universidad de león

Escuela de Ingenierías
Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

APICulture: Generación dinámica de servicios web con funcionalidades CRUD

APICulture: Dynamic generation of web services with CRUD functionalities

Autor: Sergio Chimeno Alegre
Tutor: José Alberto Benítez Andrades y Martín Bayón Gutiérrez

(Septiembre, 2022)

UNIVERSIDAD DE LEÓN
Escuela de Ingenierías Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Grado

ALUMNO: Sergio Chimeno Alegre
TUTOR: José Alberto Benítez Andrades y Martín Bayón Gutiérrez

TÍTULO: APICulture: Generación dinámica de servicios web con funcionalidades CRUD

TITLE: APICulture, Dynamic generation of web services with CRUD functionalities

CONVOCATORIA: Septiembre, 2022

RESUMEN:

Este proyecto consiste en el desarrollo de una aplicación web (cliente y sevidor) y del seguimiento de este empleando SCRUM. Esta aplicación permitirá al usuario la creación de otros servicios web en base a unos datos con formato fijo y unas operaciones que se podrán realizar sobre ellos (leer, crear, modificar o borrar). El servicio web creado no se generará en código para su ejecución independiente, sino que, una vez almacenada su configuración, se interpretará de forma dinámica y se servirá a través de una ruta en el servidor. El usuario podrá configurar a través de la interfaz gráfica qué tipos de datos quiere almacenar en su servicio y qué operaciones puede realizar sobre ellos. Además, se podrá restringir el acceso a determinadas operaciones de la API mediante el uso de tokens. Una vez desplegada la API, el usuario podrá realizar las operaciones.

ABSTRACT:

This project consists in the development of a web application (client and server) and its planification using SCRUM. This application will allow the user to create other web services based on data with a fixed format and operations that can be performed on them (read, create, modify or delete). The new web service will not be generated in code for its independent execution, but, once its configuration is stored, it will be dynamically interpreted and served through a route on the server. The user will be able to configure through the graphical interface what types of data he wants to store in his service and what the operations that he can perform. In addition, access to certain API operations may be restricted through the use of tokens. Once the API is deployed, the user will be able to perform the operations.

Palabras clave: Desarrollo web, Servicios web, API

Firma del alumno:	**VºBº Tutor/es:**

Índice de contenidos

Índice de figuras

Índice de cuadros y tablas

Glosario de signos, símbolos, unidades, abreviaturas, acrónimos o términos

API: Application Programming Interface. Es una interfaz que nos permite a los programadores acceder a través de unas funciones a ciertas características o datos.

JSON: Javascript Object Notation. Es un estándar para representar tipos de datos en forma de objetos. Utiliza una sintaxis similar a la de los objetos JavaScript pero con algunas diferencias, como que los nombres de los campos van entre comillas. Es muy popular, posiblemente gracias a su simplicidad y similaridad con JavaScript.

SCRUM: Marco de trabajo ágil para la gestión de proyectos. El nombre viene de una jugada de rugby, donde todos los miembros del equipo de unen para conseguir un objetivo.

Product Owner: Es un rol dentro de un equipo de SCRUM, es el responsable de que entreguemos el producto correcto al cliente y trata de maximizar su valor.

Endpoint: Es la parte de una ruta HTTP que le sirven a un servidor para discriminar que parte se esta pidiendo, por ejemplo, en la ruta http://google.es/api/tiempo, el endpoint sería “/api/tiempo”

Backend: Se refiere a la parte del servidor en una arquitectura cliente-servidor. Es el código que se ejecuta en el servidor.

Frontend: Se refiere a la parte del cliente en una arquitectura cliente-servidor. Es el código que se ejecuta en el navegador web.

CRUD: Por sus siglas en ingles significa Create, Read, Update, Delete. O lo que es lo mismo: creación, lectura, actualización y eliminación. Se refiere a la posibilidad de realizar todas estas operaciones sobre algo en concreto, como puede ser una ruta de una API.

UI: User Interface por sus siglas en inglés, Interfaz de Usuario en español. Es el medio por el que el usuario puede comunicarse con nuestro sistema a través de gráficos.

IDE: Por sus siglas en inglés significa Integrated Development Enviroment, es un entorno de desarrollo, un programa que permite a un desarrollador de software componer y editar programas, además de ofrecer herramientas que faciliten este trabajo.

HTTP Query Parameters: Se refiere a una serie de parámetros que pueden incluirse al final de las URL, estos se añaden anidando un símbolo “?” al final de la URL y a partir de ahí incluyendo las queries separadas por “&”.

SPA (Single Page Application): Páginas web que están formadas por una sola página, esto es, que solo piden una página al servidor. A partir de ahí simulan la navegación entre páginas y rutas.

Tarea Asyncrona: Es una tarea que se ejecuta en otro hilo de ejecución, por lo que no es necesario que esperemos a que termine de ejecutarse.

Javascript: es un lenguaje de programación interpretado, que requiere un intérprete para ser ejecutado, y que se evalúa en tiempo de ejecución. Todos los navegadores pueden ejecutar JavaScript.

HTML: Por sus siglas Hypertext Markup Language, es un lenguaje de marcas similar a xml, se utiliza para dar formato y contenido a nuestras páginas web.

CSS: Por sus siglas en inglés Cascading Style Sheets, sirve para definir los estilos de una página web, estos estilos se aplican al HTML, o mas específicamente a la estructura de nodos que este genera. Se llama hoja de estilos en cascada por la forma en que estos se aplican.

HTTP: Hypertext Transfer Protocol, es un protocolo de la capa de aplicación.

Parsear: Es una modificación al español del término Ingles “parse” o “parsing”, que normalmente se refiere al echo de tomar una cadena de texto como entrada y transformarla en otro tipo de formato normalmente estructurado, es una forma de de estructurar la información que le llego al programa parser como entrada. En español se suelen llamar analizadores sintácticos.

Token: en el contexto general de la informática, un token es una cadena de texto que representa algo [12].

Unix time: es el número de segundos que han pasado desde el 1 de enero de 1970 a las 00:00:00 [15]. Es una marca de tiempo que nos permite representar fechas

URL: Uniform Resource Locator, es un subconjunto de URI (Uniform Resource Identifier) y es una forma de localizar un recurso de manera única en la web.

ComboBox: Es un tipo de menú gráfico que permite la selección de una opción de entre un conjunto de posibilidades. Al hacer click sobre él se despliega una lista de posibilidades, de las cuales se puede realizar una selección.

IPS: Por sus siglas Internet Provider Service, se refiere a las entidades que nos proveen de acceso a la red internet.

Popover: Es un componente gráfico de Bootstrap, este se muestra al posar el cursor sobre cierto elemento. Tiene forma de globo o bocadillo típico de los comics o caricaturas.

1. Introducción

1.1 Contexto

Muchas de las aplicaciones web sigue un esquema similar, donde un cliente, a través de un navegador realiza una petición HTTP a un servidor, este servidor, accede a unos datos y ejecuta una lógica propia de la aplicación.

Teniendo este esquema en mente, podemos darnos cuenta de que parte del esquema siempre sigue el mismo patrón, y por lo tanto abstraer esta parte podrá ahorrarnos tiempo de desarrollo, permitiendo al desarrollador centrar sus esfuerzos en la parte más variable de la aplicación.

En otras ocasiones nos es necesario implementar APIs sencillas para almacenar y consultar datos de un tipo determinado, aunque ya existen muchas plantillas y herramientas, estas tienen cierta complejidad, y muchas veces requieren de cierta implementación.

Es por esto por lo que este Proyecto gira en torno a un problema, la creación de APIs web.

Las API web son Interfaces de Programación de Aplicaciones accedidas a través del protocolo HTTP. En términos más simples: es una interfaz que nos permite a los programadores, a través de unas funciones acceder a ciertas características o datos alojados en un servidor, y todo esto se realiza usando el protocolo HTTP.

Hay distintas formas de intercambio de datos con un API, el estándar más usado hoy en día es el JSON, que tiene una sintaxis similar a la de los objetos JavaScript, de su simplicidad viene el que sea tan popular. Mi Proyecto solo está pensado para abarcar este tipo de API que usan objetos JSON en sus intercambios de datos.

1.2 Motivación

Como ya se comentó, las aplicaciones web, y de forma más general, las aplicaciones que requieren de algún tipo de servicio en internet siguen un esquema similar. El cual suele ser dividido en frontend y backend.

Abstrayendo buena parte del backend podremos ahorrar tiempos de desarrollo, y nos permitirá centrarnos a los desarrolladores en las partes de la aplicación que mas varíen en ese caso. Abstrayendo el backend requeriremos aún de definir el modelo que van a seguir los datos, porque esto nos supone facilidad en las validaciones de estos y nos sirve como documentación futura, además de otras ventajas derivadas del uso de tipado estático.

Este proyecto fue propuesto por la empresa HP SCDS como parte del observatorio tecnológico [1], donde cada año proponen una serie de proyectos a los alumnos de distintas universidades. Dicha empresa me asignó un tutor, Guillermo Ménguez, el cual me ayudo con la parte técnica y de gestión del proyecto.

1.3 Descripción del proyecto

Este proyecto consiste en el desarrollo de una aplicación que nos permita crear APIs dinámicamente; estas APIs las podremos crear tanto por llamadas HTTP como a través de una interfaz web que podrá ser accedida por un navegador.

Posterior a la creación de la API, se podrán realizar peticiones sobre la API haciendo llamadas HTTP al servidor.

Esta pensado también que estas APIs sean fuertemente tipadas, y que por lo tanto se validen estos datos ante un esquema. El esquema de datos será definido por el usuario en el momento de creación de la API.

Estas APIs también poseerán un mecanismo de autorización opcional, de forma que solo ciertas entidades autorizadas puedan acceder.

1.4 Objetivos

Se han propuesto una serie de objetivos de cara a la realización de este proyecto. Estos objetivos son generales, no obstante, nos sirven para formular los resultados deseados, para planificar las acciones, y para orientar los procesos, sabiendo el punto hacia el que nos queremos dirigir.

Lista de objetivos:

- Creación de un servidor donde se alojen todas las APIs creadas. Este servidor será el encargado de servir las APIs creadas en un endpoint. Y también poseerá otro endpoint para crear y eliminar las APIs en sí.
- Creación de un cliente web que permita la fácil creación y eliminación de APIs. Y que muestre al usuario como realizar llamadas a su API.
- Seguir una planificación adaptable y flexible a nuevos requerimientos y cambios. El desarrollo de esta planificación se ira documentando.
- Buscar que la aplicación desarrollada aporte valor y sea útil, en el contexto de utilización de un backend para una aplicación.
- Facilitar el trabajo de los usuarios finales que vayan a hacer uso de esta aplicación, para ello se tratará de que sea fácil de usar.

1.5 Estructura del trabajo

La memoria se distribuye en 5 capítulos, se presenta a continuación un breve resumen de cada uno de ellos:

- Introducción: donde se expone el problema que se quiere resolver y cual es la motivación detrás del mismo.
- Gestión del proyecto: se define la planificación del proyecto, siguiendo la metodología concreta, y como ha ido siendo aplicada a lo largo del tiempo. También se detallan los costes y los recursos empleados. En un principio se sientan las bases de la planificación, y más adelante se explica como se ha llevado a cabo la planificación completa.
- Solución: va desde el diseño inicial de la aplicación y un artefacto abstracto de software hasta su desarrollo en unas tecnologías concretas y la muestra de ciertas partes de la implementación. Muestra la toma de ciertas decisiones con respecto al diseño final del sistema y porque ha sido implementado de esta forma
- Conclusiones y líneas futuras de trabajo
- Referencias bibliográficas

2. Gestión del proyecto

En este capítulo se busca mostrar cual fue el plan de acción del proyecto en base a SCRUM, y como este se fue implementando y desarrollando a lo largo de los sprints. También se muestran cuáles son los requisitos de los cuales partimos, los cuales son formulados como historias de usuario.

Después se muestra la gestión de recursos, con los medios que fueron requeridos para desarrollar el proyecto y una estimación de sus costes.

Y por último se incluyen una serie de aspectos para tener en cuenta en el futuro.

2.1 Planificación del proyecto

2.1.1 Metodología

Teniendo en cuenta las particularidades de este proyecto, al inicio de este se ha seleccionado Scrum como la metodología más adecuada para llevarlo a cabo. Aunque se ha adaptado con algunas particularidades que veremos en el punto 2.1.2.

La guía de Scrum [2], contiene una definición completa de Scrum, esta fue inicialmente escrita en inglés y posteriormente traducida a otros idiomas, es ampliamente reconocida y usada. Es por esto que para la definición de Scrum mostrada en este apartado me he basado en buena parte en esta guía.

Scrum es un marco ágil para la gestión de proyectos que permite a las personas, equipos y organizaciones a crear soluciones adaptables para problemas complejos.

SCUM se basa en buena parte en el empirismo y el pensamiento Lean. Según el empirismo el conocimiento viene de la experiencia y la toma de decisiones basadas en la observación. El pensamiento Lean reduce los detalles menos importantes y se centra en lo esencial. Scrum usa un enfoque iterativo e incremental.

Scrum define unos Artefactos, unos Eventos y unos roles dentro del equipo.

Los equipos de Scrum suelen estar conformados por un pequeño número de personas; dentro de estos equipos encontramos diferentes roles:

- Desarrolladores: son los integrantes que se dedican a crear funcionalidad dentro de cada sprint, además tiene las responsabilidades de: crear un plan para el sprint, adaptar su plan cada día para orientarse hacia el objetivo del sprint.

- Produc Owner (Propietario del producto): representa al usuario del producto, y su deber es el de maximizar el valor del producto resultante, también es responsable de la gestión de la pila de producto.
- Scrum Master: Es el responsable de se siga el marco Scrum tal y como es definido, pero dentro de la flexibilidad que presenta SCRUM. Ayuda a los miembros del equipo a comprender SCRUM. En muchas ocasiones este rol no es implementado.

Dentro de un equipo de SCRM no hay jerarquías.

Otra pieza fundamental para entender Scrum es el Sprint, un Sprint es un período corto (normalmente entre 2 semanas y un mes) de tiempo fijo en el que un equipo de Scrum trabaja para completar una cantidad de trabajo prestablecida [3]. Tienen una duración fija de un mes o menos, y cuando un sprint acaba, comienza el siguiente. Al final de un sprint obtendremos un incremento de producto. Dentro de un sprint no se pueden hacer cambios que pongan en riesgo los objetivos.

Dentro de un sprint podemos encontrar las siguientes reuniones o eventos:

- Planificación del Sprint: esta reunión se realiza al inicio del sprint, y en ella el equipo establece el trabajo que se realizará en este nuevo sprint
- Daily Scrum (Scrum diario): es una reunión diaria y rápida en la que se inspecciona como va el progreso hacia el objetivo del sprint y se adapta el sprint backlog según sea necesario.
- Revisión del sprint: Se lleva a cabo al final del sprint para inspeccionar que tareas se han completado y cuales no, todo el equipo de Scrum presenta sus resultados.
- Retrospectiva del sprint: Se realiza al final del Sprint y su propósito es obtener feedback de lo que fue bien y lo que fue mal en ese sprint, de cara a mejorar en el futuro.

Los Artefactos de Scrum representan trabajo o valor, son elementos que se producen como resultado de aplicar SCRUM. Son los siguientes:

- Pila del producto: es una lista ordenada de elementos que se necesitan para mejorar el producto, normalmente estos elementos suelen ser historias de usuario.
- Pila del sprint: es otra lista ordenada de historias, pero en este caso son solamente relativas al sprint, son las tareas que se tomaron de la pila del producto cuando se realizó la planificación del sprint.
- Incremento de producto: Al final de cada sprint el producto es mejorado, esto constituye un incremento en el mismo, un paso más hacia el objetivo final.

A partir de lo descrito anteriormente vemos cuales son los puntos fuertes de esta metodología y porque es la que mejor se adapta a este proyecto, y es tan utilizada

en desarrollo de software. Algunas de las ventajas que nos hacen escoger esta metodología para el proyecto son:

- La flexibilidad y adaptabilidad, con alta adaptación a la aparición de nuevos requisitos. En este proyecto, por ejemplo, no estaba del todo especificado en un inicio, se ha ido adaptando a lo que iba surgiendo como funcionalidad útil. También dado a que no teníamos plazos fijos.
- Es fácil de implementar y de seguir semana a semana.
- Mayor control de imprevistos, por ejemplo, algunas funcionalidades no estábamos seguro de cuanto tiempo nos iban a llevar.
- La claridad de cuál es el objetivo que tenemos en mente al inicio del proyecto, sin estar abrumados por tener que definir todo.

2.1.2 Aplicación de la metodología

Scrum nos ofrece unas normas básicas que son adaptables y moldeables, en esta sección se pretende mostrar la aplicación de Scrum que he llevado a cabo en este TFG, que, aunque no sigue Scrum a rajatabla, intenta seguirlo lo mas fielmente posible.

Hay que tener en cuenta que los recursos para un TFG son limitados, y que principalmente era yo el que formaba el equipo, junto con la ayuda que Guillermo Ménguez, el tutor de empresa me ofreció para la gestión y desarrollo del proyecto; es por esto que seguir una implementación que siga todas las buenas prácticas de Scrum se dificulta, ya que muchas de estas prácticas están pensadas para el trabajo en equipo. Aun así, a pesar de ser uno, muchas de las ventajas de Scrum siguen aplicando.

Los elementos que utilizaremos para la pila de producto son en su mayoría historias de usuario, aunque al principio del proyecto también se incluyeron ciertas tareas (no historias) que serviría para sentar las bases del proyecto.

Pero ¿Qué es una historia de usuario? Una historia de usuario es una representación de un requisito escrito en pocas frases y utilizando el lenguaje común del usuario. El propósito de una historia de usuario es mostrar cómo un elemento de trabajo entregará un valor particular al cliente [4].

Las historias de usuario suelen seguir la siguiente plantilla: As a “user" I want “something" and so that “some reason"; o en español: Como “Usuario” Quiero “conseguir algún objetivo” para “algún motivo”. Como vemos la plantilla está totalmente centrada en el usuario y en sus necesidades.

Las historias de usuario a veces también contienen “story points”, que son una medida en una escala relativa utilizadas para representar la incertidumbre, la complejidad y el esfuerzo necesarios para completar o implementar una historia

de usuario [5]. Además, se utilizará la escala de Fibonacci para medir estas (1, 2, 3, 5, 8, 13…).

