). No obstante, en *AWS* el *dominio web* debes proporcionarlo tu y de antemano.

Con todo ello, y muy a mi pesar, no sólo por la ilusión de poder tener desplegado en internet un proyecto que para mí significa tanto como lo es este, sino porque además este es el único objetivo secundario que no he logrado alcanzar. Y no precisamente por no saber hacerlo, sino por desconocimiento previo mío y la burocracia de la tramitación.

No obstante, no haber podido realizar el *deploy* no significa que se heche a perder todo el trabajo que hay detrás. Pues no olvidemos que existen dos entornos: *localhost* y *producción*.

Con todo ello, estamos adentrándonos en el final de esta aventura. Después de haber pasado por cada una de las paradas, llegamos finalmente a las últimas etapas de nuestro viaje.

11. Foto y esquema final de conocimientos obtenidos

Por un lado, tras la integración de todos los elementos mencionados en este trabajo final de grado, obtenemos el siguiente diagrama.

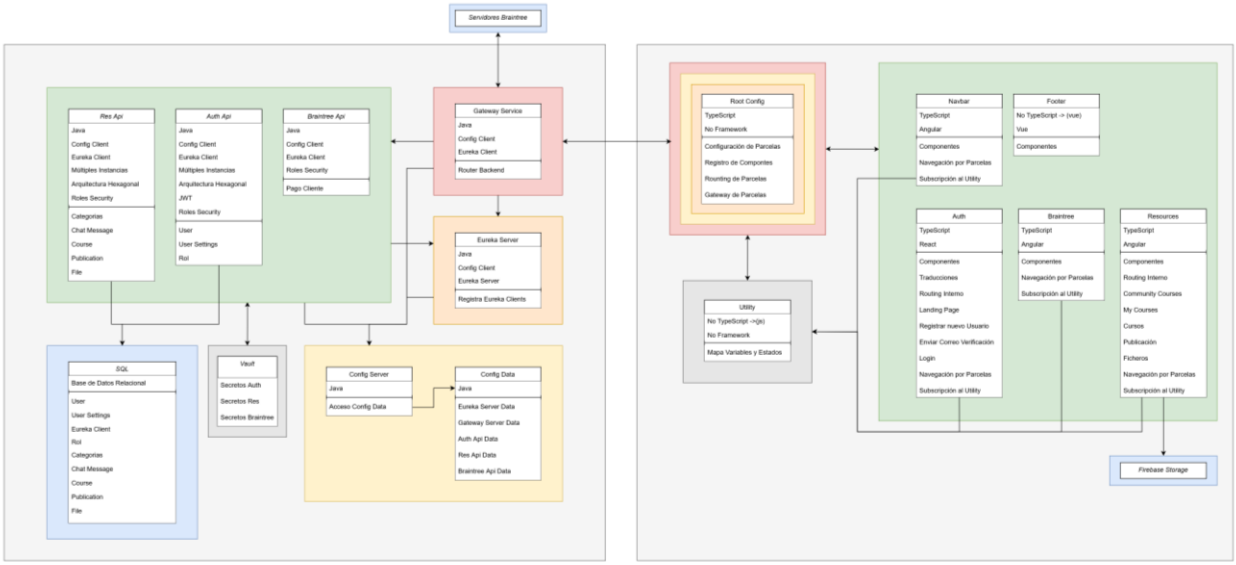

Ilustración 45: Representación de la foto final de la estructuración del código implementado con todos los microservicios, microfrontends y elementos utilizados. La sección izquierda corresponde al Backend mientras que la derecha hace referencia al Frontend. Esta misma imagen es la aportada y explicada en el Anexo1.

Tal como podemos observar a raíz de la ilustración anterior, podemos afirmar que hemos cumplido nuestro objetivo principal que era otorgar de la infraestructura necesaria a una página web para que pudiera ser utilizada por una empresa real. Además, a raíz de los conocimientos obtenidos, un nuevo mundo se abre ante nuestros ojos, pues los microfrontends son el futuro.

Por otro lado, se han dominado y se han aprendido muchos conceptos, características e implementaciones nuevas, así como se ha obtenido un amplio conocimiento sobre los microfrontends y sobre *SingleSpa*.

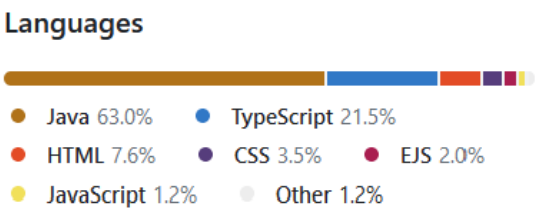

Ilustración 46: Representación de los lenguajes de programación utilizados para desarrollar este TFG. En la figura podemos observar cómo la mayor parte del código representa al Backend que esta realizado todo en Java (63%). Mientras que el Frontend será la suma de todos los otros lenguajes (37%).

A continuación (en lo que es la vigésima sexta y antepenúltima etapa de nuestro viaje), mencionaremos las posibles mejoras que podríamos implementar en este proyecto.

12. Límites y posibles mejoras

A lo largo de esta memoria mientras introducíamos y desarrollábamos varios conceptos, hemos mencionado la posibilidad de poder ser implementados de otra forma o la implementación adicional de otras características que puedan complementar e integrarse en este TFG.

No obstante, antes de exponer las posibles mejoras que se pueden hacer a la página web, cabe destacar que mi TFG no tiene límites. Su potencial no alberga ningún límite (dentro del contexto de una página web). El único límite soy yo mismo y mi propia imaginación y conocimiento, pero el potencial es infinito. Esto se debe precisamente a que, como dijimos desde un principio, no nos centramos en qué ofrece la página web, sino que precisamente nos hemos centrado en cómo lo está ofreciendo. Por ello, por las diversas tecnologías, patrones y características implementadas, no tiene límite pues su escalabilidad y máxima flexibilidad hacen que pueda añadirse cualquier elemento que se desee (sobre todo en aquellos que disponen de *Arquitectura Hexagonal*). Adicionalmente, para querer implementar cualquier característica nueva, únicamente será necesario crear un nuevo microservicio y un nuevo microfrontend con las nuevas características que se quieran implementar, pues son absolutamente independientes los unos de los otros.

A continuación, detallaremos los aspectos más relevantes que sugiero como posibles mejoras a implementar:

En primer lugar, evidentemente sería desplegar este proyecto de página web en internet utilizando *AWS,* por ejemplo. Por el lado del *Backend* realmente existe mucha documentación sobre cómo desplegar microservicios en *AWS*, además que por el lado del *Frontend* y los microfrontends utilizando *SingleSpa*, si bien es cierto que la documentación es menor, realmente con los ejemplos proporcionados en la propia documentación queda resuelta toda duda que pueda surgir.

La segunda posible mejora podría ser acabar de configurar las traducciones para el caso de los *Frameworks* de *Angular* y *Vue* (recordemos que para *React* ya está implementado y funciona perfectamente). Implementar un sistema multilingüe sin duda alguna es necesario para cualquier página web profesional, por lo que sería una gran mejora.

Cómo tercera sugerencia se podría intentar optimizar y hacer más escalable la gestión de los roles y el *Granthed Authorities* en caso de añadir una gran cantidad de roles, además de integrarse un sistema de cola de mensajes *MQ*. Si por el contrario seguimos trabajando con un número reducido de roles, esta tercera mejora no sería del todo necesaria.

Una cuarta mejora podría ser en la capa de *Dominio* de ambos microservicios “auth-api” y “res-api” que utilizan *Arquitectura Hexagonal*. Se podía mejorar esa parte añadiendo en el *Dominio* muchas más comprobaciones (para cualquier elemento y campo).

En el quinto lugar, se podrían mejorar los aspectos visuales de los microfrontends. Aunque, si bien es cierto que en términos generales están muy cuidados, en absoluto me considero un experto respecto al diseño. Dado que, en lo personal, soy más funcional que estético.

En sexto lugar y con el fin de aportar variedad a los lenguajes de programación utilizados en los microservicios en el *Backend*, propongo la mejora de implementar un recomendador de cursos y usuarios utilizando *Python* y una base de datos *Redis*. Tal como haría la plataforma de *Netflix* o *Amazon* que recomienda un catálogo de recursos a sus usuarios según sus visitas, calificaciones… Además, podría ampliarse la definición de las entidades "curso" y "publicación" incorporando un sistema de visitas, "likes", tendencias, preferencias del usuario… De esta forma, se podría integrar una capa aún mayor de complejidad a la página web.

Ahora bien, si un gran reto es lo que se quiere implementar, propongo como séptima sugerencia añadir a la página web un *Bot* creado en *Telegram* (por ejemplo) al que se le incorpore una capa de Inteligencia Artificial y que actúe como interfaz de soporte y ayuda para los usuarios de la página web. Es decir, aplicar, de forma similar, los *bots* de soporte que integra por ejemplo la página web de *Amazon*.

Como podemos observar estas son algunas de las posibles sugerencias que se podrían implementar. Sin embargo, su potencial es infinito por lo que, a niveles prácticos, se puede implementar de una forma sencilla y muy cómoda aquella funcionalidad que se desee únicamente haciendo pequeñas y localizadas variaciones en el código de este TFG.

En el siguiente apartado, (lo que representaría la vigésima séptima parada) expondremos las múltiples dificultades y experiencias obtenidas a lo largo de este recorrido.

13. Dificultades y experiencias

A lo largo de esta memoria hemos mencionado varias dificultades y experiencias que han ido surgiendo a medida que íbamos desarrollando cada uno de los puntos. No obstante, y a modo de resumen, las más importantes son las descritas a continuación:

Con toda seguridad, la mayor dificultad que tuve y que ocupa indiscutiblemente el puesto número uno es: el *CORS*. Tal como ya comenté cuando introduje y expliqué este problema, la solución parece evidente. Sin embargo, teniendo en cuenta que por un lado tenía el microservicio "braintree-api" con el "gateway-api" por encima, el cual se estaba comunicando con el *Root Config* ("micro-frontend-app"). A su vez, la petición la realizaba el microfrontend "mf-braintree-angular" que, por su implementación, se conectaba con los servidores externos de *Braintree*. Además, existía la dificultad añadida de que en ese momento la causa podía provenir de cualquiera de esos cinco elementos. Por lo que conseguir localizar el origen del fallo, sin que el propio error mostrase de dónde era, fue sin duda alguna el mayor desafío a superar. Puesto que, en caso de no conseguirlo, literalmente no podría presentar nada. Adicionalmente al ser el mayor desafío, también fue uno de los mayores conocimientos aprendidos, pues ahora domino mucho mejor la gestión del *CORS*.

El segundo lugar lo ocupa la redacción de este propio documento. Concretamente, redactar todos y cada uno de los conocimientos, el motivo de por qué eran necesarios y qué problema solucionaban. A la par que explicar su propio funcionamiento y cómo se combinaban y hacían sinergia con los otros elementos de este proyecto; ha sido sin lugar a duda la segunda mayor labor realizada. Esto se debe principalmente a la sensación de progresión, pues para cada nueva metodología era como si debiera empezar desde cero; y no es hasta el final, cuando ya se han mencionado todas, que se hace visible el gran potencial y la gran sinergia que tienen todas ellas unidas. Este trabajo es muy ambicioso y

es el motivo por el que (aun intentando resumir lo máximo posible algunos aspectos), me he visto en la obligación de exponer los conceptos para que se comprenda el valor que está aportando el hecho de utilizarlos.

La tercera mayor dificultad fue sin lugar a duda aprender y dominar *SingleSpa* en un tiempo récord. Puesto que debía hacerlo mientras lo implementaba. Mi experiencia respecto a *SingleSpa* es de máximo agradecimiento, pues me ha abierto los ojos al futuro de la implementación del *Frontend*, dado que en los próximos años las aplicaciones serán desarrolladas utilizando microfrontends en lugar de uno monolítico. Sin embargo, y aunque esté profundamente satisfecho y orgulloso de haber implementado y dominado *SingleSpa,* la experiencia ha sido muy dura no sólo en el plano temporal, sino también en el emocional.

Por ejemplo, para incluir y poder visualizar los *assets* (un recurso normalmente en formato imagen que está en una carpeta del propio código y únicamente se debe especificar el *path* relativo a esa carpeta para que muestre la imagen) en una aplicación monolítica tardaría no más de veinte segundos, pues es algo trivial. Pero, me llevó una hora exacta aprender cómo se debía hacer una tarea tan simple como mostrar un *asset* en *SingleSpa,*. Esto es porque este recurso en formato imagen que es el *asset* si bien está incluido en la propia *Parcela*, la ruta (el *path*) no es de la *Parcela* sino del *Root Config* que no tiene el *asset* pues se ubica dentro de la *Parcela.* Por ello, debe utilizarse una función concreta que ya la proporciona *SingleSpa* para poder añadir los *assets*. Realizar esta tarea ahora que ya he aprendido cómo se hace, supondría los mismos veinte segundos ya sea en un *Frontend* monolítico o uno en *SingleSpa* con microfrontends. No obstante, el coste de aprender a hacerlo es lo que a veces puede resultar algo desmotivador si lo comparas con cuánto podría haber llevado hacer la misma tarea en uno monolítico que es trivial pues sé perfectamente cómo realizarla.

Otra dificultad que tuve con *SingleSpa* fue el doble *rounting*. Hacer que el *rounting* externo que define qué *Parcelas* se van a visualizar, además del *rounting* interno para cada una de las *Parcelas* que lo requerían, supuso un auténtico desafío. Es por ello por lo que decidí contactar con el equipo de soporte en *Slack* (una plataforma que permite la comunicación entre diversas personas de forma similar a *Discord* o *Teams*) de *SingleSpa*, aunque en realidad es un conjunto de programadores y voluntarios que van respondiendo algunas dudas entre ellos. Tras hablar con dos de ellos, me dijeron que mi enfoque era erróneo y que no valía la pena realizar el doble *rounting*. Esto se debe a que, tal como se aprecia en la mayoría de los tutoriales y documentación existente, algunos tienen el concepto de que un microfrontend es una única página, un solo fichero *html*. Y que para cada una de las páginas (ficheros *html* para cada uno de los componentes) tendríamos que crear un microfrontend aparte. En mi caso, precisamente evitaba eso a toda costa, pues hubiera significado tener muchos más microfrontends con una complejidad tan básica como la que tiene el *Footer*. Por ese motivo y sabiendo que “no son oficiales”, decidí proseguir en mi camino y seguir probando hasta que finalmente di con la solución y logré mi propósito.

Adicionalmente, otra dificultad en *SingleSpa* fue la configuración de los puertos para cada uno de los microfrontends. Por lo que al final, opté por modificar todos los ficheros de configuración y así únicamente ejecutar “npm start” para cada uno de todos los microservicios (tanto el *Root Config*, como el módulo *Utility* como también todas y cada una de las Parcelas). Cabe destacar que para el módulo *Utility* será necesario instalar un paquete, pues la instrucción “npm start” estará ejecutando internamente “http-server . --cors”.

Por último, implementar microfrontends con tres frameworks distintos (*React, Angular* y *Vue*) también supuso un reto. Dado que debí aprender *React,* que desconocía por

completo. Además de tener que desarrollar tres implementaciones distintas para adoptar una misma característica, como por ejemplo las traducciones.

Concluyendo este apartado de dificultades y experiencias, añadiré que trabajando con *SingleSpa* he tenido la sensación de que paso a paso iba escalando y conquistando la cumbre del *Everest*, hasta que finalmente (después de dar con la solución al doble *rounting*), sentí como había colocado la bandera en la cumbre.

En nuestra aventura llegamos al siguiente apartado, donde analizaremos las conclusiones extraídas después de haber realizado este gran viaje.

14. Conclusión Global

Tal y como se ha mencionado anteriormente, el planteamiento y desarrollo de este TFG es sumamente ambicioso.

Aun así, a excepción del objetivo *deploy* en *AWS,* se han cumplido y desarrollado satisfactoriamente todos y cada uno de los objetivos principales y secundarios de este proyecto:

- Nuestra web no se centra en qué características ofrece sino en cómo ofrece esas características a nivel interno.
- La página web permite gestionar unos cursos que tienen asociadas unas publicaciones, las cuales tendrán asociados ficheros. Permite poder crear, comprar y visualizar estos cursos.
- Se ha logrado implementar satisfactoriamente un chat para cada curso en el que se ha prestado especial cuidado en su representación gráfica.
- Este proyecto se ha desarrollado utilizando un *Backend* basado en microservicios *Spring Boot* en *Java* y un *Frontend* basado en microfrontends al incorporar *SingleSpa.*
- Se ha profundizado y logrado dominar las tecnologías y retos nuevos que suponen los microservicios y los microfrontends.
- Se ha aplicado satisfactoriamente la *Arquitectura Hexagonal* en nuestros dos microservicios principales (“auth-api” y “res-api”) además de crear microfrontends complejos (y no simplemente páginas estáticas) al incorporar un doble *rounting*, tanto en *SingleSpa* como para los microfrontends.
- Se ha conseguido abstraer toda la lógica de los microservicios del *Backend* haciéndolos completamente independientes, flexibles, escalables, modulables y autosuficientes al incorporar *Spring Cloud Config*, *Discovery Server* y el servicio de *Gateway.*
- Se han utilizado los tres *frameworks* más comunes en la creación de *Frontends* por parte del sector empresarial: *React*, *Angular* y *Vue*.
- Se ha incorporado satisfactoriamente la pasarela de pago *Braintree* en modo sanbox a nuestra página web, permitiendo que, únicamente con cambiar las propiedades a una cuenta que no sea sandbox, pueda pasarse a un entorno de transacciones real.

Por último, me enorgullece pensar que mi TFG pueda llegar a servir como motivación para romper nuestros propios límites y aventurarnos a aprender nuevos conceptos y mecánicas. Siendo así, que pueda ser considerado como un “template” para otros trabajos o consultas de otros alumnos o incluso otros desarrolladores. Debido a que no hay ejemplos

publicados que engloben todas estas características trabajando al unísono en un único proyecto.

Del mismo modo, al ser microservicios y microfrontends realmente es muy flexible pues no es obligado modificar mi código. Sino que, tan sólo utilizándolo, ellos podrían crear nuevos microservicios y nuevos microfrontends con sus propias ideas y las funcionalidades que quieran desarrollar en su página web.