Respecto de como incluimos todo esto en el proyecto: En una primera fase del proyecto escribimos la mayor cantidad de historias de usuario que creyésemos que abarcaban todo el proyecto. Y si nos quedábamos cortos o habíamos incluido demasiadas historias ya las iríamos modificando al final de los sprints. En caso de que estas historias fueran muy grandes, ya las iríamos dividiendo en otras más pequeñas según vayamos avanzando.

Hemos escogido la duración de 2 semanas para los sprints.

Haremos 2 reuniones por sprint:

- Planificación del Sprint: al inicio del sprint, para ver que historias de usuario incluiremos para ser realizadas en ese sprint. Además, si alguna de esas historias es muy grande la dividimos en varias.
- Revisión del sprint: al final del sprint, donde vemos un poco que se ha completado, si alguna historia asignada a este sprint no pudo ser completada se pasa para el siguiente sprint.

  En esta parte, si vemos que se requiere crear alguna nueva historia de usuario porque el proyecto la requiere, la podríamos crear. Y si vemos que alguna historia de usuario requiere de ser eliminada porque ya no tiene lugar también podíamos eliminarla. O si alguna historia requiere ser reformulada también podíamos modificarlas en este momento.

Estas 3 reuniones las juntaremos en un solo meeting, que se realizara cada 2 semanas. Excepto la primera reunión del primer sprint y la última reunión del último sprint, que solo tendrán 2 meetings.

En las reuniones estaremos yo y Guillermo Ménguez, el tutor por parte de la empresa. El hacia un rol similar al del producto Owner, y también me ayudaba en la parte tecnológica si tenía alguna duda.

Por lo que los roles del equipo quedan así:

- Desarrollador: Yo, Sergio Chimeno
- Product Owner: Guillermo Ménguez, el tutor de la empresa, y quien propuso el trabajo; por lo tanto, era el que estaba interesado en maximizar el valor del producto.

2.1.2.1 Herramienta de apoyo

Para llevar a cabo todo el proyecto nos hemos apoyado en la herramienta Gitlab [26].

Gitlab es una aplicación utilizada para almacenar en la nube proyectos basados en el controlador de versiones GIT. Gitlab posee además una interfaz gráfica accesible por web que nos ofrece muchas otras aplicaciones relacionadas con el manejo de nuestro proyecto.

Usamos gitlab para manejar todo el tema de SCRUM, de las tareas, los sprints, porque tiene integradas a través de su interfaz web herramientas muy útiles para esto.

También hice uso del sistema de control de versiones GIT, y usamos un repositorio remoto de gitlab para llevar el proyecto al día. Este repositorio puede encontrarse en: https://gitlab.com/HP-SCDS/Observatorio/2021-2022/apiculture, aunque este es privado debido a que pertenece a HP SCDS, la empresa con la que realicé el proyecto.

2.1.3 Los 2 primeros sprints

Al principio del proyecto tuve una serie de tareas que me asigno Guillermo Ménguez, no contienen ninguna historia de usuario en sí, son mas bien tareas a partir de las cuales depende el todo resto del proyecto, y que sientan los pilares de este. Las incluimos dentro del Scrum, pero podían no haber sido incluidas, en algunas de ellas no se consigue ningún incremento de producto como tal.

A continuación describo estas tareas.

Ver en profundidad el prototipo y entender como funciona: esta tarea era para familiarizarme con el proyecto y entender que es lo mínimo que buscaba lograrse con él. Guillermo Ménguez había hecho un pequeño backend de prototipo en C# para los alumnos que hicieron este proyecto. Fue completado en el Sprint 1.

Technological ramp-up: esta tarea tenía 2 partes en verdad, la primera era seleccionar las tecnologías pilares que usaría para todo el proyecto, del lado del servidor fueron Nodejs con Typescript y Mongodb como base de datos. Del lado del cliente elegí Angular.

La segunda parte de esta tarea consistía en hacer una “rampa de aprendizaje” en estas tecnologías, ya que de Angular y Typescript no sabía nada prácticamente. Para finalizar esta rampa de aprendizaje se creó una aplicación CRUD rápidamente, esta aplicación era para manejo de empleados, y además tenía un login y un registro de usuarios. El CRUD en sí puede verse en la Figura 2.1.

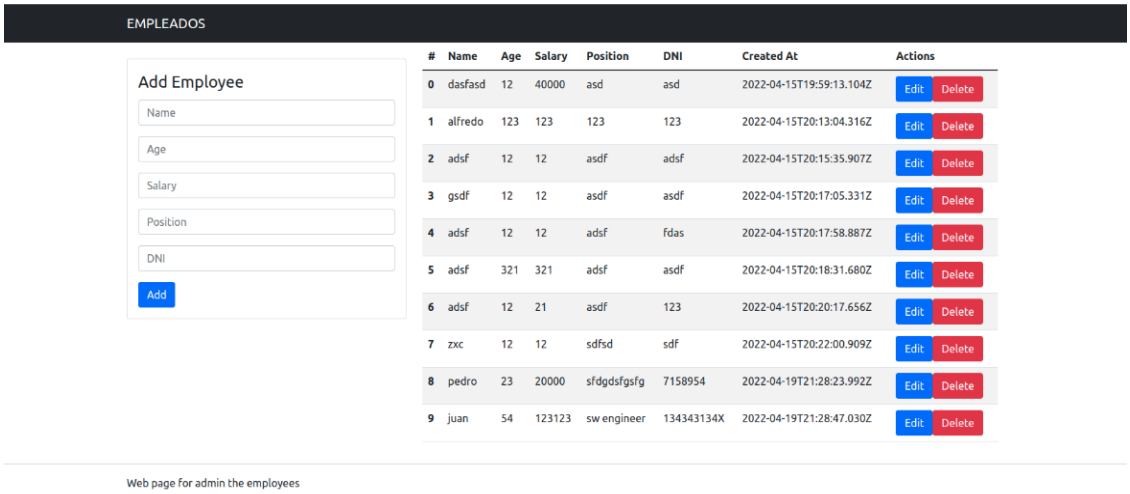

Figura 2.1. Aplicación CRUD simple

Esta tarea me llevo el Sprint 1 y 2.

Fill the backlog (o rellenar la pila de producto)**:** Consistía en crear historias de usuario de lo que preveía que sería el producto final para el cliente, de manera que se rellenase la pila de producto, y tener ya algo para empezar el proyecto a partir del Sprint 3. Esta tarea se completó en el Sprint 2.

2.1.4 Pila del producto

En este apartado se presenta en una tabla el estado de toda la pila del producto al final del último Sprint. Para cada historia se indica sprints los en los que se llevó a cabo, los story points que se le asignaron y la descripción de la historia.

Además, al final se indicarán algunas notas para dar contexto a algunas historias, ya que varias de estas historias se fueron añadiendo a lo largo del proyecto, no todas estaban desde un inicio. Fueron surgiendo varias nuevas debido a bugs o a posibles mejoras que el product owner me iba comentando al final de los sprints.

# ID	Historia de Usuario	Story Points	Sprints
1	Como cliente quiero crear nuevas APIs	13	4,5,6
2	Como cliente quiero poder registrarme, para poder tener una cuenta	3	3
3	Como cliente quiero poder logearme, para poder crear mis APIs y ser el único que las puede borrar	8	3,4

4	Como cliente quiero poder elegir el formato de los datos de mi API	8	4,5,6
5	Como cliente quiero ser capaz de guardar y recibir datos de mis APIs en formato JSON	3	4,5,6
6	Como cliente quiero tener la opción de eliminar mi API si lo requiero	3	6
7	Como cliente quiero tener la posibilidad de listar todas mis APIs	5	5
8	Como cliente quiero que se me permita actualizar el nombre de mi API	8	6,7
9	Como cliente quiero poder listar todos los datos de un API concreto para poder ver todos mis datos almacenados en ese API (GET)	3	6
10	Como cliente quiero poder realizar una operación para guardar datos en mi API (POST)	5	6
12	Como cliente quiero poder actualizar datos dentro de mi API (PUT)	5	6
13	Como cliente quiero poder eliminar datos dentro de mi API (DELETE)	3	6
14	Como cliente quiero poder establecer restricciones sobre algunos campos de los objetos JSON, de manera que pueda indicar si estos campos son únicos a lo largo de todas las instancias de objetos dentro de la API	3	6
15	Como cliente quiero poder establecer restricciones sobre algunos campos de los objetos JSON, de forma que pueda indicar si un campo de un objeto puede ser vacío o nulo	3	6
16	Como cliente quiero poder decidir que operaciones son permitidas de realizar contra mi API	5	7
17	Como cliente quiero tener la posibilidad de que algunas de las operaciones de mi API requieran autenticación por token	8	8,9
18	Como cliente quiero poder decidir si una operación requiere autenticación o si la puede realizar todo el público	3	8,9
19	Como cliente quiero tener la capacidad de editar las operaciones que se pueden realizar sobre mi API y el echo de si estas requieren o no autorización.	8	8,9

20	Como cliente quiero tener la capacidad de hacer todo esto a través de llamadas a APIs y a través de una interfaz de usuario.	8	8
21	Como cliente quiero tener algo de ayuda para saber como puedo llamar a mis APIs	3	8
22	Como cliente quiero ser redirigido a “mis APIS” después de crear una API	1	8
23	Como cliente quiero tener un mensaje descriptivo cuando trato de crear una API con un nombre duplicado	1	8
24	Como cliente quiero tener una mejor experiencia con la interfaz de usuario	5	8
25	Como cliente quiero tener la capacidad de crear nuevos tokens de autenticación	5	9
26	Como cliente quiero poder hacer inválido a un token ya existente, de forma que ya no sirva como autorización ese token	3	9
27	Como cliente quiero tener la posibilidad de marca mi API como pública para que otros usuarios la puedan encontrar y se pueda crear así una comunidad	3	9
28	Como usuario tanto registrado como no registrado quiero poder ver la lista de APIs públicas de forma que pueda usarlas	5	9
29	Como cliente quiero ver la URL correcta en la UI para poder llamar al backend	2	9
30	Como cliente quiero que se me explique como usar los tokens en la UI para así saber como formar yo mis propias peticiones al servidor	2	10
31	Como cliente quiero que se oculte el botón y el combobox en el formulario de crear un API, cuando todas las operaciones ya han sido añadidas a mi API	2	10
32	Como cliente quiero poder ver cuantas veces han sido consumidas mis APIs por día, para hacerme una idea de su popularidad	5	10

Tabla 2.1. Pila del producto

A continuación paso a hacer unos comentarios sobre algunas de las historias de usuario:

- Como cliente quiero ser redirigido a “mis APIS” después de crear una API (Historia 22): esta historia fue propuesta por el Product Owner como mejora de la experiencia de usuario, ya que, en un principio, tras crear una API, la aplicación seguía mostrando la misma pantalla
- Como cliente quiero tener algo de ayuda para saber como puedo llamar a mis APIs (Historia 21): también fue propuesta por el Product Owner, en un principio, al llamar a las APIs desde la interfaz gráfica no había nada de documentación.
- Como cliente quiero tener un mensaje descriptivo cuando trato de crear una API con un nombre duplicado (Historia 23): Fue propuesta por el Product Owner, antes de realizar esta historia, si en el formulario de creación de APIs se seleccionaba un nombre duplicado el mensaje de error mostrado era muy genérico, con esta historia se pretende que este mensaje sea descriptivo para facilitarle el trabajo al usuario.
- Como cliente quiero ver la URL correcta en la UI para poder llamar al backend (Historia 29): En un momento, el puerto del backend fue cambiado, pero no en la parte visible al usuario; el Product Owner se dio cuenta de este hecho y lo comentó, además propuso el uso de APP_INITIALIZER de Angular, de forma que las constantes de configuración de la aplicación se guarden en un archivo json que se sirve estáticamente, y que este sea pedido por la aplicación cliente al inicio.
- Como cliente quiero que se me explique como usar los tokens en la UI para así saber como formar yo mis propias peticiones al servidor (Historia 30): El objetivo de la UI es facilitarle el trabajo al usuario, y al momento de crear tokens desde la UI no estaba explicado el como usar estos, o donde se debían de incluir cuando se vaya a hacer una llamada al backend. Es por esto que se decidió añadir una pequeña explicación al usuario dentro de la interfaz de usuario.
- Como cliente quiero que se oculte el botón y el combobox en el formulario de crear un API, cuando todas las operaciones ya han sido añadidas a mi API (Historia 31): Esto fue propuesto por el Product Owner, es un pequeño detalle que mejora la experiencia de usuario. Esto es debido a que en el formulario para crear APIs hay un combobox y un botón para añadir operaciones a una lista; y en un principio, si el usuario añadía todas las operaciones, el combobox quedaba vacío, por lo que realmente no tenía sentido que se siguiera mostrando.

2.2 Recursos y estimación de costes en desarrollo

En este apartado se indican los recursos necesarios para realizar el proyecto, y una estimación de su coste en caso de ser requerido. Finalmente se mostrará un presupuesto total.

Para el cálculo de estos costes hay que tener en cuenta que la duración del proyecto son 5 meses (10 sprints)

2.2.1 Recursos hardware

Por la parte de hardware necesitamos un equipo de desarrollo con ratón y con conexión a internet. Además de tener en cuenta los costes energéticos tanto del dispositivo como de la iluminación.

En la tabla 2.2 se detallan estos gastos.

Recurso	Precio Unitario	Unidades	Importe
Portátil HP Elitebook 840 G3	470 €	1	470 €
Ratón Logitech B100	10 €	1	10 €
conexión a internet Vodafone Fibra 600Mpbs	31 €	5	155 €
Factura eléctrica	50 €	5	259 €
TOTAL			894 €

Tabla 2.2. Costes de hardware

El precio unitario de la conexión a internet y de la factura eléctrica son mensuales, por eso se requieren 5 unidades.

2.2.2 Recursos software

Respecto al software la mayoría de este es gratis, ya sea porque se utilizan las versiones gratuitas del mismo o porque tienen licencias de software libre. A continuación, se pasa a detallar este software:

- Visual Studio Code: El IDE usado para el desarrollo del proyecto. Tiene una versión gratuita.

- Windows 10: Sistema operativo del portátil, este ya viene incluido en el precio del portátil, por lo que no lo tendremos en cuenta en el presupuesto.
- Nodejs y npm, ademas de ciertos paquetes en npm, todos los que se han usado son software libre
- Angular, bootstrap para la parte del frontend, contienen licencias MIT, por lo que saldría gratis también.
- MongoDB, para la base de datos, la versión community, que es gratis, permitida para uso comercial. Pero también ser
- Microsoft 365 será necesario para redactar la memoria, los documentos y las realizar las reuniones del equipo

Recurso	Coste Total (bruto)
Licencia Microsoft 365	99 €

Tabla 2.3. Costes de software

2.2.3 Recursos humanos

El equipo ha sido integrado por un desarrollador y un Product Owner, se pasa a detallar el coste en base a las horas dedicadas en la siguiente tabla

Empleado	Coste/Hora (bruto)	Horas	Coste Total (bruto)
Desarrollador Junior	9.4 €	300	2820 €
Product Owner	14 €	20	280 €
TOTAL			3100 €

Tabla 2.4. Costes de personal

Aquí no hemos incluido los impuestos, pero probablemente habría que añadirle alrededor de un 20% al coste bruto.

2.2.4 Presupuesto

En la tabla 2.5 está indicado el presupuesto completo para el desarrollo de este proyecto.

Para calcular el presupuesto se han sumado los costes de los Recursos Software, Hardware y Humanos; y a este coste se le ha añadido el beneficio industrial, que en este caso lo hemos fijado en el 13% del coste total.

El beneficio industrial es el beneficio económico que obtienen los constructores del proyecto [28].

Recurso	Coste (bruto)
Recursos hardware	3100 €
Recursos software	99 €
Recursos humanos	894 €
Coste Total	4093 €
Beneficio industrial (13%)	532.09 €
Total	4625.09

Tabla 2.5. Presupuesto

2.3 Aspectos a tener en cuenta

En este subapartado incluimos aquellos aspectos que deberemos tener en cuenta de cara al futuro del proyecto, para evitar posibles problemas que puedan surgir.

Idealmente habríamos incluido estos aspectos dentro de este proyecto, pero el tiempo para realizar un TFG es limitado, y decidimos desde un inicio que esto quedaría fuera del alcance del proyecto, y nos centraríamos en realizar la parte funcionalidad. Otra razón para no incluir estos aspectos es que la empresa con la que realicé el proyecto no me los pidió, y es posible que estos ya los tenga en cuenta la empresa misma de cara al futuro, al menos en caso de utilizar la aplicación de forma interna.

2.3.1 Posibles recursos futuros requeridos

Para mantener la aplicación a futuro también habría que tener en cuenta estos puntos:

- Se necesitaría de los servicios de alguien que vaya solucionando los bugs que aparezcan.
- Debería tenerse en cuenta la posible escalación a futuro, por ejemplo, si aumenta sustancialmente el número de usuarios.

- El coste mensual de mantener los servidores donde estaría desplegada la aplicación.
- Necesitaríamos algún tipo de personal que se comunique con los clientes y descubra que fallos tiene la aplicación, o que posibles mejoras requieren los clientes.
- Necesitaríamos contratar a algún experto en el apartado legal para que la aplicación cumpla con la legalidad.

No pretendo explayarme más en este apartado, ya que queda fuera del alcance de este TFG.

2.3.2 Aspectos legales

Para esto lo ideal sería contratar a un experto en este campo, con el que trabajaríamos de cara a que la aplicación cumpla todas las normas establecidas.

El aspecto legal principal a tener en cuenta es con respecto a las leyes de protección de datos personales. Para esto tenemos dos Leyes, la Ley Orgánica de Protección de Datos (LOPD) [30] y la Reglamento General de Protección de Datos (RGPD) [29]. La primera es de ámbito español y la segunda de ámbito europeo. La LOPD se puede ver como una adaptación de la directiva Europea a la Ley Española.

Tendremos que tener en cuenta que nuestra aplicación hace uso de datos personales, en el registro del usuario pedimos su nombre, email y contraseña.

Además, como un usuario puede crear APIs, lo cual es muy general, podría estar creando, por ejemplo, una API de clientes, y esta a su vez contendría datos personales. Por esta parte, creo que podríamos asegurar ciertos aspectos legales generales y a partir de ahí, delegar la responsabilidad en el usuario, haciendo que seleccione una casilla o similar, donde quede constancia de que nosotros no nos hacemos cargo de esa parte, y donde se acepten nuestras condiciones.

Para los formularios que incluyan información personal tendremos que incluir también una casilla de verificación, donde el usuario acepte, y le hagamos saber que datos estamos recogiendo.