15. Fuentes Utilizadas

Atlassian. (s. f.). *Comparación entre la arquitectura monolítica y la arquitectura de microservicios*. https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith

Getting Started | Service Registration and Discovery. (s. f.). Getting Started | Service Registration and Discovery. https://spring.io/guides/gs/service-registration-and-discovery/

Spring Cloud Config. (s. f.). https://docs.spring.io/spring-cloud-config/docs/current/reference/html/

Service Discovery. (s. f.). https://reactiveprogramming.io/blog/es/patrones-arquitectonicos/service-discovery

Nicolás, J. (2022). CORS. Qué es, cómo funciona, para qué sirve y cómo solucionarlo. *Juan Nicolás*. https://www.juannicolas.eu/cors-que-es-y-como-funciona/

Picodotdev. (2021, 7 febrero). Introducción a DDD y arquitectura hexagonal con un ejemplo de aplicación en Java. *Blog Bitix*. https://picodotdev.github.io/blog-bitix/2021/02/introduccion-a-ddd-y-arquitectura-hexagonal-con-un-ejemplo-de-aplicacion-en-java/

Salguero, E. (2023, 22 enero). Arquitectura Hexagonal - Edu Salguero - Medium. *Medium*. https://medium.com/@edusalguero/arquitectura-hexagonal-59834bb44b7f

OAuth 2.0 Resource Server JWT :: Spring Security. (s. f.). https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

Getting Started with single-spa | single-spa. (s. f.). https://single-spa.js.org/docs/getting-started-overview

Configuring single-spa | single-spa. (s. f.). https://single-spa.js.org/docs/configuration/

Getting Started | Enabling Cross Origin Requests for a RESTful Web Service. (s. f.). Getting Started | Enabling Cross Origin Requests for a RESTful Web Service. https://spring.io/guides/gs/rest-service-cors/

Losada, C. (2021, 14 diciembre). *Single Page App VS Multi Page App: ¿cuál elegir para tu web? - Making Science*. Making Science. https://www.makingscience.es/blog/single-page-app-vs-multi-page-app-cual-elegir-para-tu-web/

Module Federation. (s. f.). GitHub. https://github.com/module-federation

JS Frameworks. (2021, 24 octubre). *Single spa Micro frontend - Methods of inter app communication* [Vídeo]. YouTube. https://www.youtube.com/watch?v=RvRbe0UbcSY

Luigi Code. (2021, 31 agosto). *#microservices #springboot #springcloud Microservicios con Spring Boot: Capítulo 1: User* [Vídeo]. YouTube. https://www.youtube.com/watch?v=czWbpqC1fLY

Luigi Code. (2021b, agosto 31). *#microservices #springboot #springcloud Microservicios con Spring Boot: Capítulo 1: User* [Vídeo]. YouTube. https://www.youtube.com/watch?v=czWbpqC1fLY

Hamza. (2022, 1 julio). *How to use Redis with Spring boot to store sessions and why* 🍀 [Vídeo]. YouTube. https://www.youtube.com/watch?v=4K5N7SRcyK8

Tech Guides and Thoughts. (2020, 18 abril). *Welcome to the Vault on AWS Series!* [Vídeo]. YouTube. https://www.youtube.com/watch?v=7qynYJl3lRk

Avinashkumar - The Learning Destination. (2021, 25 noviembre). *EPISODE-06: HashiCorp Vault - How to enable Token & GitHub based authentication in Vault #devops* [Vídeo]. YouTube. https://www.youtube.com/watch?v=_paG3VNufpA

Rahul Wagh. (2022, 19 octubre). *HashiCorp Vault Token Authentication & GitHub Authentication - Part 6 | HashiCorp Vault tutorials* [Vídeo]. YouTube. https://www.youtube.com/watch?v=LX2YrBbGOGQ

SuresPraja Channel. (2020, 7 julio). *SERVICE DISCOVERY IN AWS (USING EUREKA SERVER, EUREKA CLIENT)* [Vídeo]. YouTube. https://www.youtube.com/watch?v=5EhXQyMQNk4

Byte Code. (2021, 1 febrero). *Enviando Correos Electrónicos con Spring Boot #ByteCode #Stream #7* [Vídeo]. YouTube. https://www.youtube.com/watch?v=Jtprryt9Dts

BEC. (2022, 20 mayo). *Building micro frontends with Single-Spa – Kamil Dzieniszewski* [Vídeo]. YouTube. https://www.youtube.com/watch?v=lQkR-Vlnbgs

Luigi Code. (2022, 30 diciembre). *Pagos con Braintree Springboot y Angular Full Stack: Parte 1: Crear el proyecto* [Vídeo]. YouTube. https://www.youtube.com/watch?v=1RMKFBq_Z_0

Getting Started | Enabling Cross Origin Requests for a RESTful Web Service. (s. f.-b). Getting Started | Enabling Cross Origin Requests for a RESTful Web Service. https://spring.io/guides/gs/rest-service-cors/

Angular. (s. f.). https://angular.io/guide/i18n-overview

PlayList Spting Boot + AWS. Collection of AWS related Tutorial created by Techno Town Techie. *YouTube*. (s. f.). https://www.youtube.com/playlist?list=PL4TnYdea-xTJpaL2XigZ2ulcPyoRPjHAI

Leifer Mendez. (2022, 13 septiembre). *¿Esto es MICRO-FRONTEND? Angular y React explicación y ejercicio practico* [Vídeo]. YouTube. https://www.youtube.com/watch?v=_6QZFdSitHo

redbee Studios. (2021, 3 febrero). *Implementacion de Micro frontend con SingleSPA* [Vídeo]. YouTube. https://www.youtube.com/watch?v=rkYAjLVPAn4

weincode. (2021, 22 junio). *Mircro frontends en Angular 🌐Parte 2 - compartiendo info, enrutando, y otros cosas más* 🤪 [Vídeo]. YouTube. https://www.youtube.com/watch?v=eAg-bCL4Ob8

Domini Code. (2022, 6 mayo). *Microfrontend con Single-SPA & Module Federation (charla)* [Vídeo]. YouTube. https://www.youtube.com/watch?v=ymKzE3u3X_s

JS Frameworks. (2023, 5 febrero). *Single-spa Angular applications using webpack module federation* [Vídeo]. YouTube. https://www.youtube.com/watch?v=B_54k0pN59s

Single-Spa. (s. f.). *Guide or example to deploy in production · Issue #579 · single-spa/single-spa*. GitHub. https://github.com/single-spa/single-spa/issues/579

JS Frameworks. (2021b, octubre 24). *Single spa Micro frontend - Methods of inter app communication* [Vídeo]. YouTube. https://www.youtube.com/watch?v=RvRbe0UbcSY

How to translate your Angular app with ngx-translate. (2023, 14 febrero). https://www.codeandweb.com/babeledit/tutorials/how-to-translate-your-angular-app-with-ngx-translate

il8n not working in react app after converting to single spa. (s. f.). Stack Overflow. https://stackoverflow.com/questions/67659687/il8n-not-working-in-react-app-after-converting-to-single-spa

Cristian Ballesteros. (2021, 3 enero). *configurar aplication properties de Spring Boot para conectar a la base de datos MySQL O Postgres* [Vídeo]. YouTube. https://www.youtube.com/watch?v=oMfohZyQXQM

16. Anexos

En lo que es ya nuestra trigésima y última etapa de todo nuestro viaje, a continuación, se muestran los anexos que contendrán algunos aspectos importantes, pero no tan relevantes de esta memoria.

a. Anexo 1

En este anexo se muestra la foto final de la estructuración del código implementado, con todos los microservicios y microfrontends.

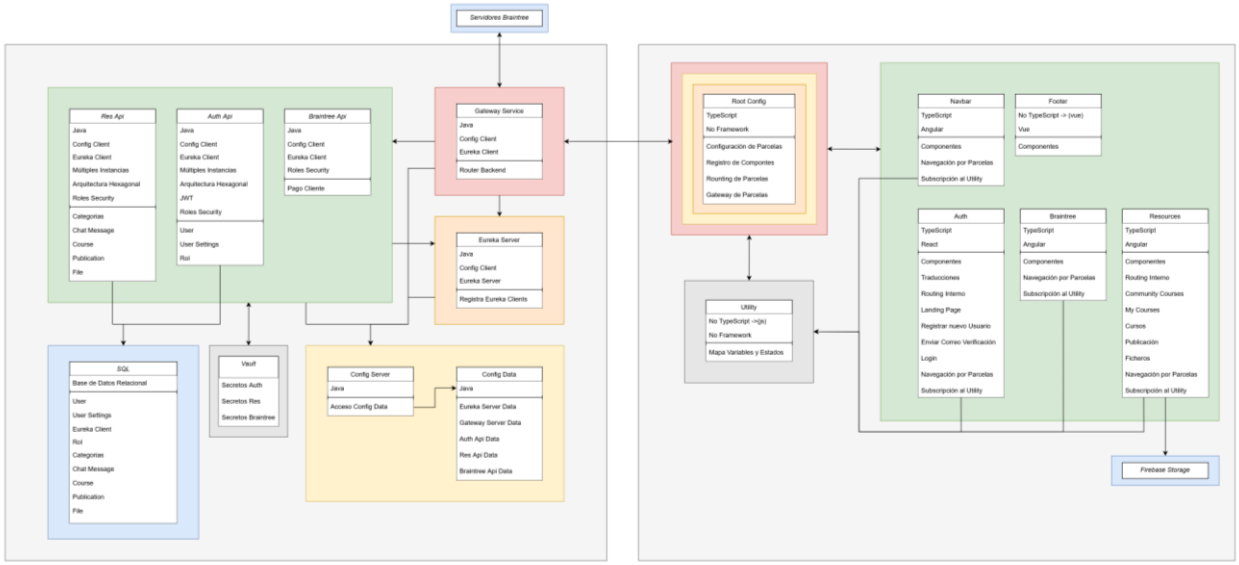

Ilustración 47: Representación de la foto final de la estructuración del código implementado con todos los microservicios, microfrontends y elementos utilizados. La sección izquierda corresponde al Backend mientras que la derecha hace referencia al Frontend. Esta misma imagen es la aportada y explicada en el Anexo1.

b. Anexo 2

A continuación, mostramos la configuración de la clase principal del microservicio “config-service” en el que se puede ver la etiqueta de configuración “@EnableConfigServer”:

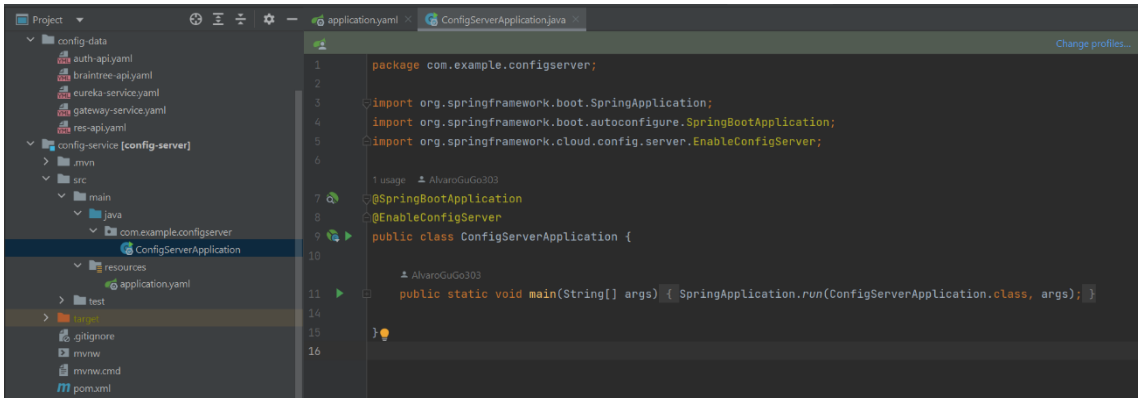

Ilustración 48: Representación del fichero “ConfigServerApplication.java” perteneciente a la clase principal del microservicio “config-service” donde podemos observar que se le ha añadido la etiqueta “@EnableConfigServer”.