Además, sería necesario añadir una página que contenga el aviso legal, otra para la política de privacidad y tal vez otra para los términos y condiciones.

Reitero que esto deberá de ser comprobado por un experto en la materia, a fin de evitar posible problemas legales.

3. Solución

En este capítulo se pretende explicar como es el desarrollo del sistema desde un punto de vista técnico, se mostrarán sus partes principales, como se han constituido, que patrones se han seguido y el porqué se han tomado ciertas decisiones. Además se mostrarán algunos detalles de implementación y el resultado final.

3.1 Visión general

Partimos de que el sistema tendrá un frontend y un backend, y que los usuarios podrán tanto contactar con el backend como con el frontend.

Frontend y Backend es una forma de dividir funciones entre 2 subsistemas. En este caso el frontend se refiere a la parte que se estará ejecutando en el navegador del cliente, y por lo tanto es responsable de mostrarle los elementos visuales con los cuales puede interaccionar. Por otro lado, el backend se refiere a la parte que se estará ejecutando en un servidor remoto, esta contiene toda la lógica de la aplicación y los datos, con esta se puede contactar a través de llamadas a la API.

Lo que provee la principal funcionalidad al usuario es el backend, pero el frontend es una parte esencial que facilita mucho el uso de la aplicación y ahorra tiempo.

Resumen de funcionalidades implementadas tanto en el backend como en el frontend:

- Login y registro de usuarios
- Creación de APIs, Modificación de APIs (A excepción de su esquema de datos), eliminación y lectura.
- Para una API en concreto se puede realizar las siguientes operaciones: GET, POST, PUT, DELETE, o lo que es lo mismo: obtener datos, subir datos, actualizar datos y eliminar datos.
- Dada una API existe la posibilidad de autorización mediante tokens. Por esto, también se necesita la posibilidad de crear tokens.
- Almacenamiento de veces al día que se consume una API

3.2 Tecnologías

El desarrollo del proyecto ha sido en base a unas tecnologías, las principales han sido escogida al principio del proyecto, y luego se han ido añadiendo ciertos paquetes que podrían brindar utilidad para una tarea concreta.

Como lenguaje de programación en un principio se pensó en escoger JavaScript, para que tanto el cliente como el servidor tuviesen el mismo lenguaje, además ese un lenguaje muy flexible y para el que existen muchas librerías. Pero tras cierta reflexión se consideró que sería mejor usar TypeScript, porque nos puede ofrecer todas las ventajas de JavaScript y más, ya que es un superconjunto de JavaScript.

TypeScript es un lenguaje de programación libre desarrollado por Microsoft. Es un superconjunto de JavaScript, que principalmente añade tipos [6]. Estas son algunas de las razones por las que elegí TypeScript en lugar de JavaScript:

- Soporte del IDE: al tener tipado estático, es posible resolver los tipos sin tener que ejecutar el programa, esto hace que el IDE nos pueda ofrecer características como autocompletado, navegación dentro del código (ya que puede inferir donde esta definido un tipo), sugestiones, o simplemente decirnos si tenemos errores en tiempo de compilación, ya que estamos asignando a un tipo de datos un valor incorrecto.
- Tipado estático: ya mencionado antes, aunque este es opcional, la idea es usarlo siempre que podamos, ya que hace nuestros proyectos más robustos. De acuerdo con el tipado estático que nos ofrece TypeScript una variable de un tipo no puede cambiar su tipo y solo puede tomar un conjunto de valores durante su tiempo de vida.
- Legibilidad: El incluir tipado estricto hace al código más expresivo, y es posible ver la intención del código escrito más fácilmente. Esto puede ser muy útil para un programador nuevo que se una al proyecto.
- Soporta mejor el paradigma de orientación a objetos con respecto a JavaScript, añadiendo interfaces, por ejemplo.

Por último, hay que comentar que TypeScript se transpila a JavaScript y lo que realmente acaba ejecutándose es JavaScript, esto permite que las librerías de JavaScript puedan ser usadas también por TypeScript.

He decidido utilizar un lenguaje común para todo el proyecto, pero también esta la posibilidad de utilizar un lenguaje en backend y otro en frontend.

El resto de las tecnologías que constituyen el servidor son:

- Nodejs como entorno de ejecución de JavaScript.
- Yarn como gestor de dependencias de JavaScript
- Express como framework de desarrollo web, que nos permite crear APIs

- MongoDB como base de datos. Es un sistema de base de datos NoSQL, orientado a documentos. En un principio elegimos esta tecnología porque no conocíamos del todo bien las estructuras de datos del proyecto. Pero en caso de requerirse se puede hacer el cambio a una base de datos SQL como como mySQL, al estar las capas de la arquitectura de esta aplicación bien separadas, solo habría que reescribir el parte del modelo. MongoDB también nos provee de atlas, una base de datos alojada en la nube.
- Mongoose como modo de conexión con la base de datos MongoDB.
- Zod, como librería que nos permite declarar esquemas de datos y posteriormente validar nuestros datos contra este esquema. Útil para validar que los datos de las APIs siguen un esquema.
- Uuid, como librería para generar UUID. Estos podemos utilizarlos como tokens de acceso. Para los tokens de las APIs.
- Jsonwebtoken para la generación y la validación de tokens de usuario, estos los usamos solo para los tokens de las sesiones de usuario.
- Api-query-params: potente librería para parsear complejas queries que vienen en la URL, una vez pareadas las transforma en objetos. Estas queries pueden contener operadores como los de comparación, igualdad, de acceso a campos de un objeto…
- Config: nos permite definir una serie de parámetros de configuración en archivos. Luego estos parámetros pueden ser cambiados simplemente en el archivo, no requiriendo modificar el código de la aplicación
- Pino y pino-pretty: dos módulos para mostrar logs por pantalla con distintos niveles de log, formato y otras características.

Por el lado del cliente tenemos otro conjunto de tecnologías:

- HTML y CSS: Estos siempre están presentes en tecnologías web del cliente, HTML es el lenguaje de marcado que indica el formato de la página web y CSS sirve para dar estilo a los elementos declarados en el HTML
- Angular: framework desarrollado en TypeScript, que se utilizar para crear páginas web SPA (Single Page Application), aplicaciones de una sola página. Angular posee gran variedad de características que nos ayudan a aumentar nuestra productividad, como son la división del proyecto en componentes independientes y reutilizables, interpolación de variables, binding de variables en TypeScript con elementos del HTML. El binding y la interpolación son muy útiles, ya que nos permiten que nuestro código en TypeScript interaccione de una forma muy sencilla con el HTML de la página web. Sin este binding tendríamos que usar otras cosas como la librería document de JavaScript o jQuery.
- Bootstrap: Es un framework basado en HTML, CSS y JavaScript, el más popular para crear páginas web responsivas, esto es, que se adaptan al

tamaño de la pantalla del dispositivo. La principal utilidad de Bootstrap es con respecto a la parte visual de la página web, ofreciendo diferentes estilos, layouts, componentes estilizados, iconos… De forma que no tengamos que crear nosotros todos estos estilos. Además contiene una extensiva documentación llena de ejemplos de uso.

También hemos usado otro tipo de tecnologías auxiliares para facilitar el desarrollo del proyecto:

- Visual Studio Code: utilizado como editor de texto, y usando algunos de sus plugins para resaltar la sintaxis y para ayudarnos de su autocompletado y sugerencias.
- Postman: es una aplicación de escritorio que sirve como cliente web, nos permite realizar llamadas a al servidor, permitiéndonos configurar todos los parámetros de una petición HTTP, como son las cabeceras, ruta, cuerpo… Una vez parametrizadas estas llamadas, podemos guardarlas y organizarlas en carpetas. También tiene herramientas que permiten documentar APIs a través de ejemplos.
- GitLab: usado como repositorio de GIT en la nube, además de utilizar las herramientas que tiene accesibles a través de su interfaz web para organizar la planificación de proyectos.

3.3 Arquitectura del sistema

En este apartado se muestra la arquitectura del sistema, y la de sus partes. De esta forma podemos dividir el sistema en varias partes, cada una con su responsabilidad, y así facilitar las abstracciones.

Este proyecto está basado en una arquitectura cliente-servidor, que es el modelo que sigue la web.

La arquitectura cliente-servidor es un estilo de diseño de software en el que las tareas se reparten entre los proveedores de recursos o servicios, llamados servidores, y los demandantes, llamados clientes [7]. Dentro de este marco, el programa cliente realiza peticiones al programa servidor, y este le contesta con una respuesta.

Para que exista esta comunicación se necesita tener un canal de comunicación entre estos dos puntos, en el caso de producción la comunicación ocurriría a través de internet, teniendo nuestro servidor desplegado en algún sitio de la nube, como AWS y teniendo el usuario cliente conexión a internet a través de su IPS.

Además, esta comunicación se lleva a cabo a través de un protocolo, en este caso HTTP, al tratarse de la web. Este protocolo es conocido tanto por el cliente como por el servidor.

En este caso el cliente es un navegador web, que ejecuta el HTML, CSS y JavaScript que nosotros le pasamos desde el servidor, así que el código del cliente podemos actualizarlo en cualquier momento realmente sin necesidad de que el usuario descargue o actualice nada nuevo.

A su vez el cliente y el servidor tendrán sus arquitecturas propias.

3.3.2 El protocolo HTTP

HTTP, por sus siglas en inglés Hypertext Transfer Protocol, es el protocolo de utilizado en el intercambio de datos en la web, y sigue la arquitectura cliente-servidor mencionada antes.

Clientes y servidores se comunican intercambiando mensajes. Los mensajes que envía el cliente, normalmente un navegador Web, se llaman peticiones, y los mensajes enviados por el servidor se llaman respuestas [10].

Un cliente manda una petición a un servidor y este le responde con una respuesta, una representación de esto puede verse en la Figura 3.1.

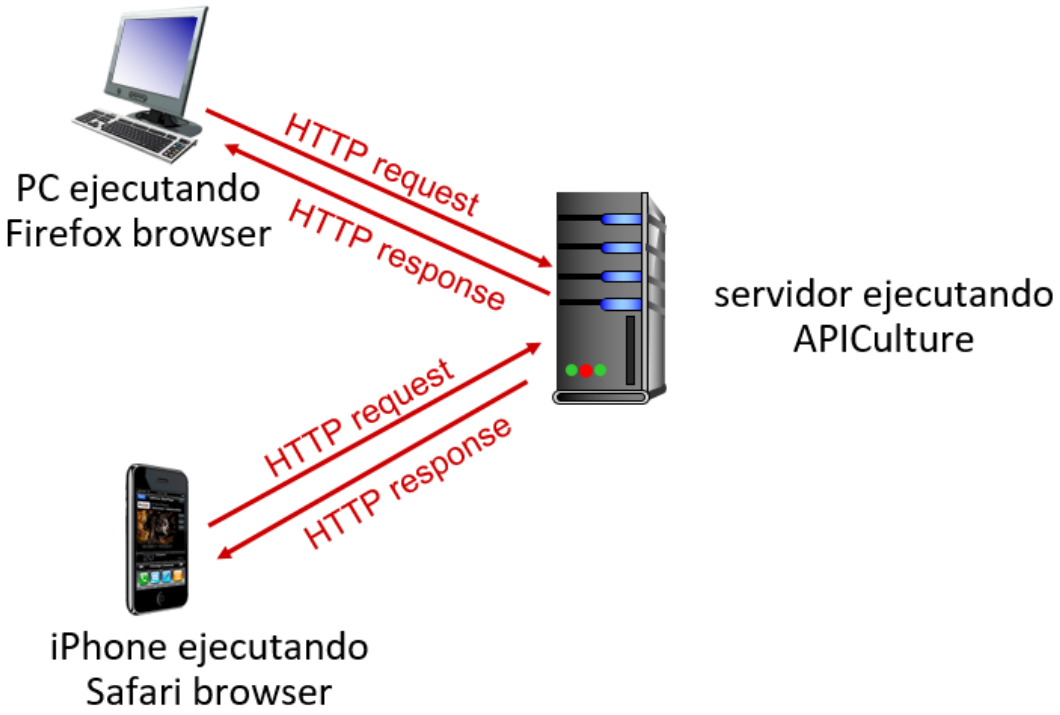

Figura 3.1. Modelo de comunicación utilizando HTTP [19]

Tanto peticiones como las respuestas HTTP siguen una estructura formada por varios campos, que está definida en el protocolo. La estructura de una petición y

una respuesta poseen diferencias. A continuación, se muestran los campos de una petición:

- Un verbo, como GET, POST, PUT, DELETE… Se pueden usar como mas se desee, pero hay una convención en torno a ellos debido al significado semántico que tienen.
- Una URL: la cual contiene el dominio, opcionalmente el puerto y una ruta del servidor. La ruta especifica un endpoint. Además, al final de la URL se puede incluir un símbolo "?" e incluir parámetros de consulta (definidas como HTTP Query Parameters en el glosario).
- Cabeceras: lista de pares clave, valor; cada clave de la cabecera tiene un significado distinto y por lo tanto actuará de una forma distinta. Algunas de las usadas en este proyecto son: Authorization para incluir el acess token, Content-type para incluir de que tipo es el contenido del cuerpo y X-refresh para el refresh token.
- Cuerpo o body: En este campo se pueden incluir caracteres en el formato que se quiera, normalmente solo usamos este campo cuando hacemos peticiones PUT o POST.

Estructura de una respuesta HTTP:

- Estado o Status: es un número decimal de 3 dígitos, cada código de estado tiene un significado establecido por el protocolo HTTP. Nos suele indicar si nuestra petición tuvo éxito o no.
- Cabeceras: Esta parte es igual que en las cabeceras de las peticiones HTTP, pero algunos de las posibles claves serán distintas.
- Cuerpo o body: En caso de que la respuesta nos tenga que devolver más información, esta irá aquí, como por ejemplo al pedir un archivo o algún dato del servidor en formato JSON.

La estructura de las peticiones y respuestas HTTP están mostradas con mayor detalle en la guía "Generalidades del protocolo HTTP" de Mozilla [10].

Paso a nombrar los Verbos de petición HTTP que he usado en este proyecto, y que normalmente son los más utilizados:

- "GET" le pide al servidor retornar toda la información identificada por la URL
- "POST" le pide al servidor que añada un nuevo recurso que se incluye en el cuerpo de la petición
- "PUT" le pide al servidor que actualice el recurso identificado por la URL, con los datos que se proveen en el cuerpo
- "DELETE" le pide al servidor que elimine los recursos identificados por la URL

Estos verbos y otros pueden encontrarse en el apartado “Method definitions” de la web del World Wide Web Consortium (W3C) [27], que son los que generan estos estándares.

Estos son los status de las respuestas HTTP que he usado en este proyecto:

- 200 (OK) significa que la petición ha tenido éxito.
- 400 (BAD REQUEST) significa la consulta tenía algún error de sintaxis.
- 403 (FORBIDDEN) significa que el cliente no tiene acceso a el recurso solicitado.
- 404 (NOT FOUND) el recurso que se pido al servidor no ha sido encontrado.
- 409 (CONFLICT) error en la petición debido a un conflicto con el estado del servidor, normalmente pasa cuando hacemos un POST de un recurso que ya existe y que debería de ser único.

Pueden encontrarse estos status de respuesta y otros definidos en el apartado “status code definitions” de la web del W3C [11].

3.3.3 Arquitectura del cliente

El cliente sigue un formato de SPA (Single Page Application) o aplicación de página único, esto es que la página web realmente está formada por una sola página que se descarga una única vez desde el servidor al cliente, es decir, los archivos en HTML, CSS y JavaScript solo se descargan una vez. La vez que se descargan es con la primera petición que el usuario hace al servidor, y a partir de ahí se mantienen en el navegador.

Realmente en este proyecto no he usado una SPA pura, porque la parte que puede ser vista por usuarios ya logeados he decido cargarla una vez se logee el usuario. Esta decisión ha sido debido a que muchos usuarios entrarán a la página solo para verla, pero es posible que no se registren ni logeen. Entonces realmente estaría formada por 2 páginas. La parte de retrasar la carga de una zona de la aplicación en angular se conoce como “lazy loading”.

La principal ventaja que nos ofrecen las SPA es la reducción de las comunicaciones entre el cliente y el servidor, al cargarlo todo de una vez. Esto es beneficioso, porque:

- Reducimos la carga sobre el servidor
- Reducimos los tiempos de navegación en el cliente, al tomar estas comunicaciones a través de la red varios milisegundos generan un posible cuello de botella en cuanto a rendimiento.

Debido al uso de Angular el cliente sigue una arquitectura MVVM, por sus siglas en ingles Model-View-ViewModel.

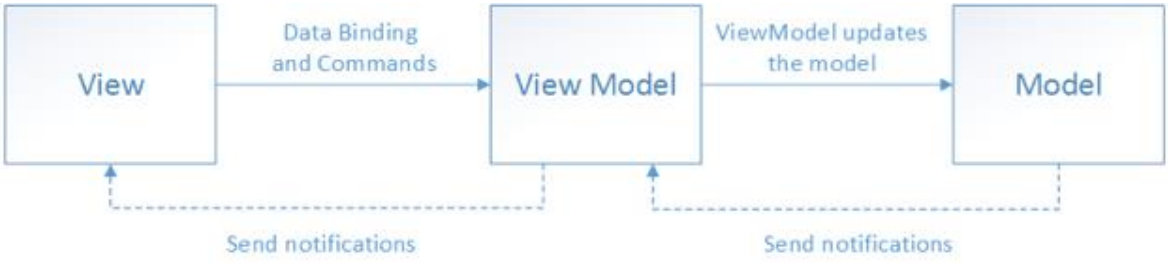

Figura 3.2. Interacción entre componentes del patrón MVVM [24]

En el patrón MVVC hay 3 componentes, como se puede ver en la imagen 3.2, el Modelo (Model), la Vista (View) y el modelo de la vista (ViewModel).

Este patrón nos permite separar los datos de la interfaz, pero en vez de haber un controlador que nos permita controlar los cambios en la vista o en el modelo (como en el patrón MVC), estos se actualizan directamente. En Angular tenemos diferentes tipos de binding que nos permiten hacer esta actualización de los datos o de la vista en una dirección (del modelo a la vista), en la contraria (de la vista al modelo), o en ambas a la vez. El tipo de actualización más utilizado en angular es el two way data binding, o actualización en ambos sentidos.