De la misma forma, en el archivo *pom* del Servidor se deberá añadir la siguiente dependencia:

```
21      <dependency>
22          <groupId>org.springframework.cloud</groupId>
23          <artifactId>spring-cloud-config-server</artifactId>
24      </dependency>
```

Ilustración 49: Representación del fichero “pom.xml” del microservicio “config-service” en el que podemos apreciar la dependencia que es necesaria añadir.

c. Anexo 3

La configuración de las dependencias necesaria que hay que añadir en los ficheros *pom* para que cada uno de los microservicios actúen como *Config Clients* será:

```
16      <properties>
17          <jakarta-servlet.version>5.0.0</jakarta-servlet.version>
18          <java.version>17</java.version>
19          <!--Need for Config Client-->
20          <spring-cloud.version>2022.0.2</spring-cloud.version>
21      </properties>
```

Ilustración 50: Representación del fichero “pom.xml” del microservicio “auth-api” en el que será necesario añadir la versión de Spring Cloud.

```
45          <!--Need for Config Client-->
46          <dependency>  <!---->
47              <groupId>org.springframework.cloud</groupId>
48              <artifactId>spring-cloud-starter-config</artifactId>
49          </dependency>
50          <!--Need for Config Client-->
51          <dependency>  <!---->
52              <groupId>org.springframework.cloud</groupId>
53              <artifactId>spring-cloud-starter-bootstrap</artifactId>
54          </dependency>
```

Ilustración 51: Representación del fichero "pom.xml" del microservicio "auth-api" en el que deberemos añadir las siguientes dependencias para que actúe como un Config Client.

```
104     <!--Need for Config Client-->
105     <dependencyManagement>
106         <dependencies>
107             <dependency> <!---->
108                 <groupId>org.springframework.cloud</groupId>
109                 <artifactId>spring-cloud-dependencies</artifactId>
110                 <version>${spring-cloud.version}</version>
111                 <type>pom</type>
112                 <scope>import</scope>
113             </dependency>
114         </dependencies>
115     </dependencyManagement>
```

Ilustración 52: Representación del fichero "pom.xml" del microservicio "auth-api" en el que será necesario añadir la siguiente dependencia en el bloque de "dependencyManagement".

d. Anexo 4

En la ilustración 4 se mostró la versión contraída de la configuración del token que permite que el *Config Server* pueda acceder al repositorio privado de *Github*. A continuación, se muestra la versión expandida de dicho token:

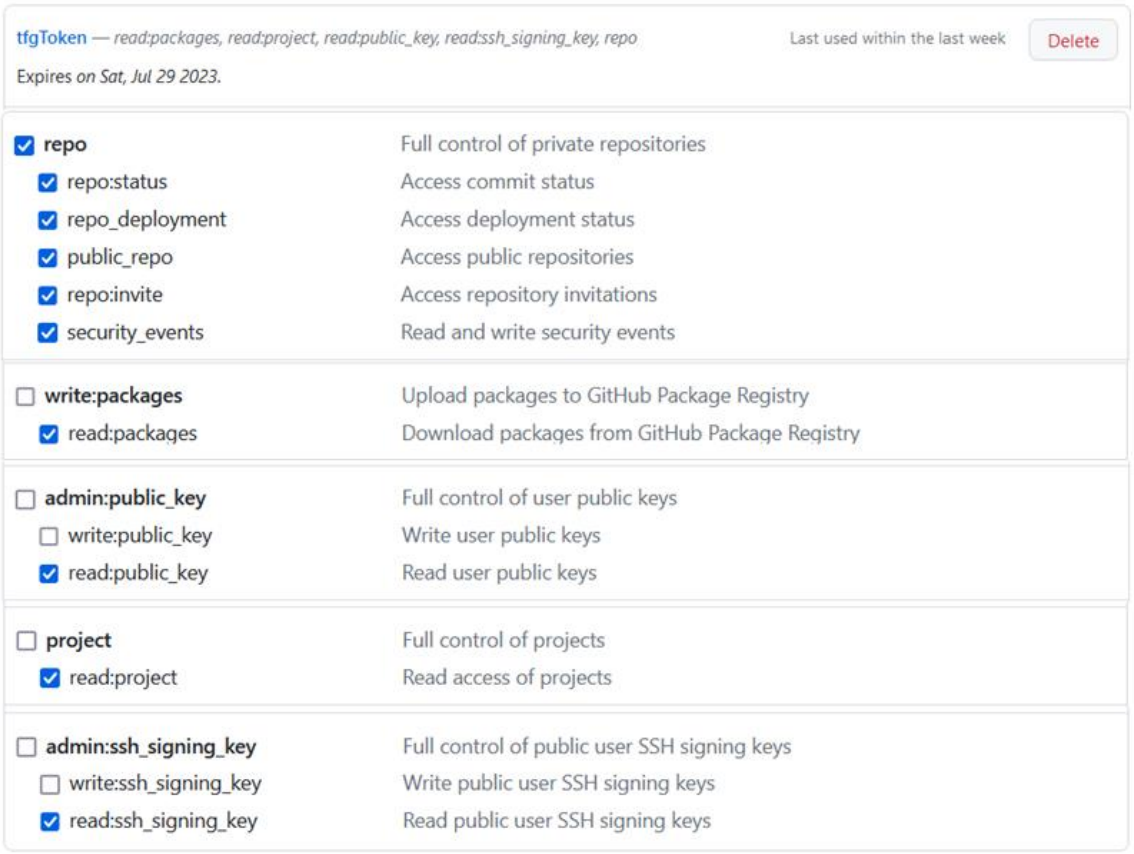

Ilustración 53: En la ilustración se observa la configuración expandida de cada uno de los campos necesarios del token que accederá a nuestro repositorio privado de Github.

e. Anexo 5

En este anexo se explicarán en detalle capa una de las propiedades necesarias para configurar *Vault* en el fichero de propiedades “bootstrap.yaml” de cualquier microservicio.

El campo de “authentication” corresponde a los distintos métodos de autentificación que hay (TOKEN, APPID o APPROLE), en este caso (en su uso localhost) se ha utilizado TOKEN.

El campo de “host” establece cuál es nombre del host que se utilizará para la validación del certificado SSL; en nuestro caso será “localhost”.

El campo de “token” será necesario en caso de utilizar TOKEN como método de autentificación. Este campo establecerá el token estático para poderlo usar.

El campo de “port” indicará donde está desplegado Vault.

El campo de “application-name” indicará cuál es el secreto que le pertenece a este microservicio; este campo es fundamental que coincida exactamente con el nombre del microservicio para que no haya ningún tipo de problema.

Por último, el campo de “scheme” podrá ser http o https. En nuestro caso, y al tratarse de localhost, se ha configurado como http.

f. Anexo 6

En este anexo se mostrará como añadir un secreto a *Vault* con la interfaz que se visualiza en el propio navegador. A continuación, lo primero que debemos hacer es crear un nuevo secreto tal como se puede observar a continuación:

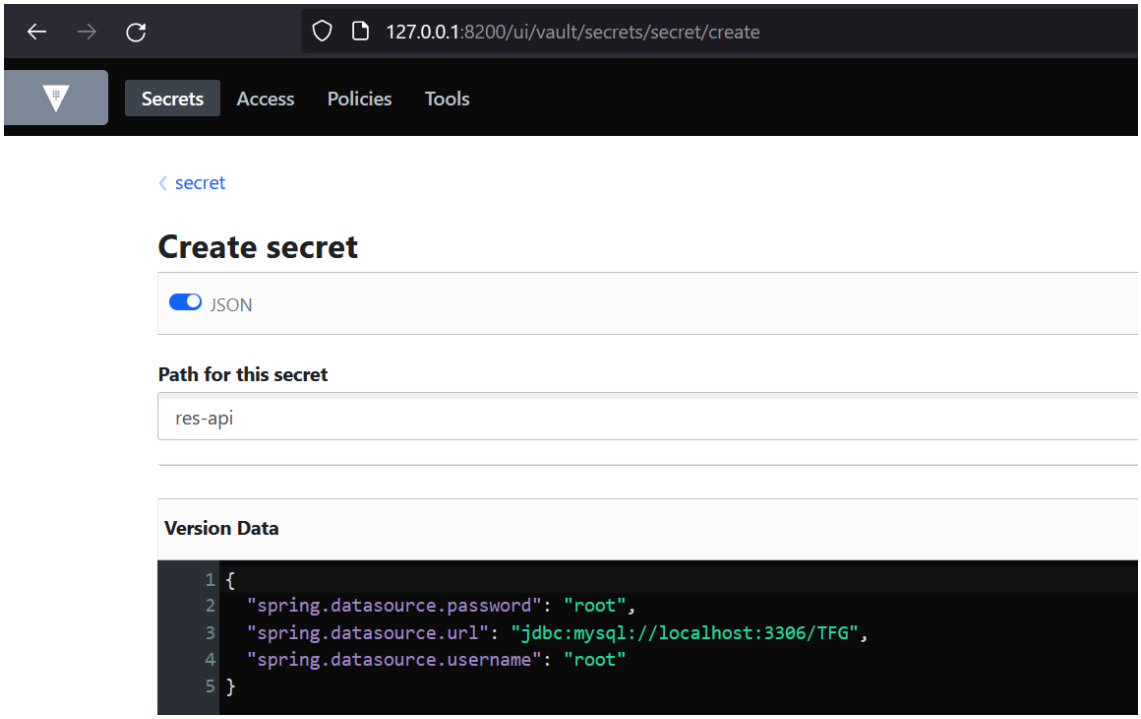

Ilustración 54: Ilustración en la que se visualiza el momento de la creación de un secreto en Vault.