Definición de los 3 componentes del MVVM:

- Vista: Define la apariencia y la estructura de lo que se ve en la pantalla
- Modelo: Objetos o variables que encapsulan los datos en el lado del cliente
- Modelo de la vista (ViewModel): Es responsable de automatizar la comunicación entre la vista y el modelo.

A parte de este patrón, alojado en los componentes de angular, tendremos servicios, los cuales son una abstracción. Estos servicios son los que se comunicaran con la API del backend y poblarán los datos del modelo para que se ejecute la lógica correcta en el cliente.

3.3.4 Arquitectura del servidor

Para la realizar la arquitectura inicial del servidor me he basado en un post [8], que además contiene un video de un programador con gran experiencia en construir APIs con las tecnologías que uso en este proyecto.

Figura 3.3. Arquitectura del servidor visto desde una perspectiva de como fluyen las peticiones HTTP [8]

La arquitectura está basada en la forma en la que vamos a procesar las peticiones HTTP, ya que al fin y al cabo lo principal que va a hacer el servidor es dar respuestas a estas peticiones HTTP que le lleguen desde un cliente.

La petición HTTP nos llegará a nuestro endpoint (puede verse la definición de endpoint en el glosario), luego pasaría por una serie de middlewares, en caso de haberlos para esa ruta, si pasa con éxito por los middlewares, la petición llegará a nuestro controlador.

El Controlador ejecutará algún tipo de lógica, puede que requiera que los Servicios realicen alguna tarea, por lo que los llamaría. Los Servicios se comunicarán con los modelos, y estos con la base de datos para realizar operaciones CRUD.

Cuando los Servicios acaben sus tareas, llamarán al controlador y este acabará de formar la respuesta HTTP y la retornará.

Es posible que los servicios realicen alguna tarea asíncrona a la cual el controlador no requiera la necesidad de esperar por su resultado.

3.4 Implementación

En este apartado y sus subapartados se pretende mostrar detalles de implementación del proyecto, atendiendo a la arquitectura mencionada en el punto anterior.

Primero se ofrecerá una visión general del servidor, y posteriormente se verán las partes más importantes de la implementación de ciertas funcionalidades establecidas en las historias de usuario. También se mostrarán aquí varias de las pantallas de la página web, puesto que son producto de una implementación concreta.

3.4.1 Visión general del servidor

Puesto que el cliente web al fin y al cabo está haciendo peticiones HTTP usando ajax al servidor, primero vamos a mostrar un resumen de las rutas del servidor.

En la Figura 3.4 se puede ver un resumen de las rutas, podemos ver los middlewares por los que estas pasan, hasta llegar a un Handler, que sería la función del controlador encargada de devolver la respuesta. El Handler es una función dentro del controlador que se encarga de manejar una ruta, y que puede a su vez hacer uso de servicios, que son los que pueden interactuar con el modelo.

```
30  export default function routes(app: Express){
31
32      // Users
33      app.post("/users", validateResource(createUserSchema), createUserHandler);
34
35      // User Sessions
36      app.post("/sessions",validateResource(createSessionSchema),createUserSessionHandler);
37      app.get("/sessions", requireUser, getUserSessionsHandler);
38      app.delete("/sessions", requireUser, deleteSessionHandler);
39
40      // API as an entity
41      app.get("/api", getUserAPIsHandler);
42      app.get("/api/public", getPublicAPIsHandler);
43      app.get("/api/:apiName", validateResource(getAPISchema), getAPIHandler);
44      app.post("/api", [requireUser, validateResource(createAPISchema)], createAPIHandler);
45      app.use("/api/:apiName", requireAPI, requireUser, userOwnsAPI); // For existing APIs
46      app.put("/api/:apiName", validateResource(updateAPISchema), updateAPIHandler);
47      app.delete("/api/:apiName", validateResource(deleteAPISchema), deleteAPIHandler);
48
49      // Serve APIS
50      app.use("/server/:apiName",requireAPI,isOperationAllowed,isOperationAuthorized, collectUsageStatistics);
51      app.post("/server/:apiName",validateServerResource(postOperationServerSchemas),postOperationAPIHandler)
52      app.use("/server/:apiName",parseAPIFilterQuery); // We need to parse the query in the URL for the following routes
53      app.get("/server/:apiName",getOperationAPIHandler)
54      app.put("/server/:apiName",validateServerResource(putOperationServerSchemas),putOperationAPIHandler)
55      app.delete("/server/:apiName",deleteOperationAPIHandler)
56
57      // Tokens
58      // Middleware in line 45 is applied
59      app.get("/api/:apiName/token", getTokenHandler);
60      app.post("/api/:apiName/token",[validateResource(createTokenSchema),tokenOperationsAuth], createTokenHandler);
61      app.put("/api/:apiName/token/:tokenUUID",[validateResource(updateTokenSchema),tokenOperationsAuth, requireToken], updateTokenHandler);
62      app.delete("/api/:apiName/token/:tokenUUID",[validateResource(deleteTokenSchema), requireToken], deleteTokenHandler);
63
64      // Statistics
65      app.get("/statistics/:apiName",[requireAPI, requireUser, userOwnsAPI],getStatisticsHandler)
66
67  }
```

Figura 3.4. Rutas del servidor desde el código

Estos son los significados de las distintas rutas:

- “/users” nos permite crear los usuarios de nuestra aplicación, es lo mismo que registrarse
- “/sessions” nos permite ver, crear y eliminar las sesiones. Las sesiones son utilizadas para el login del usuario en nuestra aplicación
- “/api” nos permite crear, modificar, editar y eliminar las APIs como entidad. Los datos que maneja esta ruta incluyen los esquemas de las APIs, su nombre, descripción, operaciones y si es pública

- “/api/public” sirve las APIs públicas. Los usuarios pueden marcar una opción que les permite mostrar públicamente sus APIs, de forma que otros las puedan ver.
- “/server” es la ruta donde de sirven las APIs ya creadas, admite las operaciones CRUD; aunque no siempre, como ya veremos estas pueden no permitirse, o hacer que se requiera un token para acceder a ellas.
- “/statistics” expone estadísticas de uso de las APIs, recogidos por día. Se uso un nombre general de cara a exponer más estadísticas en un futuro, como por ejemplo el número de veces que se ha recibido un código de respuesta concreto.

Además de los middlewares ya mostrados en la figura 3.4, tenemos otros 2 middleware por los que pasan todas las peticiones:

- Express.json(): parsea los objetos JSON que vengan en el cuerpo de la respuesta en formato de texto.
- logNewRequest: cada vez que llega una nueva petición saca los logs de esta, mostrando varios de sus parámetros, esto con el fin de ayudarnos a debugear la aplicación

3.4.2 Middleware de validación de datos

Una parte muy importante de esta aplicación es la correcta validación de los datos de entrada por parte del servidor. Aunque realmente en el cliente también se realiza alguna validación, pero estas son redundantes en el servidor, y realmente se incluyen en el cliente para que la carga sobre en el servidor sea más pequeña, y no se realicen peticiones innecesarias.

La validación es importante porque necesitamos estar seguros de que recibimos los datos de entrada en el formato requerido, y en caso de que el formato de la entrada sea incorrecto deberemos comunicárselo al cliente. De esta forma, el controlador puede asumir que está recibiendo los datos de entrada en un formato concreto, y no tener que preocuparse de hacer las validaciones por sí mismo. El validar los datos de entrada siempre es necesario, de otro modo podríamos tener bugs en nuestra aplicación, ya que no nos podemos fiar de lo que introduzca el usuario o un tercero con malas intenciones.

Para crear los esquemas de validación de datos he elegido zod.

Zod es una biblioteca de declaración y validación de esquemas. el término "esquema" se refiere en este caso a cualquier tipo de datos, desde una cadena simple hasta un objeto anidado complejo [9].

Para este middleware he creado una función en el archivo validateResource.ts. Es una función de orden superior o “high order function” en inglés, porque retorna

una función que está definida en su interior. La función se puede ver en la Figura 3.5.

```
function validateResource(schema: AnyZodObject){

    return function(req:Request, res:Response, next:NextFunction){
        log.info("[validate] init");
        try{
            let parseResult = schema.parse({
                body: req.body,
                query: req.query,
                params: req.params
            });

            req.body = parseResult.body;
            log.info("[validate] OK");
            next();
        }catch(e: any){
            return res.status(400).send(e.errors);
        }

    }
}
```

Figura 3.5. Función validateResource

Como podemos ver, recibe el objeto schema como parámetro y retorna una función que es el middleware. El middleware simplemente parsea la entrada y confirma que sigue el esquema. Si la entrada no tiene el formato correcto se devolverá una respuesta con status 400 y en su interior se encontrará un objeto JSON con los detalles de porque ha fallado la validación.

De esta forma, nosotros solo tendremos que pasarle la definición del esquema a la función y esta nos devolverá el middleware.

Se he requerido el uso de una función de alto orden porque el cuerpo de esta función se repite siempre en el middleware de validación, entonces, había 2 opciones: Crear esta función a la que le pasamos el esquema, o crear una función middleware distinta por cada esquema. Se ha elegido la primera opción a fin de reducir la duplicidad de código.

3.4.3 Registro

El objetivo del registro es que se pueda crear un usuario válido, y que este quede guardado en la base de datos.

Para eso, tenemos en el servidor la ruta “/users”, a la que podemos hacer un POST e incluir en el cuerpo de la petición el usuario que queremos crear.

La contraseña real que introduce el usuario no es guardada en la base de datos, en su lugar es guardado un hash que sirve como representación de esta, esto con el fin de dotar de más seguridad a la aplicación. Ya que en el improbable pero no imposible caso de que un atacante consiga acceder a nuestra base de datos podría hacerse con las credenciales de los usuarios. Esto podría suponer un grave problema, ya que es bastante habitual que los usuarios utilicen credenciales iguales o similares en múltiples aplicaciones. Para la generación del hash he usado bcrypt [20]. A la hora de hacer el login tendremos esto en cuenta, y pasaremos la contraseña que introduzca el usuario por bcryp también a fin de compararla con el hash que tenemos guardado en la base de datos.

Si todo va bien a la hora de realizar el registro el servidor nos devolverá una respuesta con status 200.

Si hay un problema de validación del cuerpo de la respuesta, el servidor nos devolverá una respuesta con status 400.

Si el email del usuario ya existe en el sistema se devolverá una respuesta con status 409.

En el cliente tenemos el formulario mostrado en la figura 3.6, una vez se hace click en el botón “Create Account”, se comprueba que los datos son correctos en el lado del cliente y solo entonces se manda una petición POST a la ruta “/users”.

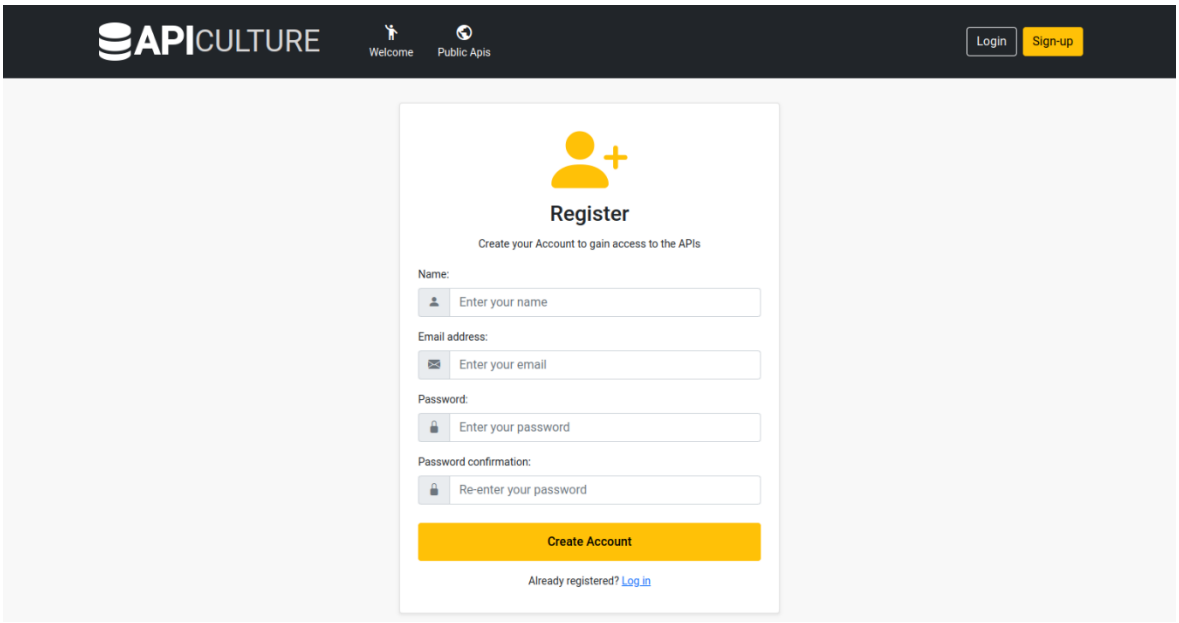

Figura 3.6. Cliente: pantalla de registro

3.4.4 Login

El login es la parte de la aplicación que nos permite dar acceso a un usuario a ciertas partes del sistema, a las que sin estar logeado no podría acceder. Para esto requerimos identificarlo y autenticarlo. Una vez sabemos que el usuario es quien dice ser podemos darle autorización a una parte del sistema. Esta parte del sistema a la que el usuario tendrá autorización son las rutas, recursos y datos a los que podrá acceder, por ejemplo, en este proyecto, un usuario puede modificar un API creada por el mismo, pero no puede modificar la API creada por otro usuario, y si el usuario no está logeado directamente no podrá crear un API.

Para que un usuario pueda acceder a la aplicación primero ha de autenticarse, creando una sesión de usuario. La sesión le proveerá de dos tokens que le servirán para no tener que volver a logearse de nuevo. Aunque los tokens tienen una fecha de expiración, por lo que en algún momento se le volverá a requerir al usuario que se logee.

Por lo tanto, al usuario solo le será necesario autenticarse una vez, y a partir de entonces, proveyendo el token en una cabecera podremos reconocer que el usuario se autenticó en el pasado, y que por lo tanto es el.

En este caso un token es una cadena de texto, que representa el hecho de que un usuario ya ha sido autenticado y ha creado una sesión.

Para la creación de los tokens se ha decidido utilizar JWT.

3.4.4.1 Json web token (JWT)

JWT, por sus siglas en inglés JSON Web Token, es un estándar que define una forma compacta y autocontenida de transmitir información entre pares usando objetos JSON. Esta información puede ser verificada y confiable debido a que está digitalmente firmada [13]. Para este proyecto hemos usado un par de clave privada/pública generadas por RSA para firmar y validar la información contenida en estos tokens. Para obtener este par de claves se ha hecho uso de una herramienta web que las genera [14].

Un token JWT tiene la siguiente estructura:

- cabecera: contiene el tipo de token y el algoritmo de firma
- payload o carga útil: aquí va el objeto JSON contenido en el token. Este es variable y somos nosotros los que decidimos esta información.
- firma: es el resultado de aplicar el algoritmo sobre la cabecera, y el payload.

Cuando el servidor recibe un token puede, con la clave pública, verificar el JWT, esta verificación, nos indica si el token es válido o no. Si es válido significa que un

payload y una cabecera idénticos a los recibidos fueron firmados con nuestra clave privada. De esta forma podemos confirmar que nosotros mismo firmamos este token en el pasado, con esta misma payload. Y, por lo tanto, si el usuario intenta modificar los valores del payload o de la firma del token, el servidor se dará cuenta de ello y dará una respuesta de error.

Luego, el servidor, puede usar el payload para tomar ciertas decisiones, en este caso, decidir qué usuario es al que le estamos dando el acceso y a que sesión hace referencia este JWT, ya que en la carga útil estamos incluyendo los siguiente campos:

- email del usuario
- id de la sesión
- cuando expira el token, representado en “unix time”

3.4.4.2 El proceso de login

El proceso de login es el siguiente:

1. El cliente hace un post a “/session”, incluyendo en el cuerpo de la respuesta un objeto JSON, con los campos “email” y “password”, los cuales han de representar a un usuario previamente registrado.
2. El servidor valida las credenciales del usuario, y si estas son correctas de responderá con status 200, y en el cuerpo de la respuesta ira un accessToken y un refreshToken
3. Ahora que el cliente tiene el accessToken, podrá hacer peticiones al servidor de ciertos recursos protegidos. Para ello, deberá incluir en la cabecera “Authorization” la palabra “Bearer” seguida de un espacio y el accessToken.

El accessToken es un token JWT que nos dará acceso al sistema por un tiempo configurable, en el caso de mi aplicación este tiempo es de 15 min. Es por eso que el usuario también recibe un refreshToken. El refreshToken nos permite solicitar nuevos accessToken a la aplicación, de forma que no se requiera al usuario logearse de nuevo. El tiempo de expiración de un refreshToken en este caso es de 1 año, aunque este es configurable. Es por esto que al cabo de 1 año el usuario requerirá logearse de nuevo.

El servidor puede saber cuando expira un token porque el tiempo de expiración va en la carga útil del mismo.

El hecho de usar refreshToken es para mejorar la seguridad, este nos permite que el tiempo de expiración de un accessToken sea pequeño, de forma que si un tercero interceptase este token, su tiempo de uso es pequeño (15 min.), la alternativa es tener accessToken de larga expiración (1 año por ejemplo), estos,

en caso de estos ser interceptados, le estarían dando al perpetrador acceso al sistema por un largo periodo de tiempo. Los refreshToken también pueden ser interceptados, pero al ser usados de forma mucho más infrecuente, esta probabilidad de ser interceptados es menor.

Como ya comentamos, en el payload del JWT incluimos el id de la sesión, esto es para reconocer la sesión a la que hace referencia el token. Esta sesión se guarda en la base de datos y tiene un campo que nos permite ver si una sesión ha sido expirada o si aún es válida. De forma que el usuario también pueda declarar una sesión como inválida, esto haría que el accesToken y el refreshToken queden inutilizables, incluso cuando el tiempo de expiración aún no haya pasado. La forma en la que un usuario puede declarar una sesión como inválida es haciendo una petición delete a “/session” incluyendo el accessToken; esto está pensado para ser llamado cuando el usuario presiona el botón de deslogearse o “Log Out” en una página web.

3.4.4.3 Login y autorización en el cliente

El cliente tiene un formulario para llevar a cabo el login. En él se pide al usuario un email y una contraseña. Este formulario puede verse en la figura 3.7.