Finalmente, una vez añadido podemos regresar a la página anterior y visualizar cómo realmente se está guardando la información de los secretos en *Vault*:

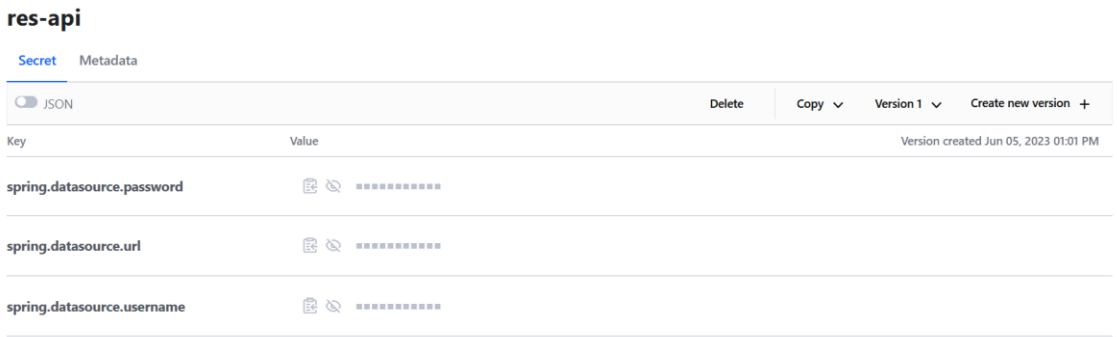

Ilustración 55: Representación de la visualización de los secretos en Vault.

g. Anexo 7

En este anexo se mostrará la configuración necesaria en la clase principal del microservicio “eureka-service” así como también la configuración de su *pom*.

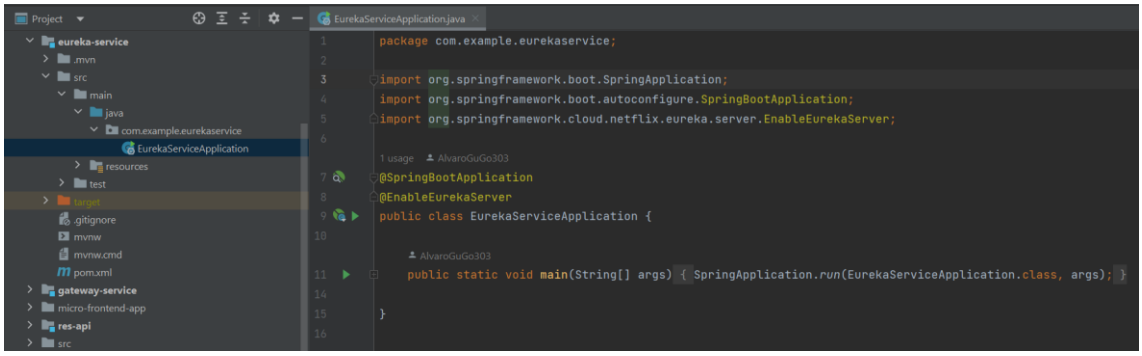

Ilustración 56: Representación del fichero “EurekaServiceApplication.java” que muestra la clase principal de este microservicio “eureka-service” en el que se le ha agregado la etiqueta: “@EnableEurekaServer”.

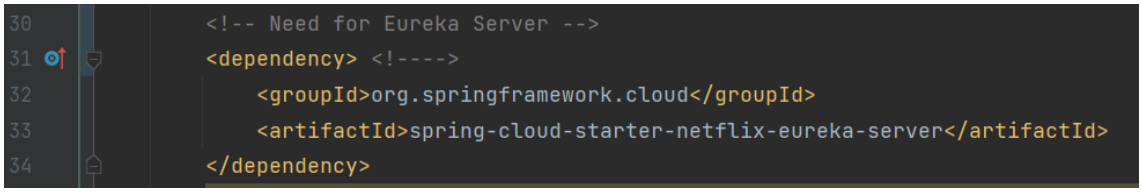

Ilustración 57: Representación del fichero “pom.xml” del microservicio “eureka-service” en el que será necesario añadir la siguiente dependencia para que actúe como un Eureka Server.

h. Anexo 8

En este anexo mostraremos la configuración necesaria que se debe añadir para que un microservicio pase a ser un *Eureka Client*, es decir, implemente bel patrón *Eureka* por el lado del Cliente.

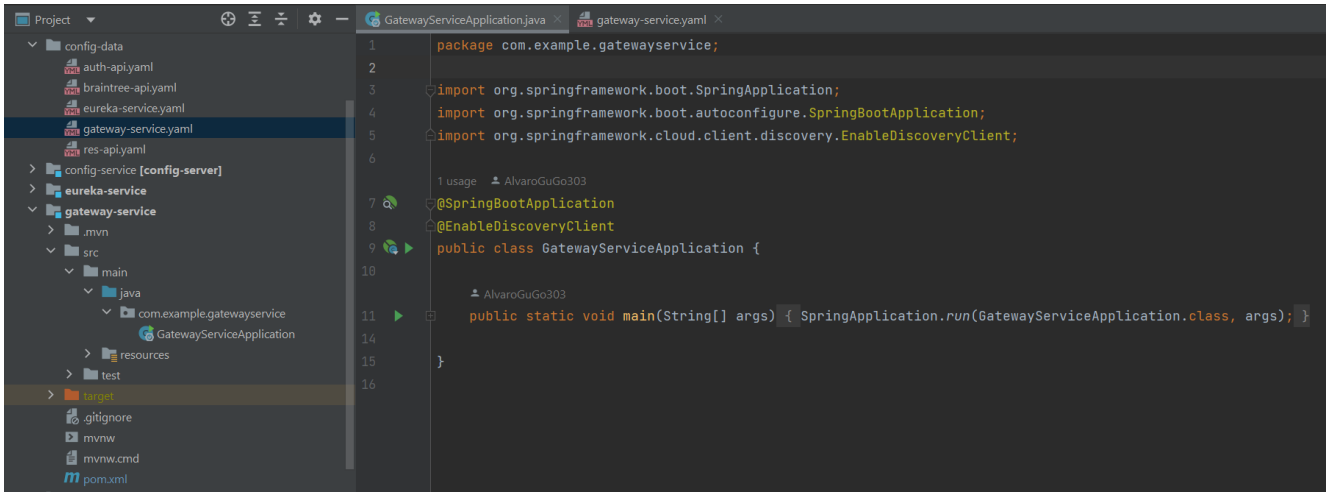

Ilustración 58: Representación del fichero “GatewayServiceApplication.java” que muestra la clase principal de este microservicio “gateway-service” en el que se le ha agregado la etiqueta: “@EnableDiscoveryClient”.

```
34        <!-- Need for Eureka Client -->
35        <dependency>
36            <groupId>org.springframework.cloud</groupId>
37            <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
38        </dependency>
```

Ilustración 59: Representación del fichero "pom.xml" del microservicio "gateway-service" en el que será necesario añadir la siguiente dependencia para que actúe como un Eureka Client.

i. Anexo 9

Para implementar el patrón Gateway, será necesario modificar el *pom*, pues deberemos añadir la siguiente dependencia:

```
30        <!-- Need for Gateway -->
31        <dependency>
32            <groupId>org.springframework.cloud</groupId>
33            <artifactId>spring-cloud-starter-gateway</artifactId>
34        </dependency>
```

Ilustración 60: Representación del fichero "pom.xml" del microservicio "gateway-service" en el que será necesario añadir la siguiente dependencia para que actúe como un Gateway.

j. Anexo 10

En este anexo se mostrará el contenido del microservicio "auth-api", pues este microservicio cuenta con cuatro paquetes generales ("config", "security", "shared" y "users").

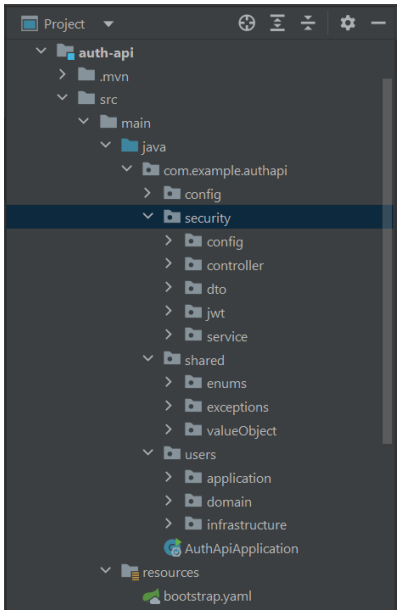

Ilustración 61: Representación interna de la estructura de carpetas del microservicio "auth-api". De entre los cuatro, será únicamente el paquete "users" el que implemente la Arquitectura Hexagonal.

k. Anexo 11

El propósito de este anexo es mostrar la entidad del usuario en la capa de *Dominio* de la *Arquitectura Hexagonal*. Pues para la gestión de los roles será necesario que implemente a la clase “UserDetails” tal como puede verse a continuación:

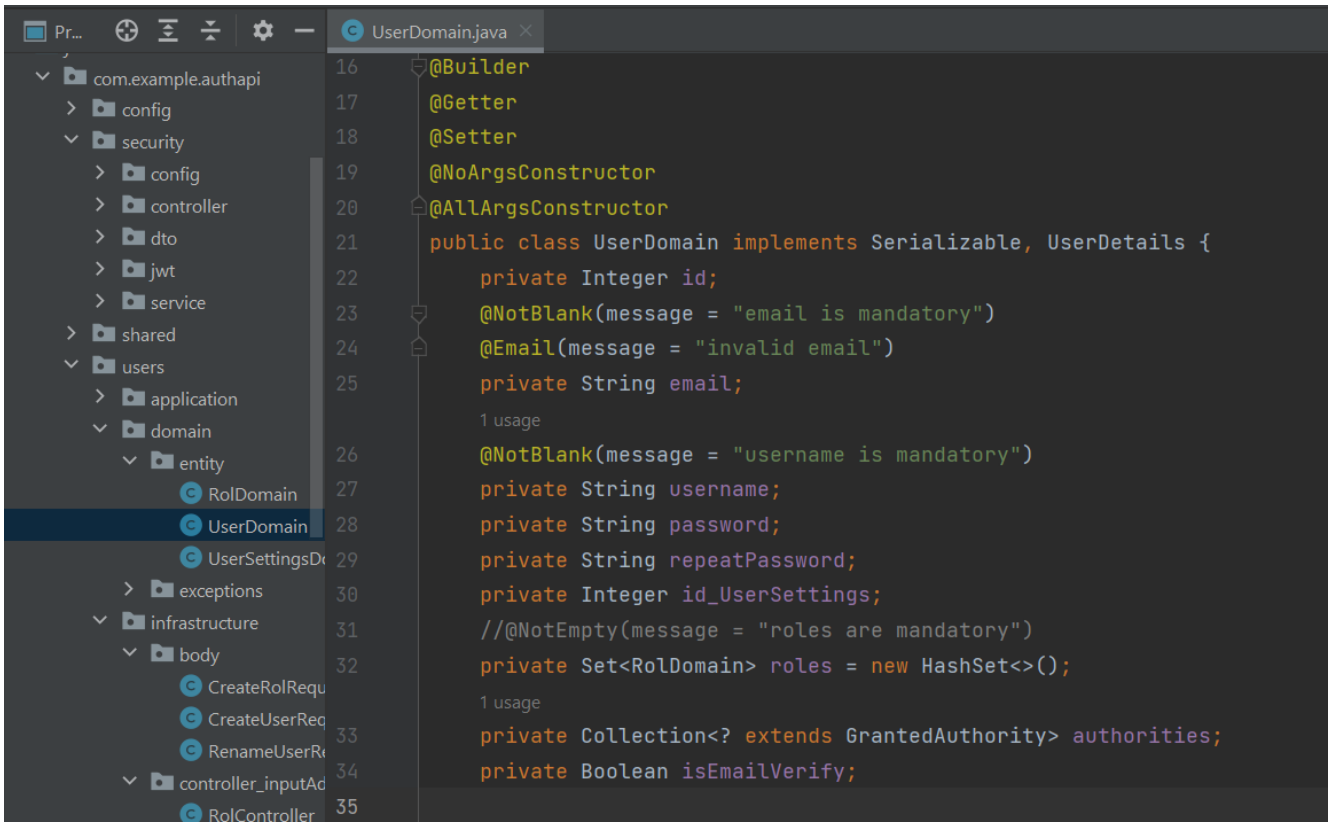

Ilustración 62: Representación del fichero “UserDomain.java” presente en la capa de Dominio del paquete “users” en el microservicio “auth-api”. En la ilustración puede observarse la implementación y el campo necesario para utilizar GrantedAuthorities con UserDetails y así poder validar el token.

Esto se debe a que, a la hora de hacer la autentificación del token, la clase que está esperando es esa pues contiene una serie de argumentos internos sobre los que él mismo trabaja.

l. Anexo 12

En este anexo se visualizarán las imágenes correspondientes a la configuración de la cuenta de Google utilizada para permitir que se envíen correos electrónicos por la aplicación web en el momento de la creación de la cuenta por parte del usuario.

El primer paso será activar la verificación en dos pasos ubicada en el apartado de seguridad de los ajustes de la cuenta de Google.

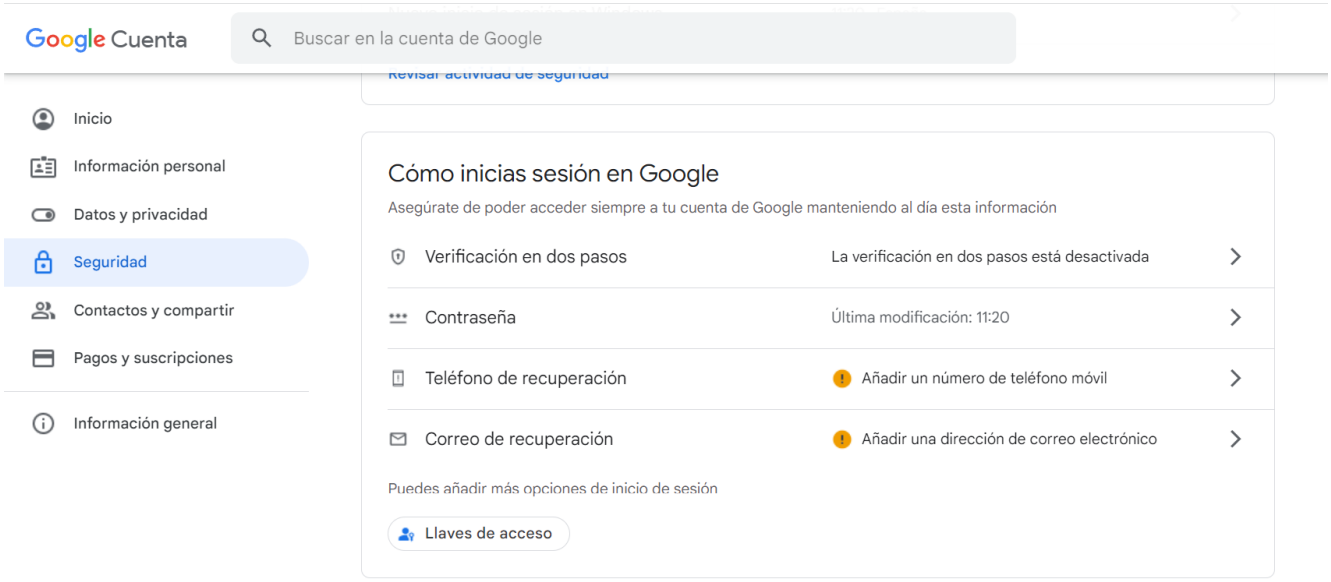

Ilustración 63: Representación de la configuración de seguridad de tu cuenta de Google, concretamente en el apartado de verificación en dos pasos que puede observarse que está desactivado.

A continuación, seleccionamos la opción que deseemos entre recibir un mensaje de texto o una llamada telefónica.

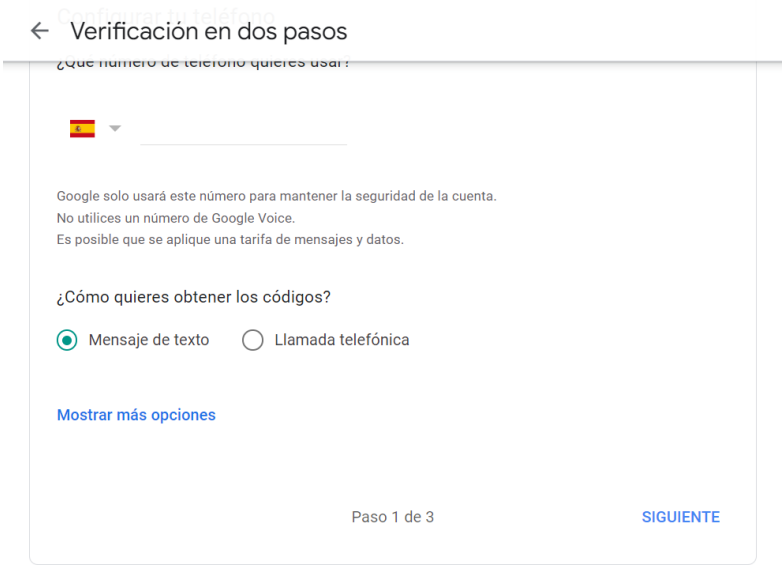

Ilustración 64: Representación de los pasos a seguir para activar la verificación en dos pasos en tu cuenta de Google.

Continuamos pasando por cada uno de los tres pasos hasta que finalmente activamos la verificación en dos pasos por lo que deberíamos observar lo siguiente:

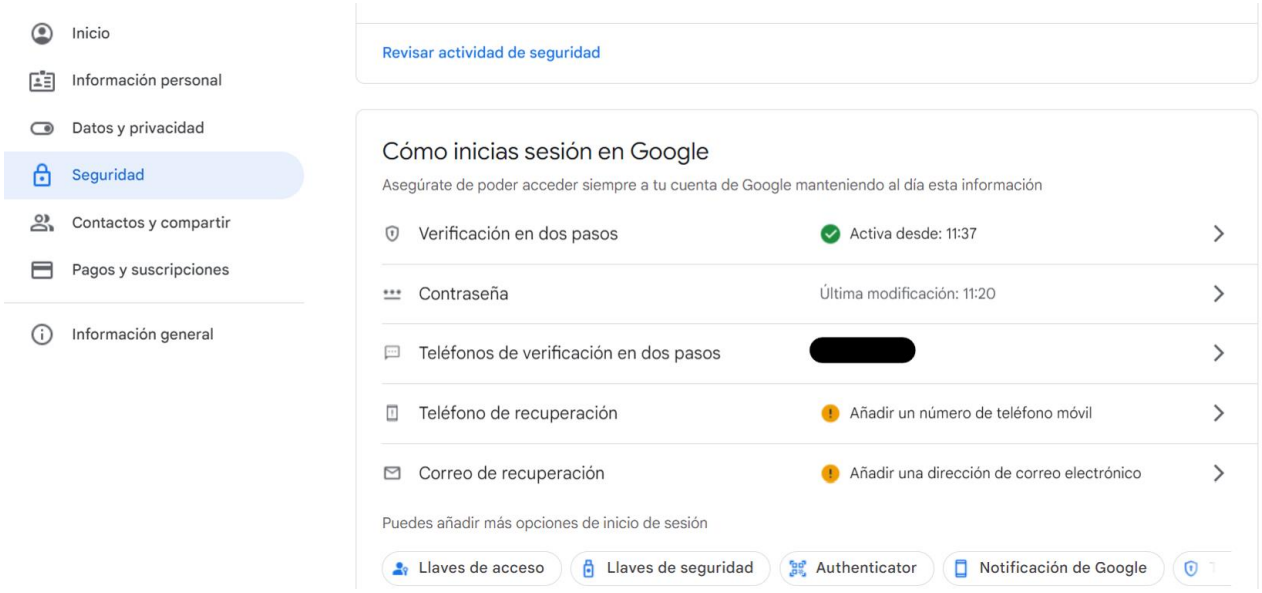

Ilustración 65: Representación de la configuración de seguridad de Google donde puede apreciarse que la verificación en dos pasos ha sido activada.

Una vez activada esa característica, debemos buscar las contraseñas de aplicaciones puesto que es lo que necesitaremos activar para poder mandar los correos electronicos.

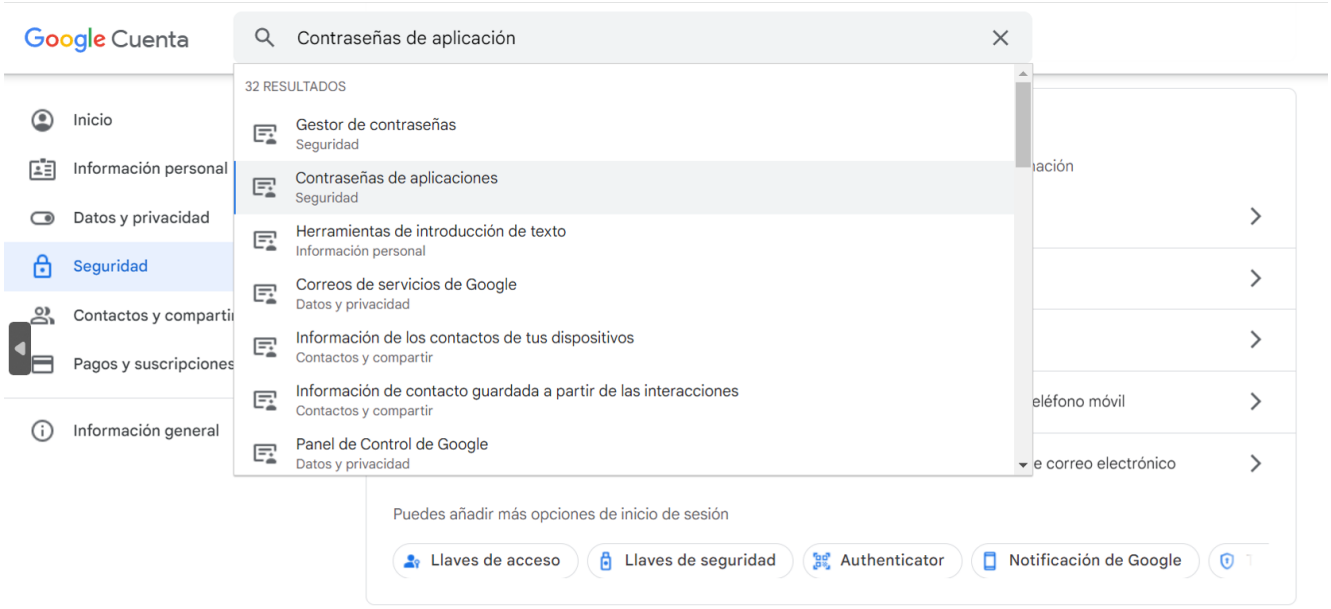

Ilustración 66: Representación de la búsqueda dentro de la configuración, de la sección "Contraseñas de aplicaciones".

A continuación debemos seleccionar el tipo de aplicación y el dispositivo al cual damos acceso, para que pueda realizar esa petición (enviar correos electrónicos). En nuestro caso, seleccionaremos lo siguiente:

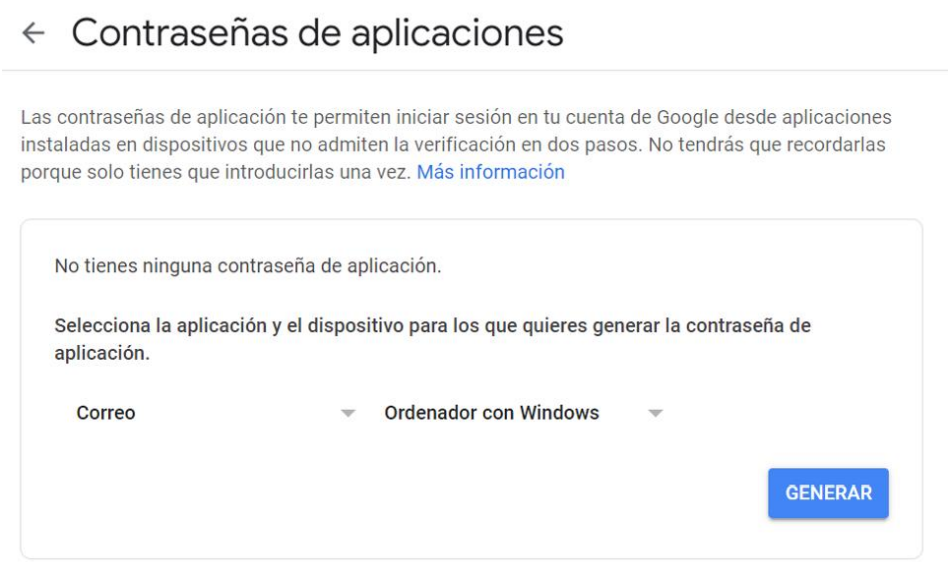

Ilustración 67: Representación de la configuración seleccionada para generar una contraseña de aplicación.

Por último, al pulsar en el botón “Generar” deberemos guardarnos esta contraseña generada automáticamente pues es la que nos permitirá poder enviar los correos desde esa cuenta de Google.

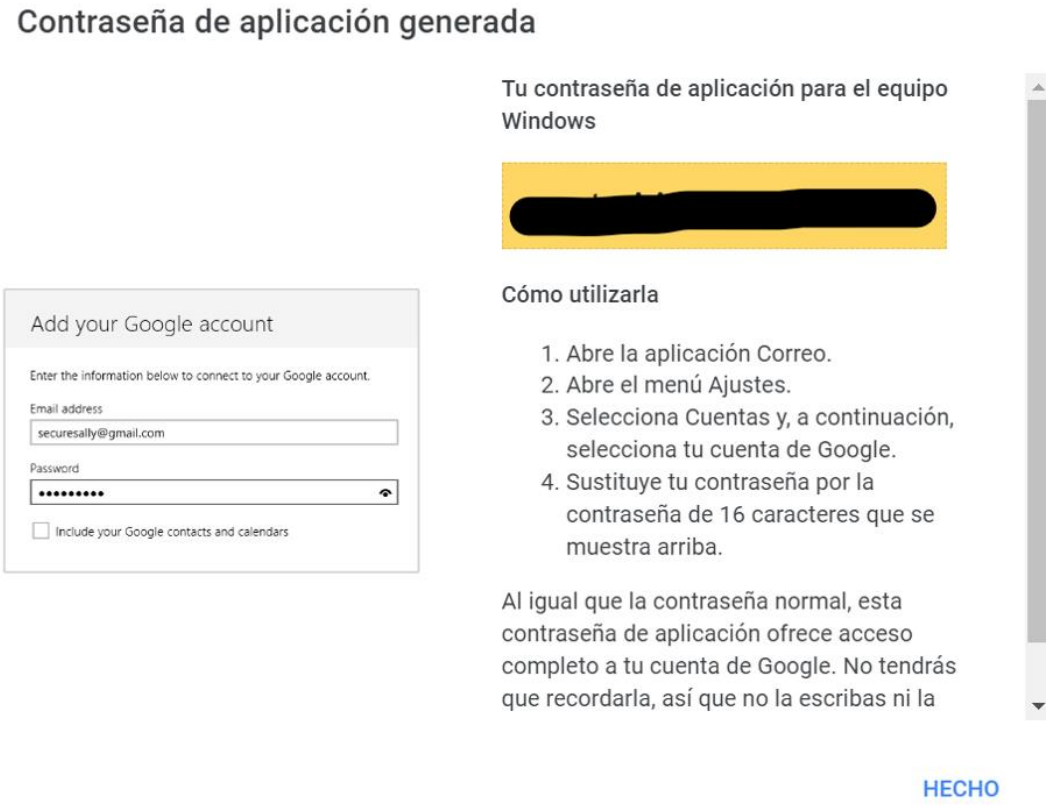

Ilustración 68: representación de la obtención de la contraseña de aplicación que añadiremos al en el Vault *del microservicio “auth-api”, pues será quien mande el correo electrónico.*

m. Anexo 13

En este anexo se muestra el diagrama de casos de uso de la totalidad de nuestra página web. En este diagrama están representadas las diversas funcionalidades que puede desempeñar un usuario con la web, independientemente del microservicio o microfrontends al cual se estén redirigiendo internamente.