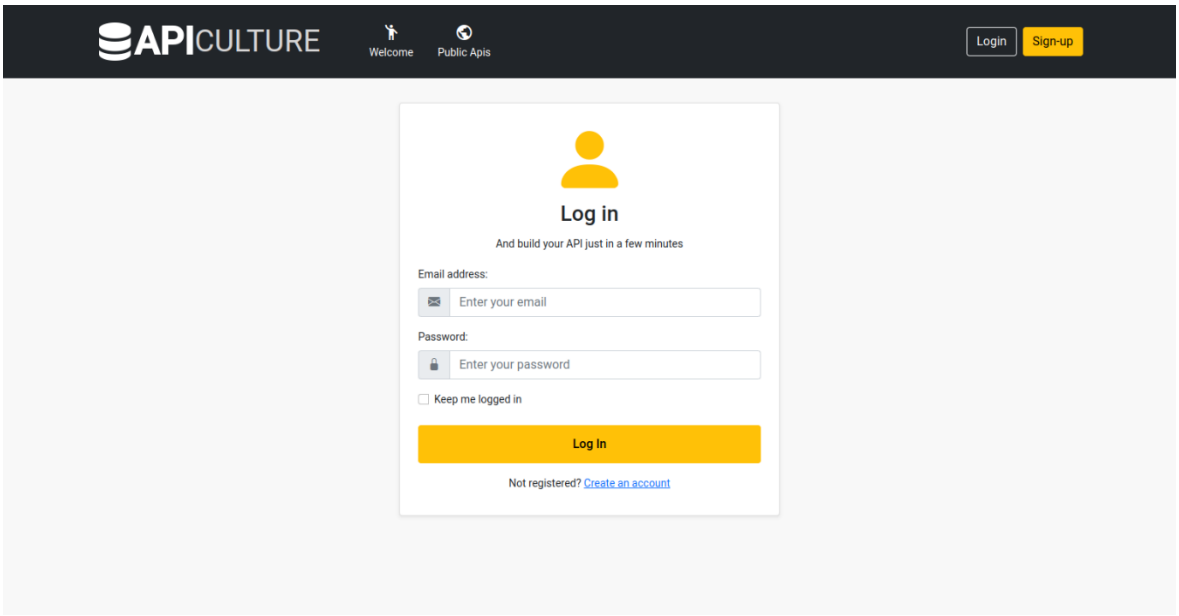

Figura 3.7. Cliente: pantalla de login

El login básicamente hace una llamada post a /session, como ya se indicó anteriormente.

Si las credenciales de login son correctas, entonces recibiremos los 2 tokens, y procederemos a guardarlos en LocalStorage. LocalStorage es una API de HTML5

que nos permite guardar datos en el lado del cliente, funciona como un almacén de pares clave-valor, y los datos persistidos en el no expiran a no ser que se indique.

Una vez tenemos en el LocalStorage del navegador guardados estos tokens, en casi todas las subsecuentes consultas al backend, incluiremos este token en las peticiones HTTP. Esto lo hacemos mediante el uso de HTTP_INTERCEPTORS, que son una clase de proveedores en Angular que nos permiten interceptar todas las consultas que el forntend le hace al backend. Lo que hacemos en este caso es inyectar en la cabecera “Authorization” el token de acceso.

Otra parte que manejamos en los interceptores es cuando el tiempo de vida de un token expira, y necesitamos pedir un nuevo token de acceso al servidor, por medio de un refreshToken. Esto lo hacemos a través de interceptar las respuestas del servidor en las que se devuelve un código de status “unauthorized”, en este momento aplicamos la lógica de petición de un nuevo accessToken, pero es posible también que el refreshToken haya expirado, por lo que si recibimos una segunda respuesta como “unauthorized”, esta vez borraremos los tokens de la sesión en el navegador y a continuación redimiremos al usuario a la página de login. Este suceso es muy poco habitual, 1 vez al año.

Además de esto se usan guardas para proteger las rutas no permitidas cuando el usuario no está logeado.

3.4.4.4 Login y autorización en el servidor

Como ya comenté anteriormente tenemos el post a la ruta /session para obtener los tokens, y el delete a /session para eliminar una sesión.

Además, tenemos 2 middleware para llevar a cabo la autorización:

El primero, llamado “deserializeUser”, cuyo objetivo es obtener el usuario a partir del token y guardarlo en una variable, siempre que este token sea válido. Además, si detecta que en las cabeceras se ha incluido un refreshToken, y que el accessToken, en caso de ser provisto, esta expirado, entonces nos proporcionará un nuevo accessToken en una de las cabeceras de la respuesta.

El segundo middleware, “requireUser” es usado para proteger ciertas rutas, a las que solo se puede acceder si se ha provisto un accessToken valido. Básicamente comprobamos que en la variable rellenada por “deserializeUser” haya un usuario.

No se incluyen más detalles sobre esta parte porque está hecho de forma similar a lo expuesto en la referencia el post comentado anteriormente [8].

3.4.5 Creación de un API

Para poder crear un API se ha de hacer una petición post al servidor, incluyendo un token válido en ella, ya que solo los usuarios logeados pueden crear APIs, en el cuerpo de la petición debe ir la información del API, como son su nombre, descripción, operaciones, la opción de listado público y su esquema de datos.

3.4.5.1 Creación de un API en el servidor

Para este fin tenemos la ruta “/api” en el servidor, donde tenemos que hacer una petición post. Esta ruta está protegida solo para usuarios logeados, por lo que se requiere incluir el accessToken en la cabecera “Authorization”.

En el cuerpo de esta petición irá la configuración de la API que queremos crear. Este cuerpo es validado contra un esquema de zod, para asegurarnos de que el usuario no incluye datos erróneos.

Además del esquema como tal, zod nos permite llamar al método superRefine sobre el esquema para realizar validaciones que no están soportadas por zod, y así poder añadir nuestra propia lógica de validación. En la figura 3.8 se puede ver un resumen de este esquema zod, el cuerpo de las funciones superRefine no se muestra porque es bastante largo.

```
const innerFieldSchema: z.ZodType<innerField> = z.lazy(() =>
  object(
  {
    name:optional(string()),
    type:z.enum([types[0],...types.slice(1)]),
    nullable:optional(boolean()),
    innerData: optional(array(innerFieldSchema).nonempty().or(innerFieldSchema))
  }).strict().superRefine(innerFieldSuperRefine)
);

const fieldSchema = object(
  {
    name:string(),
    type:z.enum([types[0],...types.slice(1)]),
    nullable:optional(boolean()),
    unique:optional(boolean()),
    innerData: optional(array(innerFieldSchema).nonempty().or(innerFieldSchema))
  }
).superRefine((val, ctx)=>{···
})

const operationsSchema=array(object({
  verb: string(z.enum([operationVerbValues[0],...operationVerbValues.slice(1)])),
  requires_auth: optional(boolean()),
}).strict()).refine((val)=>val.length === new Set(val.map(data => data.verb)).size, {message:"Operations can't be repeated"});

const apiSchema = {
  body: object({
    name: string({ required_error: "Api name is required" }),
    description: string(),
    operations: operationsSchema,
    public_listing: boolean(),
    data: array(fieldSchema).nonempty()
  }).strict().superRefine((val, ctx) => {···
  })
};
```

Figura 3.8. Esquema de Zod para la creación de APIs

Para los Verbos de una operación y los posibles tipos de datos que se aceptan se han creado dos enums. También se a creado una interfaz “innerField”, esto es porque según la documentación de zod [9], hay una deficiencia de typescript respecto a las definiciones recursivas y por lo tanto el propio zod no es capaz de crear esta definición por sí mismo.

```
export enum OperationVerb{
  get = "GET",
  post = "POST",
  put = "PUT",
  delete = "DELETE",
}

const operationVerbValues = Object.values<string>(OperationVerb);


export enum Type {
  String = "String",
  Array = "Array",
  Integer = "Integer",
  Object = "Object",
  Boolean = "Boolean",
  Double = "Double"
}

const types = Object.values<string>(Type);


export interface innerField {
  name?:string;
  type:string;
  nullable?:boolean;
  innerData?: innerField[] | innerField;
}
```

Figura 3.9. Algunas estructuras en usadas en la creación APIs

El cuerpo de la petición ha de contener un objeto JSON, con los siguientes campos:

name, es nombre de la API, ha de ser único, no pueden haber 2 APIs con el mismo nombre.

description, es la descripción de la API, aquí se puede incluir una explicación de la API, o algún tipo de documentación si se considera necesario.

public_listing, este campo es un booleano, que sirve para indicar si los datos relativos a la definición de nuestra API pueden ser vistos por otros usuarios, incluso si no están logeados, es decir, si este campo es “verdadero”, se podrá hacer un get a /api/public y se retornará esta API como uno de los resultados, si el campo está a “falso”, entonces no se incluirá esta API entre los resultados. El fin

de este campo es que otros usuarios puedan conocer nuestra API, y así saber como usarla.

operations es la lista de operaciones que soporta la API, cada elemento de la lista es un objeto formado de dos campos, el primero indica el verbo de la operación, que a su vez indica el significado semántico de la operación; el otro campo es “requires_auth”, que indica si para acceder a esta operación del API va a ser necesario proveer un token de autenticación, en caso de estar a false o no definirse todo el mundo podrá llamar a esta operación del API.

```
"operations": [
    {
        "verb":"GET",
        "requires_auth":false
    },
    {
        "verb":"POST",
        "requires_auth":true
    },
    {
        "verb":"PUT",
        "requires_auth":true
    }
],
```

Figura 3.10. Ejemplo concreto para el campo “operations” de una API

data es la parte que contiene la definición del esquema de datos de la API. Este esquema que ponemos en “data” es el que tienen que seguir los datos que se almacenan dentro de la propia API, y además, será utilizado para las validaciones en las operaciones contra el API.

Como podemos ver en la Figura 3.8, data es un array de campos no vacíos, que siguen el esquema “fieldSchema”. “fieldSchema” se usa para el primer nivel de anidación, que tiene algunas propiedades diferentes del resto de niveles de anidación; el resto de los niveles de anidación siguen el esquema “innerFieldSchema”.

“fieldSchema” e “innerFieldSchema” tienen 3 propiedades compartidas, pero he preferido repetir código en ambas definiciones a tener que refactorizar, porque considero que así se leen mejor los esquemas.

Cada campo a su vez puede tener las siguientes propiedades:

- name: el nombre del campo.
- type: es su tipo, y puede ser uno de los siguientes, que está definido en un enumerado: String, Array, Integer, Object, Boolean, Double.
- nullable: si el valor del campo puede ser “null”.

- unique: si el valor de este campo es único a lo largo de todas las instancias almacenadas en un API.
- innerData: este campo es opcional, y solo se debe de incluir en caso de que el valor de type sea Array o Object. En caso de que type sea object innerData es contiene un objeto que sigue el esquema “innerFieldSchema”, pero el cual no contendrá el campo “name”, ya que un array es indexado por números enteros y no por nombres. Si type es Object, entonces innerData es un array de elementos que siguen el esquema “innerFieldSchema”.

“innerFieldSchema” está definido de forma recursiva, es de esta forma que podemos validar esquemas con anidación infinita, ya que un objeto puede contener a su vez un objeto o un array, que a su vez pueden contener objetos o arrays… El nivel de anidación es definido por el usuario que crea la API a fin de que el esquema satisfaga sus necesidades.

En la Figura 3.11 tenemos un ejemplo concreto y válido para el campo data, en este caso tenemos que los objetos de esta API tendrían un nombre, una dirección (address), que a su vez es un objeto, y unas preferencias de compra (shoppingPreferences), que es un array de strings.

```
"data": [
  {
    "name": "name",
    "type": "String",
    "nullable": false,
    "unique":false
  },
  {
    "name": "address",
    "type": "Object",
    "innerData": [
      {
        "name": "street",
        "type": "String"
      },
      {
        "name": "number",
        "type": "Integer"
      }
    ]
  },
  {
    "name": "shoppingPreferences",
    "type": "Array",
    "innerData":
      {
        "type": "String"
      }
  }
]
```

Figura 3.11. Ejemplo concreto para el campo “data” de una API

Una vez el cuerpo de la petición ha sido validado, el servidor procede a guardar los datos de la API en la base de datos, añadiendo el usuario que creó la API.

Una vez el API está guardado en la base de datos, hay que desplegar la API para que pueda ser consumida por los clientes. Para esto, el controlador llama a los servicios, para que realicen las siguientes instrucciones:

- Creación de un modelo en la base de datos para la API, siguiendo el esquema que se definió en el campo “data”. Luego se insertará este modelo en un diccionario de modelos, donde existe un modelo para cada API, siendo la clave del diccionario el nombre de la API, y su valor asociado el modelo de la API.
- Creación de un esquema de zod para la validación de las operaciones post que añadan nuevos datos al API, luego este esquema se añade a un diccionario de esquemas de zod, donde hay uno para cada API, siendo el nombre de la API la clave y siendo su valor el esquema de zod.
- Creación de un esquema de zod para la validación de las operaciones put que actualicen datos sobre el API. Al igual que con los esquemas para las peticiones post, este esquema será incluido en un diccionario.

3.4.5.2 Creación de un API en el cliente

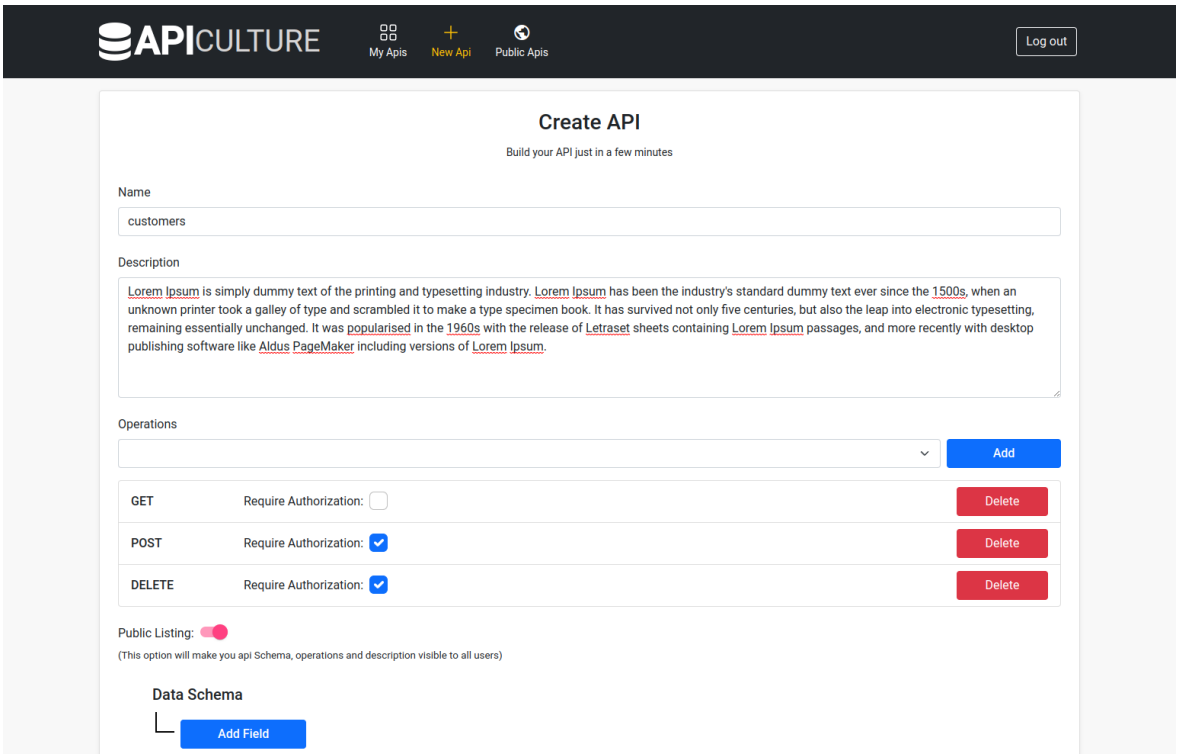

Figura 3.12. Cliente: Formulario de creación de APIs

Para ello tenemos un formulario llamado “Create API”.

Podemos ver un formulario rellenado para el ejemplo de clientes en la Figura 3.12, a falta del campo “data schema”, este se puede ver en su totalidad en las Figuras 3.13 y 3.14.