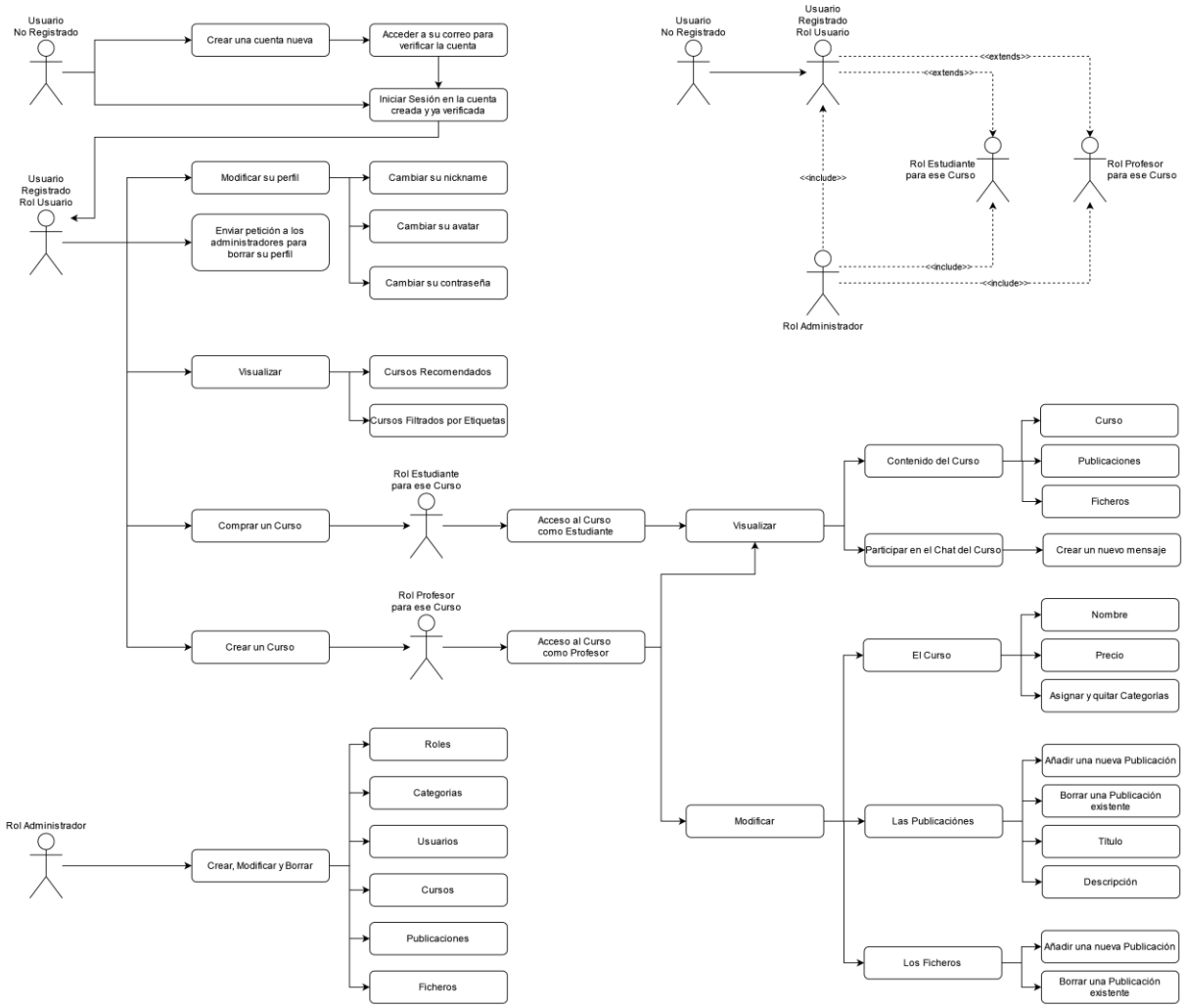

Ilustración 69: Representación de la totalidad del diagrama de casos de uso que pueden realizar los usuarios en la web. En la sección superior derecha de la ilustración se aprecian las distintas relaciones entre los diversos tipos de roles.

n. Anexo 14

En este anexo se visualiza la jerarquía y disposición de microfrontends que englobarán a la totalidad de nuestro *Frontend*.

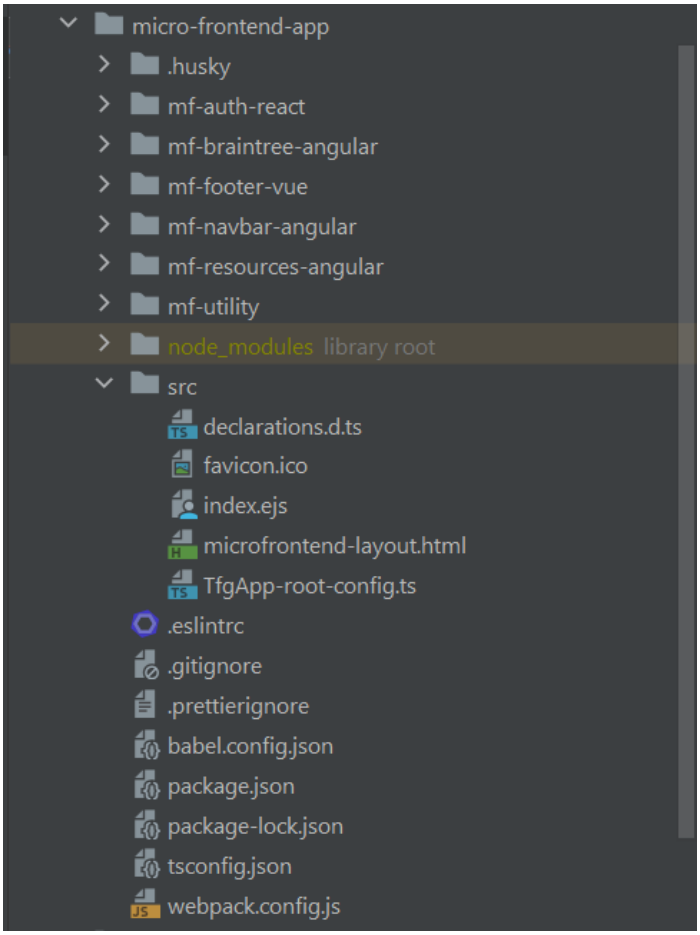

Ilustración 70: En la ilustración puede apreciarse que todas las parcelas, y el módulo Utility están ubicados bajo el Root Config ("micro-frontend-app") que actúa como contenedor monorepo de los distintos microfrontends que engloban la parte del Frontend de nuestra aplicación web.

o. Anexo 15

El objetivo de este anexo es mostrar que, en lugar de tener que importar las distintas librerías de terceros que importaríamos en cada una de nuestras *Parcelas*, si utilizamos *SingleSpa*, el *import* debe realizarse en el *Root Config*.

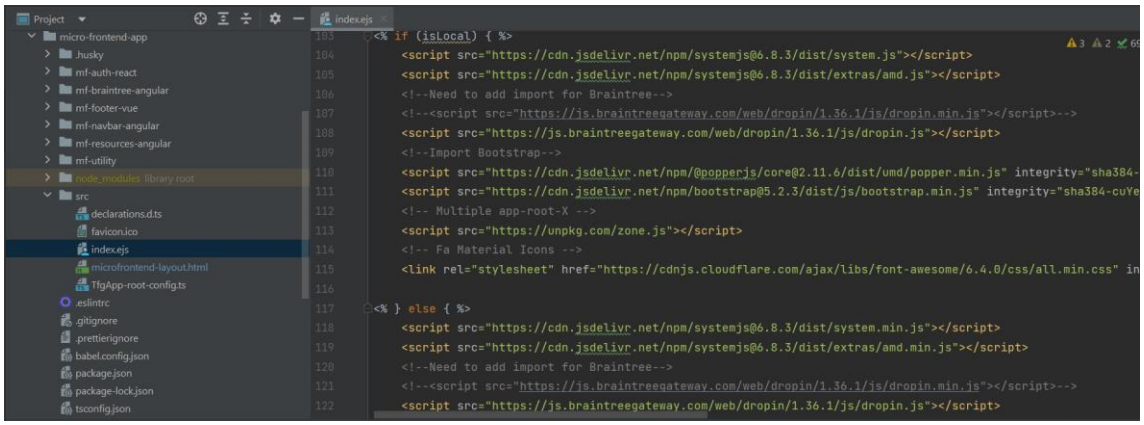

Ilustración 71: Representación del fichero "index.ejs" del microfrontend Root Config en donde observamos los múltiples imports de las librerías que necesitarán las Parcelas.

Por ello, en el caso de los *imports* del *Material Icons* o el *Bootstrap* deberían ser añadidos allí. No obstante, (y con el fin de no diferenciar entre ambos entornos: producción y local, pues en ambos el *import* será el mismo); existen algunos *imports* que se ubican fuera tal como se muestra en la siguiente imagen:

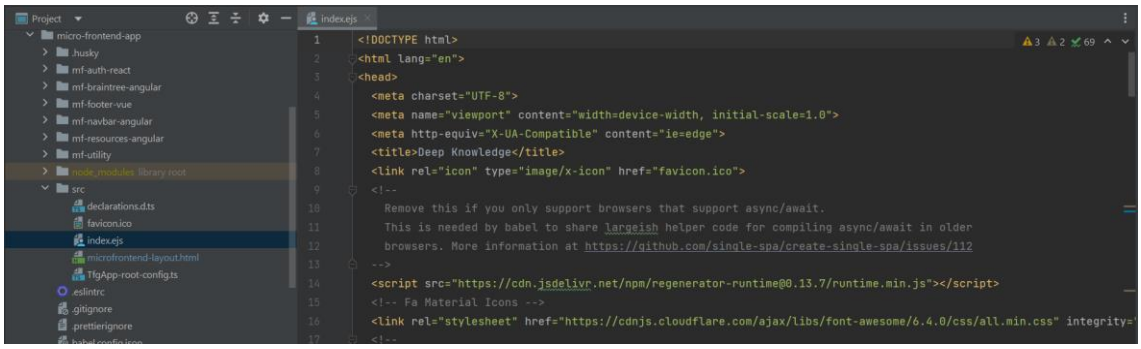

Ilustración 72: Representación del mismo fichero "index.ejs" del microfrontend Root Config en donde observamos la presencia de algunos imports que no dependen de si es en modo local o en modo de producción.

Tal como podemos observar, el *import* del *Material Icons* ademas de otros *scripts* se hace directamente sin diferenciar entre los entornos de producción y desarrollo; pues en ambos casos serán las mismas lineas de código.