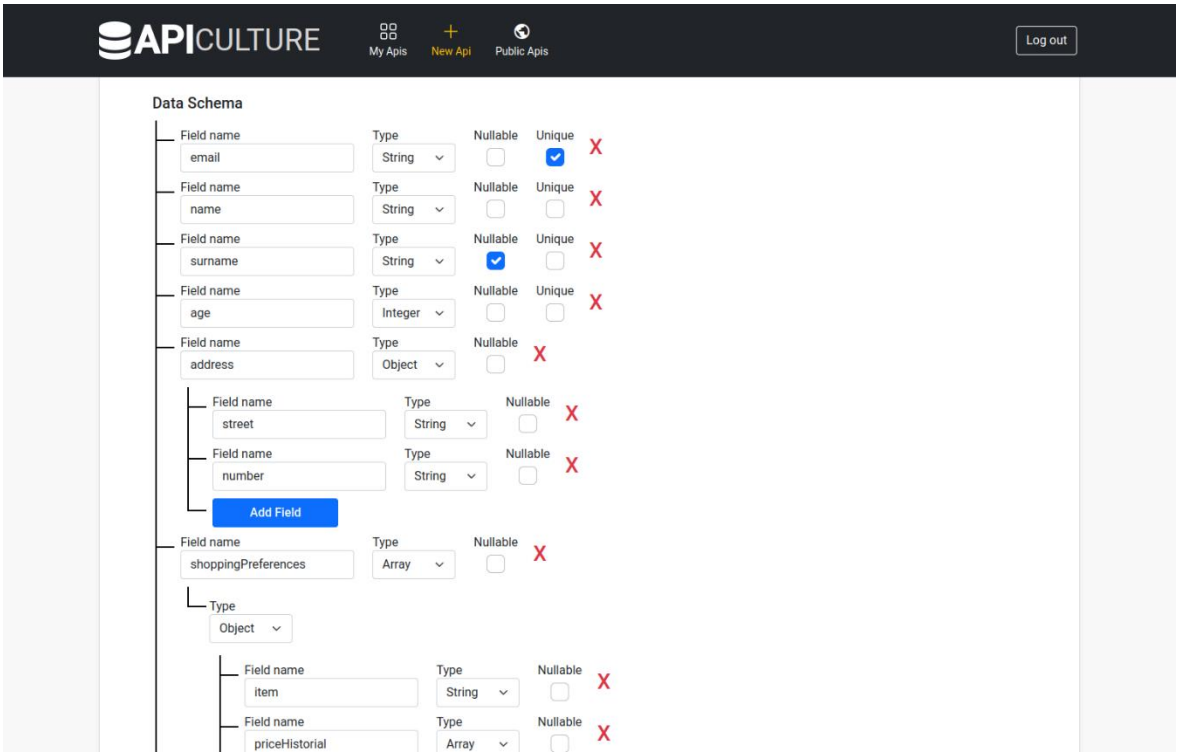

Figura 3.13. Cliente: campo “Data Schema” del formulario de creación de APIs

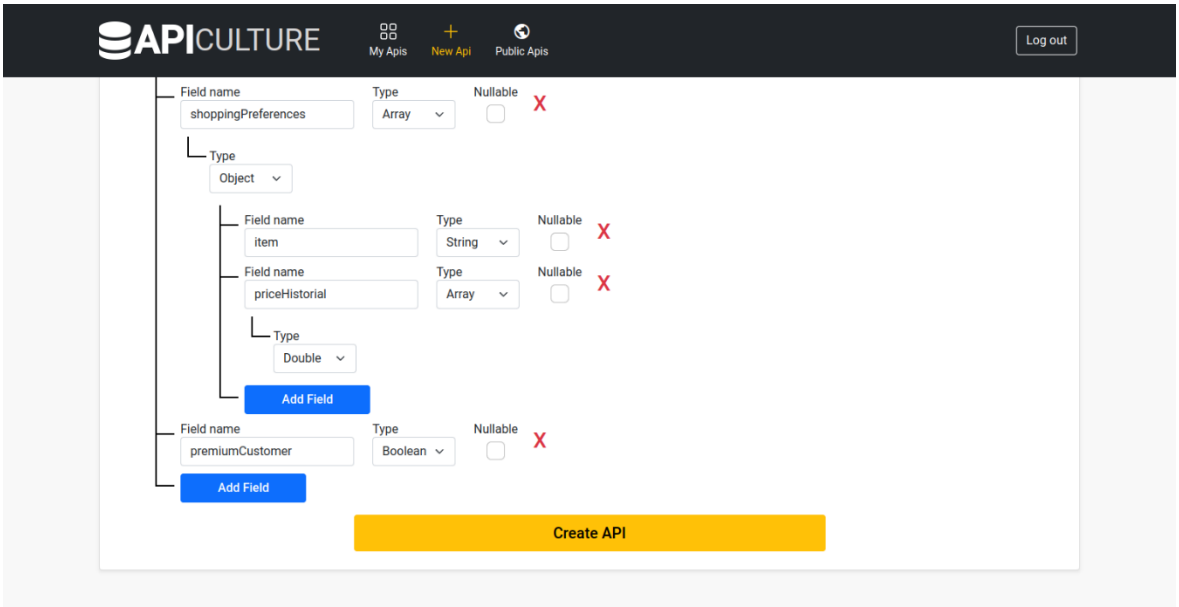

Figura 3.14. Cliente: Continuación a la Figura 3.13

En la sección de operaciones tenemos un combobox en el cuál podemos seleccionar operaciones y añadirlas con el botón “Add”, estas operaciones, una vez añadidas son mostradas en una lista, donde se puede indicar si estas requieren autenticación; también pueden ser eliminadas, indicando que esta API no las soporta.

La sección Data es para la configuración del esquema de los datos que seguirá la API, presionando en el botón de “Add Field” mostrado en la parte inferior de la Figura 3.12 se añadirá un nuevo campo al formulario, haciendo click en la “equis” roja de la derecha podemos volver a eliminar este campo.

Cuando marcamos el tipo de un campo como “Object”, se nos crea un nuevo nivel de anidación, donde podemos incluir los campos que tendrá este nuevo objeto, creándose así un esquema en forma de árbol.

Otro de los tipos que puede tener un campo es “Array”, cuando esto ocurre, en el desplegable, solo podremos elegir un tipo de datos que contendrá el array.

También puede verse que las posibilidades para la creación de esquemas que nos ofrece el cliente son consecuentes con las posibilidades que nos ofrece el servidor, por ejemplo, solo se nos muestra el checkbox de “unique” en el primer nivel de anidación, y solamente si el tipo del campo es string o integer.

Una vez el usuario haga click en el botón “CREATE API”, se hará primero una validación en el lado del cliente, y después se hará la petición post al servidor. En función de la respuesta recibida del servidor, el cliente mostrará el mensaje de error específico para informar al usuario, o en caso de que todo haya ido bien, el cliente redirigirá al usuario a la lista de sus APIs ya creadas.

3.4.5 Modificación, eliminación y obtención de APIS

El usuario puede consultar todas sus APIs haciendo una petición get a /api, en el cliente tenemos la pantalla equivalente donde al usuario se le muestran todas sus APIs. En la figura 3.15 podemos ver la ruta del navegador “/dashboard/my_apis”, donde al usuario se le muestra una lista con todas sus APIs.

Vemos que a la derecha hay dos botones, el que esta más hacia la derecha, Delete, sirve para eliminar un API, si hacemos click se muestra un diálogo que nos pregunta si estamos seguros, si confirmamos esto eliminará el API, haciendo una petición delete a “/api/apiName”, donde apiName es el nombre de la API.

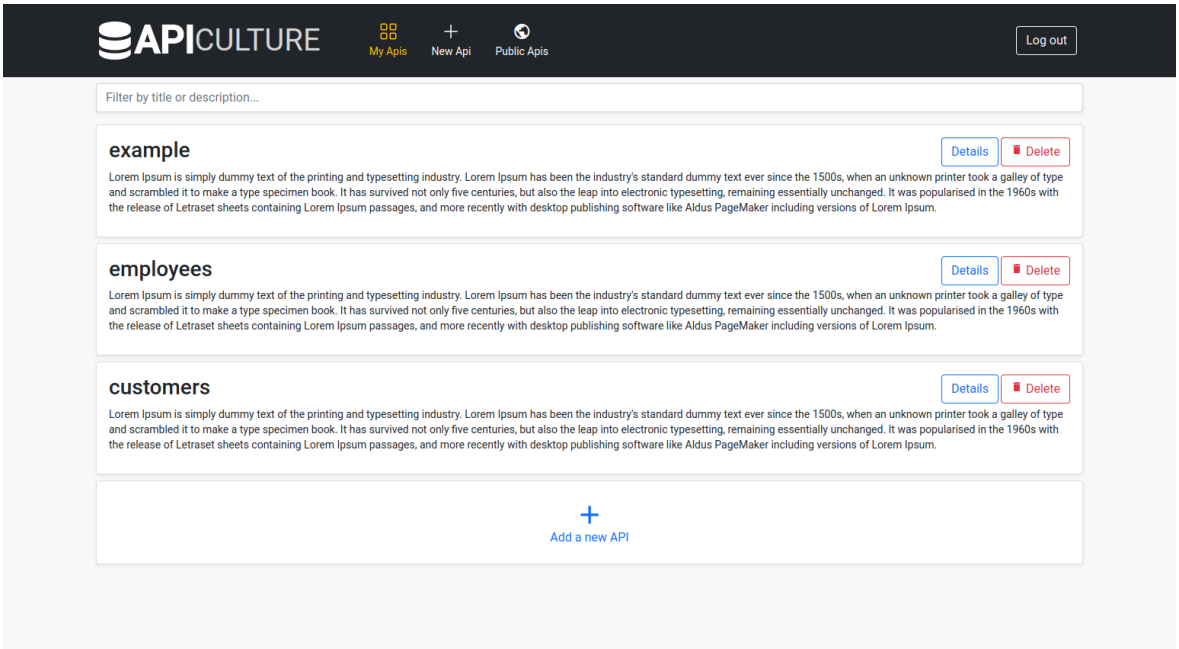

Figura 3.15. Cliente: pantalla “My Apis” de algún usuario

La eliminación del api elimina a su vez múltiples estructuras internas:

- Elimina la instancia de la base de datos, con esto nos referimos a la instancia que contiene nombre, descripción, esquema…
- Se elimina la colección de la base de datos.
- Se elimina el modelo de la API almacenado en el diccionario de modelos, que se encarga de la comunicación con la base de datos.
- Además, se eliminan los esquemas de validación de la API de los diccionarios en los que se guardan

El otro botón que está más a la izquierda, “Details”, nos muestra un diálogo con los detalles de la API.

Además, podemos hacer click sobre el panel de una API y se nos mostrará una nueva vista, con un panel de navegación a la izquierda para navegar por las diferentes opciones de nuestra API.

Por defecto, tras hacer click iremos al panel de “Details”, como vemos en la Figura 3.16

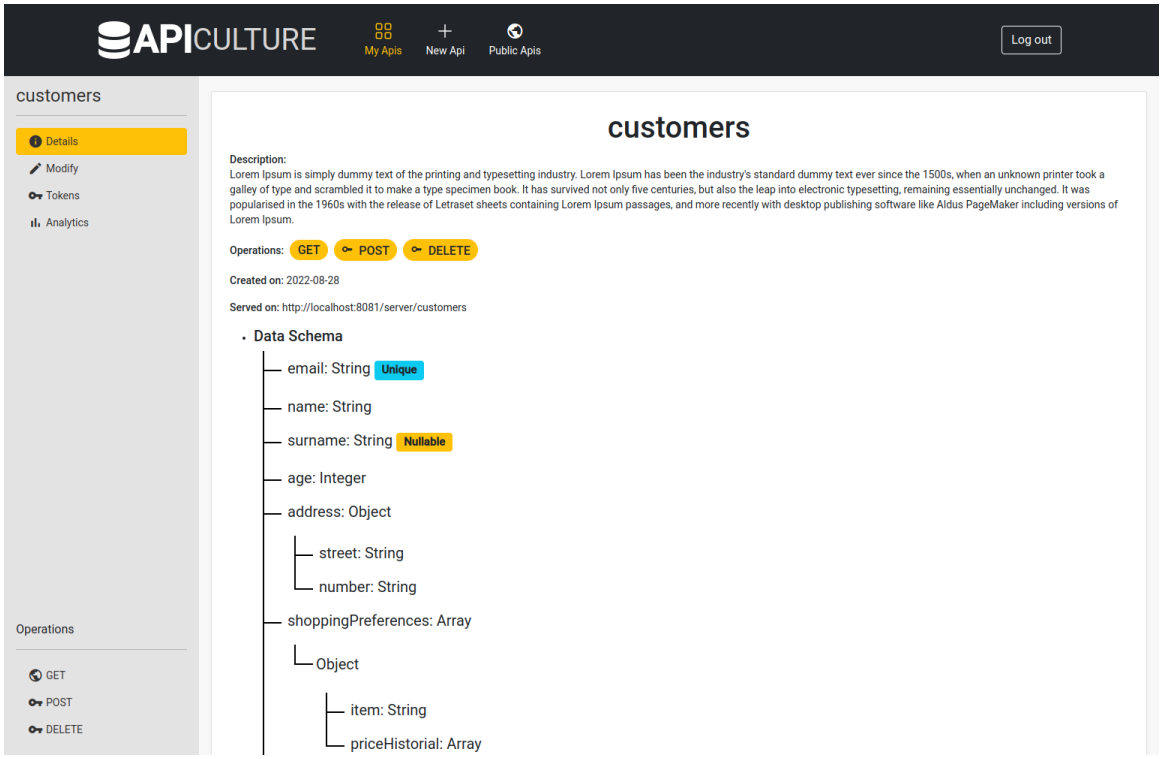

Figura 3.16. Cliente: Detalles de un API

En el panel modify, como puede verse en la Figura 3.17, podemos modificar nuestra API. Podemos modificar todo menos el esquema de datos.

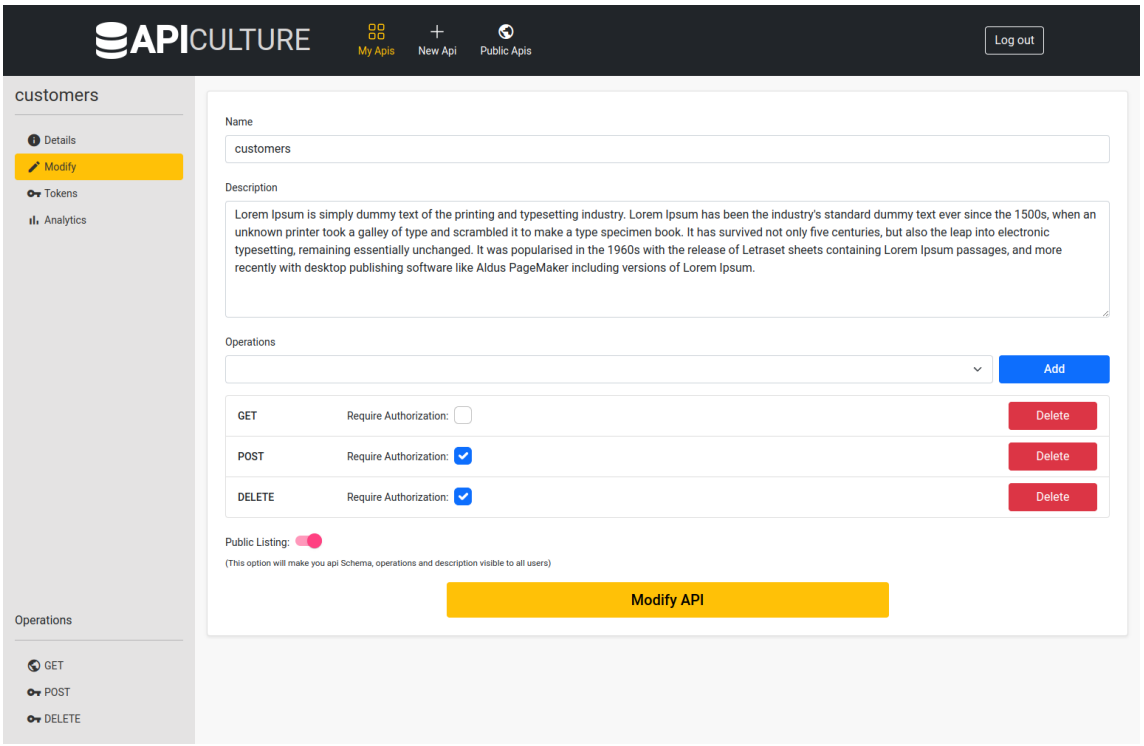

Figura 3.17. Cliente: Modificación de un API

La posibilidad de modificar el esquema de datos se deja como una posible ampliación del proyecto, esta fue planteada en el inicio del proyecto, pero finalmente fue descartada debido a que requería la toma de distintas decisiones, y sobre todo, por el lado del cliente tiene cierta dificultad de implementación.

Al editar un esquema podemos estar eliminando campos, esto haría que en todos los objetos almacenados en esa API tengamos que eliminar esos campos, perdiendo esta información; si por el contrario vamos a añadir un campo nuevo tendremos que indicar como vamos a rellenar esos campos en las instancias de nuestra API que ya haya creadas en la base de datos.

O otras decisiones, como que debería pasar si cambiamos el valor de un campo ya existente a que sea unique, es decir, que debería pasar con los valores de esos campos que no sean únicos; algo similar pasa en el caso de pasar de un campo que era nullable a que ahora no lo sea.

Además, esto también causaría problemas a los clientes que estén consumiendo estas APIs, ya que, si el esquema cambia, los clientes deben de cambiar su código para adaptarse a las nuevas necesidades, y esto requiere cierto coste y tiempo.

3.4.6 Sirviendo las APIS

La forma en la que se va a servir un API depende de los parámetros con los que la hemos creado. Para empezar, todas las APIs se sirven a partir de la ruta “/server”, anidándole el nombre de nuestra API detrás, por ejemplo, si nuestra API se llama clientes, esta será servida en la ruta “/server/clientes”, esta es una de las razones por la que el nombre de nuestra API debe de ser único.

Sobre esta ruta se permiten realizar una serie de operaciones, estas operaciones fueron definidas a la hora de crear la API, en el campo operations de la misma. Como ya vimos en el punto 3.4.5.1, el campo operations está formado por un array, siendo cada elemento del array un objeto con las propiedades “verb” y “requires_auth”. “verb” es el verbo de la operación, y “requires_auth” es el campo que indica si para realizar esta operación se debe de pedir autenticación por un token previamente creado.

Cada verbo solo se puede incluir en el listado de operaciones una vez, porque cada uno define una operación distinta sobre la API.

Cuando vayamos a hacer una petición a la API, el verbo solo puede ser una de las opciones que declaramos en la lista de operaciones. El verbo, puesto que está cargado de un significado semántico, define la operación en sí misma, es decir, lo que la operación hace. A continuación se detalla el significado de cada una de las operaciones posibles:

- GET, sirve para obtener datos de la API. En el cuerpo de la respuesta vendrá un array con los objetos de la API.

Esta operación tiene un posible parámetro que es la query, que sirve para filtrar datos dentro de la API, está query debe de incluirse después de la ruta, añadiendo un símbolo “?” e insertando nuestra query a continuación. La forma de crear estas consultas o queries se definen más adelante, después de los verbos.

- POST, sirve para añadir nuevos datos a la API. Tenemos que incluir en el cuerpo de la petición los datos que queremos añadir a nuestra API
- PUT, sirve para actualizar datos en nuestra API, tiene 2 parámetros, el primero, contesta a la pregunta ¿Qué datos queremos actualizar?, y este viene definido por una query, que al igual que la operación “GET”, esta se incluye al final de la URL. El otro parámetro responde a la pregunta ¿Con que queremos actualizarlo?, y esta parte ha de ir dentro del cuerpo de la petición.
- DELETE, sirve para eliminar una serie de datos dentro de nuestra API. Hay que indicar que datos son los que queremos borrar, esto se hace a través de una query también, que se incluye al final de la URL.

Como hemos visto, las peticiones sobre un API las podemos parametrizar de 2 formas, la primera a través del cuerpo de la petición, y el segundo a través de una query que se incluye al final de la URL de la petición.

La forma en la que estamos incluyendo las queries se denomina “query params”, la cual está definida para incluir queries dentro de las URL. Normalmente los “query params” solo nos permiten el uso del operador de comparación “=”, pero esto ofrece muy poca flexibilidad, por lo que he hecho uso de la librería api-query-params [16], la cual nos ofrece muchos más operadores, en la tabla 3.1 se puede ver un resumen de estos operadores.

Un ejemplo de uso podría ser: petición GET a http://localhost/server/customer?name=pedro&age>12&address.numero=15

La cual seleccionaría a los clientes que se llaman pedro, tienen más de 12 años y la dirección de su casa tiene el número 15.

Operadores de Filtro	URI	Ejemplo
Igual	key=val	type=public
Mayor	key>val	count>5
Mayor o igual	key>=val	rating>=9.5
Menor	key<val	createdAt<2016-01-01
Menor o igual	key<=val	score<=-5
Distinto	key!=val	status!=success
Si está dentro de un conjunto de valores	key=val1,val2	country=GB,US
Si sigue la expresión regular value con las opciones opts	key=/value/<opts>	email=/@gmail\.com$/i

Tabla 3.1. Algunos de los filtros disponibles en api-query-params [16]

3.4.6.1 Autorización

Una de las opciones a la hora de definir las operaciones es “requires_auth”, esta opción lo que implica es que a la hora de realizar una petición con el verbo de esa operación sobre la API, se debe de incluir un token válido en la cabecera “Authorization”.

Un breve inciso, a partir de ahora, también nos referiremos a las operaciones como “publicas” cuando estas no requieran de autenticación, y diremos que son privadas cuando estas requieran autenticación.

La forma correcta de incluir el token es poniendo en el valor de la cabecera “Authorization” la palabra “Bearer” seguida de un espacio y el valor token.

Si no hemos marcado la opción “requires_auth”, no se requiere ningún tipo de autorización, esto significa que cualquiera puede hacer esa operación sobre la API.

El tener tokens es una ampliación de cara a usuarios que no quieran tener sus APIs totalmente abiertas al público. Al poder marcar solo ciertas operaciones como privadas, esto añade la posibilidad de solo exponer ciertas operaciones de cara al público. Por ejemplo, una API podría tener solo la operación GET sin requerir autorización de cara a que los datos de su API puedan ser consumidos por cualquiera.

Para la creación de los tokens tenemos la ruta “/api/apiName/token”, donde “apiName” debe de ser sustituido por el nombre concreto de la API. En la ruta de los tokens podemos realizar las 4 operaciones de un CRUD sobre los tokens.

Los tokens han de cumplir las siguientes propiedades:

- ser únicos
- ser seguros, que los valores aleatorios que contienen estos tokens no puedan ser adivinados, es decir, que sean criptográficamente seguros.

Además, necesitamos poder hacerlos inválidos si es necesario, y necesitamos saber qué operaciones soporta el token y sobre qué API.

En un principio se pensó en utilizar JWT para la creación de los tokens, pero visto que necesitamos poder hacer que un token se vuelva inválido, necesitaríamos guardar el JWT en la base de datos, por lo que algunas de sus ventajas pasarían a ser superfluas; es por eso es que finalmente decidí hacer uso de UUID para los tokens.

Además del UUID del token, en la base de datos se guardan las operaciones soportadas por el token, sobre qué API y si este token es aún válido.

Un UUID, por sus siglas Universal Unique Identifier, o Identificador Universalmente Único, es una cadena de 128 bits de longitud capaz de garantizar su unicidad en el espacio y en el tiempo [17].

Se ha optado por el paquete uuid[18] para la generación de los UUID, porque sirve para la generación de UUID criptográficamente seguros, a pesar de que en el estándar que define los UUID esto no está incluido.

Por otro lado, en el cliente, una vez estamos en el menú de una API, podemos seleccionar el menú “tokens” en el panel de navegación a la izquierda, y esto nos llevará a la vista en la Figura 3.18.

Aquí tenemos, 2 paneles, en el panel superior podemos crear un nuevo token para esta API, pero antes tenemos que seleccionar las operaciones que soportará el nuevo token, para eso tenemos el combobox de múltiple selección [21] encima del botón de crear el token. Como se ve en la Figura 3.18, en el combobox se han seleccionado las operaciones POST y DELETE.

En el panel inferior pueden verse todos los tokens creados expuestos en una tabla. Para cada token se muestra su clave, en formato UUID y las operaciones a las que nos da acceso ese token. En la parte de operaciones de la tabla podemos eliminar un token y hacerlo válido o inválido. El botón para hacer válido el token es “make valid”, y solo se nos mostrará en caso de que el token no sea válido; sin embargo, si el token es actualmente válido, se nos muestra el botón “invalidate” para hacer que el token sea inválido.

El hecho de poder invalidar un token nos permite quitarle el acceso por un tiempo a un usuario, pudiendo volver a devolvérselo en un futuro, haciendo el token válido para ello.

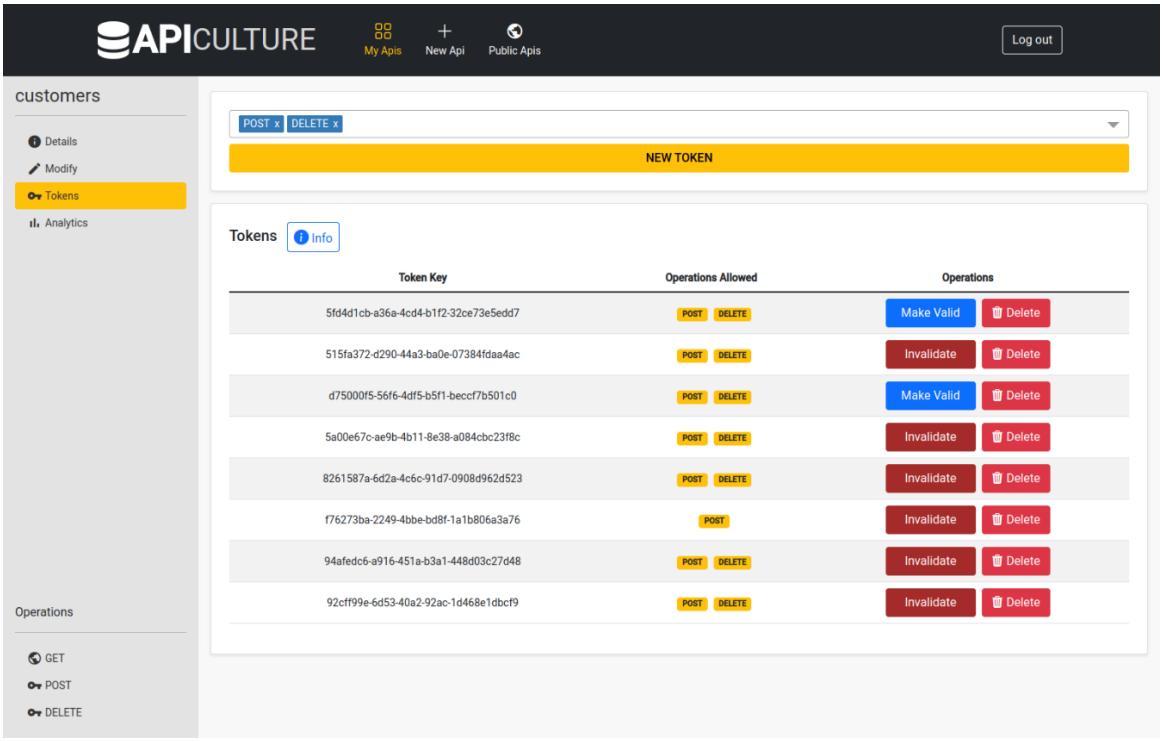

Figura 3.18. Cliente: creación y administración de tokens

Por último, el botón “info” le muestra al usuario un breve diálogo donde se le muestra que es un token y como puede utilizarlo, la ayuda del diálogo se muestra en la Figura 3.19.

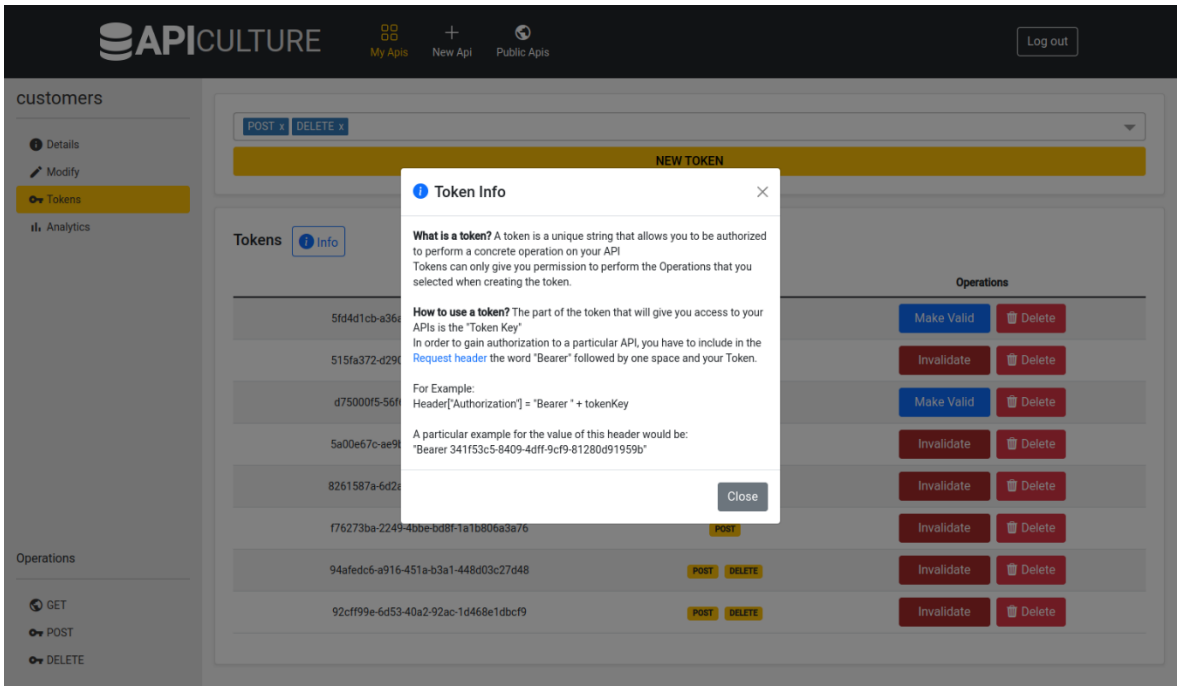

Figura 3.19. Cliente: información sobre tokens

3.4.6.2 Ejecutar peticiones sobre un api desde el cliente

Desde el cliente también se ha añadido la posibilidad de hacer llamadas a la API, basándose en un estilo básico, similar en su esencia a Postman [22], pero ciertamente mucho más sencillo y con menos posibilidades. Lo que se buscaba con esto es poder hacer llamadas rápidas a nuestra API sin tener que abrir otra aplicación. La otra razón es que esta forma de realizar llamadas es intuitiva y puede servir como una especie de documentación al usuario sobre como llamar a sus APIs, de forma que luego pueda hacer el mismo las llamadas desde donde requiera. Además, esta parte requiere poco esfuerzo de implementación.

Para acceder a estas llamadas simplemente hay que seleccionar el verbo de la operación que queremos realizar en la parte inferior de la barra de navegación izquierda.

La Figura 3.20 representa una llamada a la operación GET, esta operación es pública, esto podemos saberlo por el icono que tiene el GET en la barra de navegación, cuando se trata de una llamada que requiere autenticación se muestra una llave, por el contrario, si no requiere de autenticación se muestra el planeta tierra, lo cual indica que puede acceder todo el mundo. Simplemente, vemos que indica la ruta donde se realizará la llamada, además, incluye un campo donde podemos introducir la query para filtrar datos; la query, a su vez, incluye un icono que al posar el cursor sobre él se muestra un popover de bootstrap con información sobre como usar ese campo.

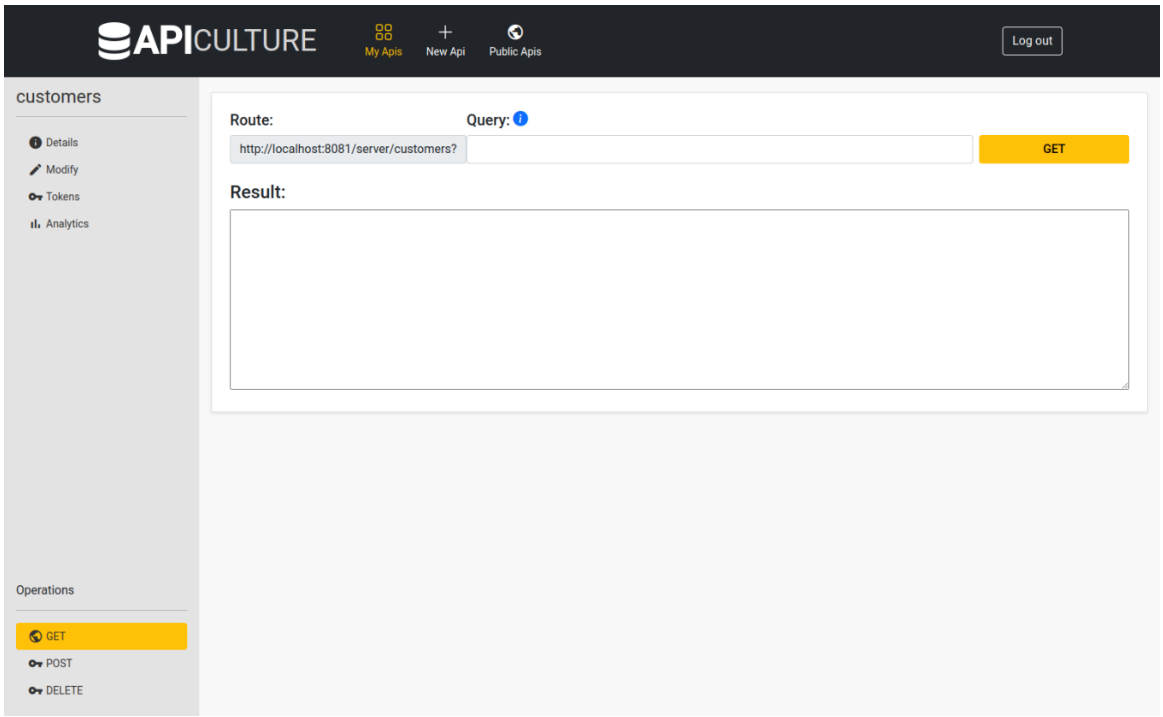

Figura 3.20. Cliente: realizar operación GET sobre una API

En la Figura 3.21 vemos la vista para una operación PUT que requiere autenticación por token. Al ser un PUT, tenemos un body, que es el cuerpo de la petición, la ruta y la query se mantienen como en la operación GET, ya que en una petición PUT tenemos que seleccionar los datos que queremos actualizar mediante query. Adicionalmente, puesto que la operación es privada, tenemos un nuevo campo que es el token, aquí se ha de incluir la clave del token para que el backend nos pueda dar acceso; en el icono de información que esta tras la palabra token se muestra una pequeña ayuda al usuario sobre cómo usar el token y como se debe incluir cuando haga una petición al servidor por su cuenta.

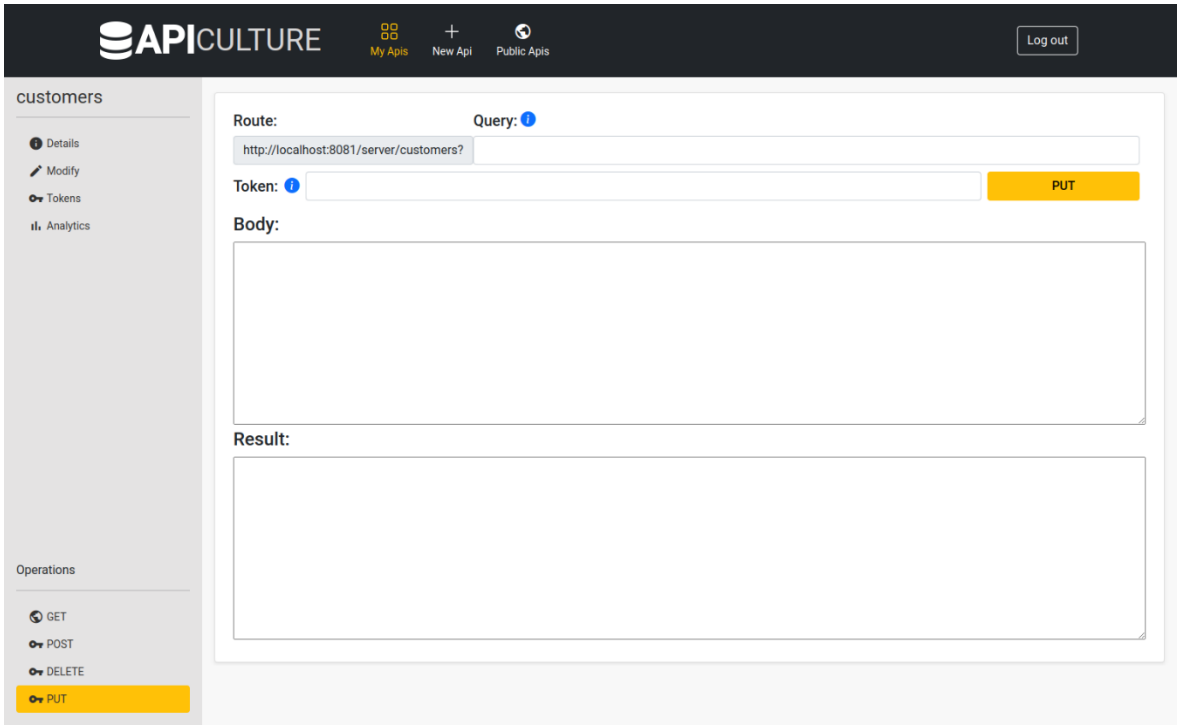

Figura 3.21. Cliente: realizar operación PUT sobre una API

3.4.7 Estadísticas de uso

Otra de las especificaciones de la API era la recolección del número de peticiones que llegaban por día, separadas por el tipo de operación, y con la posibilidad de agruparlas según el token que se ha usado (en caso de que la operación sea privada).

Para recoger estas estadísticas tenemos el middleware “collectUsageStatistics” en el servidor.

En el cliente web tenemos unas gráficas, para cada API, mostradas en la Figura 3.22, que muestran el número de peticiones que ha habido cada día. También se

puede filtrar por token, en la parte de arriba, vemos un combobox para esto, que nos ofrece los siguientes filtros:

- Filtrar por las peticiones que se han hecho sin token
- Filtrar por las peticiones que se han hecho con un token concreto
- Incluir todas las peticiones, ya hayan usado un token o no.

Para la realización de los gráficos se ha usado chartjs [23].

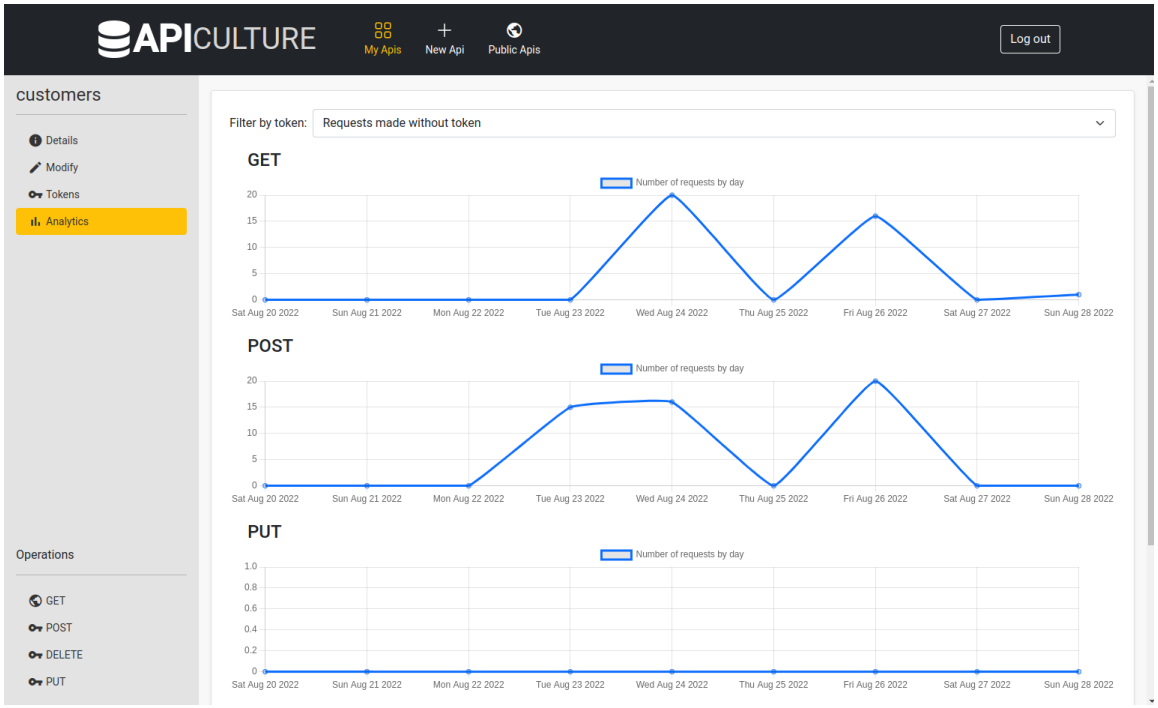

Figura 3.22. Cliente: estadísticas de uso de una API

3.4.8 APIs públicas

A la hora de crear una API hay un campo que es “public”, en el formulario de la Figura 3.12 se muestra como “public listing”, esta opción hace que la entidad API que hemos creado pueda ser vista por cualquiera, es decir, seleccionando esta opción, nuestra API podrá ser servida en 2 ocasiones:

- GET a “/api/nombreApi”, donde nombreApi es el nombre de nuestra API. Esta ruta normalmente solo es accesible incluyendo una sesión de usuario o un token de acceso en las cabeceras de autorización, pero si nuestra API es pública, esta ruta es siempre accesible.
- GET a “/api/public”, donde obtenemos una lista de todas las APIs públicas.

Esta opción es interesante, ya que de esta forma la aplicación no solo te permite crear APIs, sino también compartirás y darles visibilidad. Además, es posible que

accedan nuevos usuarios que no quieran crear APIs como tal, ni tan siquiera registrarse, sino que quieren usar una de estas APIs públicas, y accedan a la página web para ello.

Por eso, desde el cliente web, tenemos una vista donde se mostrarán todas estas APIs, a la que pueden acceder tanto usuarios logeados como los que no lo están. En la Figura 3.23 tenemos la vista para un usuario que no está logeado.

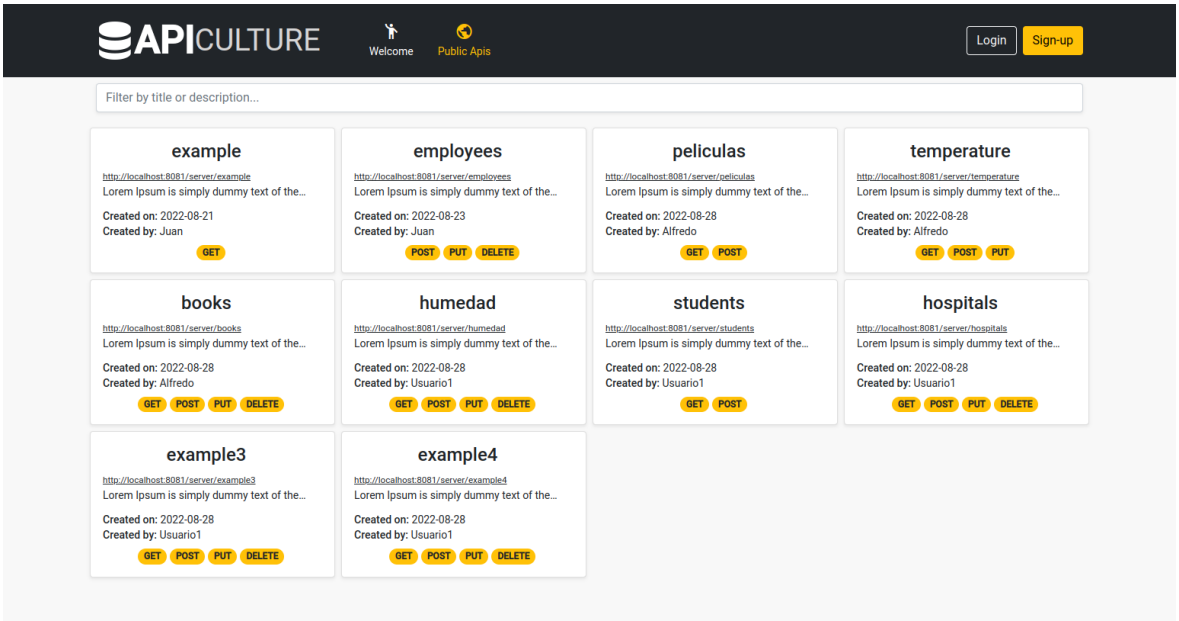

Figura 3.23. Cliente: APIs públicas

Para cada API se muestra un cuadrado con información sobre la misma. Las operaciones que se muestran en la parte inferior del recuadro son solo las operaciones públicas. Si hacemos click en una de estas operaciones se nos muestra un diálogo que nos permite realizar la petición al estilo Postman, es lo mismo que lo expuesto en la sección 3.4.6.2, pero en este caso cualquiera puede acceder, y solo funciona con operaciones que no requieren autenticación; un ejemplo de esto se muestra en la Figura 3.24, donde hemos pulsado el botón GET de la API “customers”.

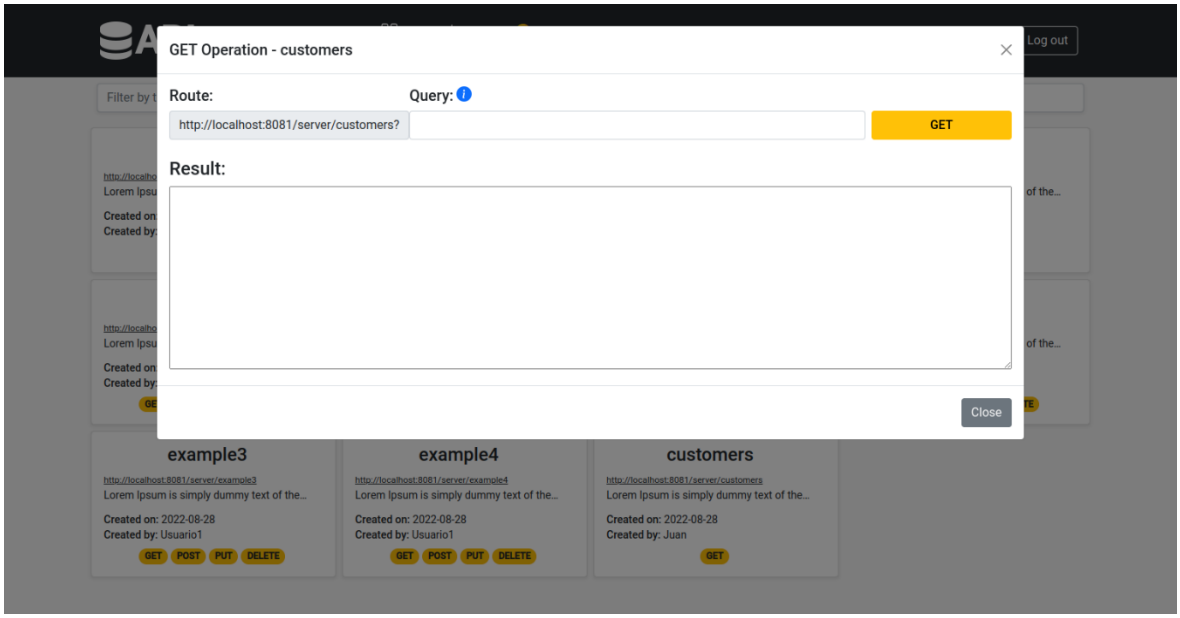

Figura 3.24. Cliente: realizar operación GET sobre una API pública

Si hacemos click en el nombre de la API o en su descripción se nos mostrará otro diálogo con los detalles de la API, esto puede verse en la Figura 3.25 para el mismo ejemplo de customers.

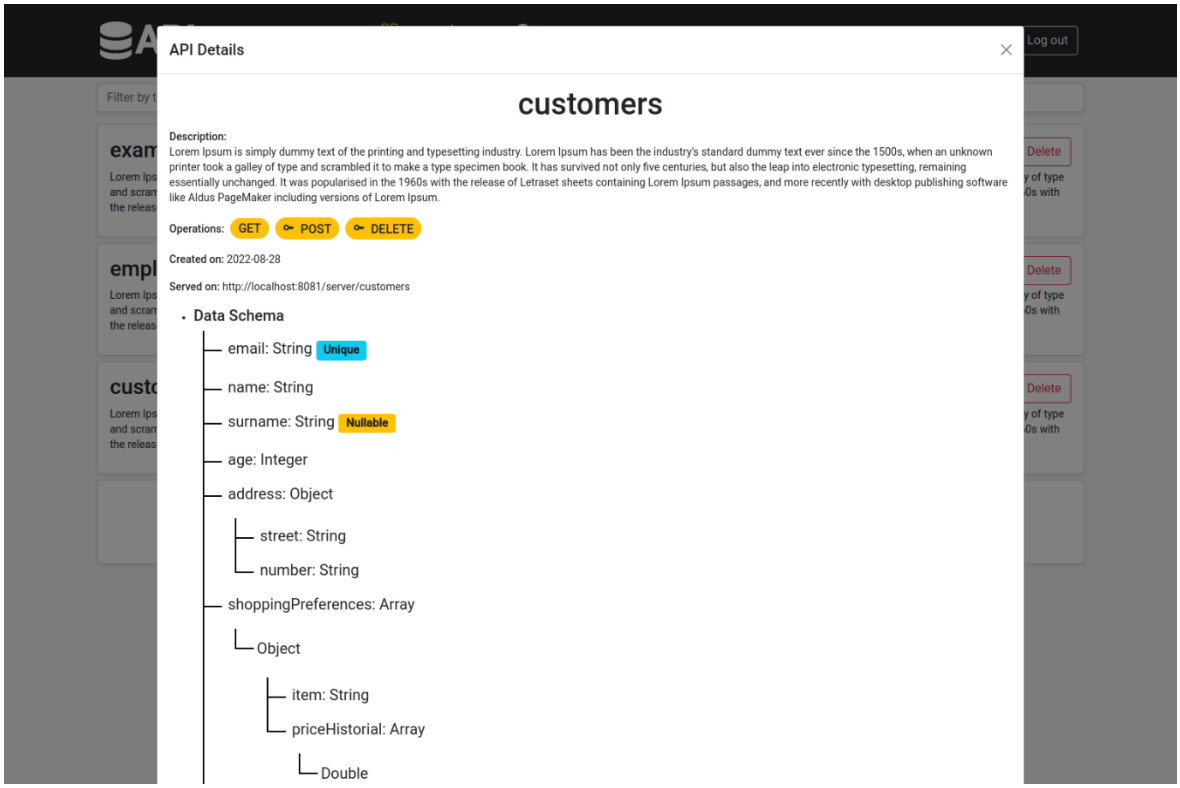

Figura 3.25. Cliente: diálogo con los detalles de una API

4. Conclusiones

En principio la propuesta de proyecto de la empresa ha sido llevada a cabo, incluyendo todas las funcionalidades iniciales previstas.

Además, ha supuesto un gran aprendizaje, sobre todo por la parte de Angular, que nunca había tocado, por TypeScript y por hacer la Interfaz de usuario en Bootstrap, que requiere de mayor esfuerzo que otras librerías de componentes gráficos.

El hecho de llevar una planificación casi semanal con una persona haciendo el rol de Product Owner también ha supuesto un aprendizaje a nivel práctico de SCRUM.

Un reto de este proyecto ha sido el gran componente dinámico de la aplicación, y el hecho de que se incluyan varias definiciones de datos recursivas a lo largo del código. El otro reto ha sido hacer la memoria, ya que no estoy acostumbrado a escribir tanto texto.

4.1. Futuras líneas de trabajo

La aplicación, en su estado actual, tiene muchas mejoras que se le pueden hacer, algunas de las cuales las pensé en un principio, pero decidí dejar fuera del proyecto porque el tiempo es limitado.

- Permitir una especie de operación GET por websockets, de forma que un cliente se pueda suscribir y ser notificado cada vez que ciertos datos cambien, se debería de poder incluir una query para seleccionar solo los datos a los que se quieren suscribir.
- Guardas más estadísticas relativas a las APIs para que los usuarios las puedan ver, como por ejemplo, guardar los status de las respuestas, y mostrarlos en un gráfica por el lado del cliente.
- Permitir securizar no solo a nivel de una operación de una API, sino a nivel de ciertos datos dentro de una API, por ejemplo, a través de una query, que permita seleccionar ciertos datos de la API que se quieran proteger mediante tokens.
- Añadir varios niveles de autorización.
- Permitir algún otro tipo de autorización a las APIs, que permita la creación de usuarios y su registro de alguna forma; Firebase [25] tiene algo de ese estilo.

- Permitir relaciones N:M dentro de una API, ya que actualmente solo ser permite 1:M, tal vez permitiendo crear varias APIs que tengan relaciones N:M entre ellas.
- Añadir un nivel de administrador a la aplicación, desde el cual se puedan administrar a los usuarios y sus APIs.

4.2. Agradecimientos

Gracias a los profesores de la Universidad de León, y en especial a José Alberto y a Martín Bayón por haberme llevado este proyecto y haberme permitido compaginarlo con el curso de IOT de la Universidad de León.

Gracias a Guillermo Ménguez, compañero de HP SCDS que propuso la idea del TFG y que me ayudó a llevarla a cabo con la parte técnica y la planificación de tareas.

Por último, gracias a mis padres y amigos.

5.Lista de referencias bibliográficas

[1] << Observatorio Tecnológico HP >> [Online]. Available: https://hpscds.com/observatorio-hp/

[2] << The Scrum Guide >> [Online]. Available: https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf

[3] << Sprints de scrum >> [Online]. Available: https://www.atlassian.com/es/agile/scrum/sprints

[4] << Historias de usuario >> [Online]. Available: https://es.wikipedia.org/wiki/Historias_de_usuario

[5] << Using the Fibonacci scale in Agile estimation >> [Online]. Available: https://www.lucidchart.com/blog/fibonacci-scale-for-agile-estimation

[6] << TypeScript >> [Online]. Available: https://es.wikipedia.org/wiki/TypeScript

[7] << Cliente-servidor >> [Online]. Available: https://es.wikipedia.org/wiki/Cliente-servidor

[8] << Build a REST API with Node.js, Mongoose & TypeScript >> [Online]. Available: https://github.com/TomDoesTech/REST-API-Tutorial-Updated

[9] << Zod >> [Online]. Available: https://github.com/colinhacks/zod

[10] << Generalidades del protocolo HTTP >> [Online]. Available: https://developer.mozilla.org/es/docs/Web/HTTP/Overview

[11] << Method Definitions>> [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[12] F. Mörk, << What is the exact definition of "Token?" >> [Online]. Available: https://stackoverflow.com/questions/4448661/what-is-the-exact-definition-of-token

[13] << Introduction to JSON Web Tokens >> [Online]. Available: https://jwt.io/introduction

[14] << Online RSA Key Generator >> [Online]. Available: https://travistidwell.com/jsencrypt/demo/index.htm

[15] << What is the unix time stamp? >> [Online]. Available: https://www.unixtimestamp.com/

[16] << api-query-params >> [Online]. Available: https://www.npmjs.com/package/api-query-params

[17] << A Universally Unique IDentifier (UUID) URN Namespace >> [Online]. Available: https://www.ietf.org/rfc/rfc4122.txt

[18] << uuid >> [Online]. Available: https://www.npmjs.com/package/uuid

[19] J.Kurose y K.Ross, << Chapter 2 slides, computer networking: a top-down >> [Online]. Available: http://gaia.cs.umass.edu/kurose_ross/ppt.php

[20] << node.bcrypt.js HTTP >> [Online]. Available: https://www.npmjs.com/package/bcrypt

[21] << Angular Multiselect Dropdown >> [Online]. Available: https://www.npmjs.com/package/ng-multiselect-dropdown

[22] << Postman >> [Online]. Available: https://www.postman.com/

[23] << Chart.js >> [Online]. Available: https://www.chartjs.org/

[24] << El patrón Model-View-ViewModel >> [Online]. Available: https://docs.microsoft.com/es-es/xamarin/xamarin-forms/enterprise-application-patterns/mvvm

[25] << Firebase >> [Online]. Available: https://firebase.google.com/?hl=es-419

[26] << Gitlab >> [Online]. Available: https://gitlab.com/HP-SCDS

[27] << Status Code Definitions >> [Online]. Available: https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

[28] << beneficio industrial >> [Online]. Available: https://dpej.rae.es/lema/beneficio-industrial

[29] << Reglamento General de Protección de Datos >> [Online]. Available: https://www.boe.es/doue/2016/119/L00001-00088.pdf

[30] << Ley Orgánica de Protección de Datos >> [Online]. Available: https://www.boe.es/eli/es/lo/2018/12/05/3/con