
Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

# Aplicación para monitorización y tratamiento de pacientes con síndrome de intestino irritable

*Application for monitoring and treating patients with Irritable Bowel Syndrom*

Alexis Rodríguez Casañas

La Laguna, 29 de junio de 2021

D. **Casiano Rodríguez León**, con N.I.F. 42020072S profesor Catedrático de Universidad del área de Lenguajes y Sistemas Informáticos, adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

D. **Coromoto León Hernández**, con N.I.F. 78605216W profesora Catedrático de Universidad del área de Lenguajes y Sistemas Informáticos, adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

**I N F O R M A (N)**

Que la presente memoria titulada:

"*Aplicación para monitorización y tratamiento de pacientes con síndrome de intestino irritable*"

ha sido realizada bajo su dirección por D. **Alexis Rodríguez Casañas**, con N.I.F. 54054901W.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 29 de junio de 2021

# Agradecimientos

Con este trabajo, pongo punto y final -al menos hasta el horizonte que llego a vislumbrar- a mi vida académica. Quisiera agradecer en primer lugar, a mis tutores Casiano Rodríguez León y Coromoto León Hernández por su continuo acompañamiento, preocupación y grado de compromiso tanto en el plano personal como académico. Quisiera también reconocer la labor y agradecer a todo el profesorado de la Escuela Técnica Superior de Ingeniería y Tecnología que en esta época de pandemia tan peculiar que nos ha tocado vivir, ha sabido sobreponerse y adaptarse con rapidez para minimizar el impacto sobre el alumnado. Si quisiera agradecer a las personas responsables de que este trabajo sea posible, tendría que mencionar a todos los profesores que he conocido a lo largo de todas las etapas de mi vida académica y que en mayor o menor medida me han ayudado y motivado para llegar hasta aquí. Pero en realidad, eso tampoco sería suficiente. Durante el transcurso de mis estudios, he utilizado material que hombres y mujeres anónimos de todos los lugares del planeta, sin anhelo de recibir nada a cambio se han tomado el tiempo de elaborar y compartir libremente. Sin que nunca lleguen a saberlo, ellos y ellas también forman parte de este hito en mi vida. Por eso, quiero utilizar estas líneas para reconocer la figura del profesor, y dedicarlas a todos los hombres y mujeres del mundo, de todas las culturas y lenguas, los que alguna vez conoceré y los que no, que cada día se involucran en compartir su saber. Porque el conocimiento nos hace libres, nos hace avanzar, y es la única cosa que cuando se regala, se multiplica.

# Licencia

**Resumen**

*El síndrome del intestino irritable es un trastorno que afecta el intestino grueso y que provoca molestos y dolorosos síntomas a quienes lo padecen. Es un trastorno crónico y actualmente la dieta es el tratamiento más efectivo que se conoce. Sin embargo, esta dieta tiene el inconveniente de que son numerosas las variables a tener en cuenta, ya que los síntomas del intestino irritable son producidos por numerosos compuestos. Esto hace que seguir el plan de alimentación sea prácticamente imposible para las personas que lo padecen y aquellas que lo intentan pierden una enorme cantidad de tiempo realizando cálculos. El objetivo de este trabajo es desarrollar una aplicación informática que facilite los cálculos y ofrezca una interfaz de usuario amigable que ayude a mejorar la calidad de vida de los pacientes de esta afección.*

**Palabras clave:** FODMAP, síndrome del intestino irritable, app, móvil, NodeJS, Ionic, microservicios

**Abstract**

*Irritable bowel syndrome is a disorder that affects the large intestine and causes uncomfortable and painful symptoms for those who suffer from it. It is a chronic disorder and diet is currently the most effective treatment known. However, this diet has the drawback that there are many variables to take into account, since irritable bowel symptoms are caused by numerous compounds. This makes following the eating plan virtually impossible for people who suffer from it, and those who do waste an enormous amount of time doing calculations. The objective of this work is to develop a computer application that facilitates calculations and offers a friendly user interface that helps to improve the quality of life of patients with this condition.*

**Keywords:** FODMAP, irritable bowel syndrome, app, mobile, NodeJS, Ionic, microservices

# Índice general

# Índice de Figuras

# Índice de Tablas

# Capítulo 1

# Introducción

El objetivo de este capítulo es explicar en qué consiste el síndrome del intestino irritable, la definición del estado del arte en relación a las aplicaciones informáticas asociadas y la introducción de los conceptos técnicos básicos y la metodología de trabajo seguida para el desarrollo del proyecto.

## 1.1. Contexto médico

El Síndrome del Intestino Irritable (SII) es un trastorno caracterizado clínicamente por la asociación de dolor abdominal y alteraciones en la frecuencia o consistencia de las deposiciones, sin una causa clara (incluida la enfermedad celíaca) en los estudios y exploraciones realizadas. Si bien el origen del SII y los mecanismos implicados no han sido completamente esclarecidos, existe una creciente evidencia sobre la existencia de alteraciones estructurales objetivas en el intestino de los pacientes con SII; como inflamación, alteración de la barrera epitelial, hipersensibilidad neuronal y cambios en la composición de la flora intestinal. El SII afecta aproximadamente a un 11 % de la poblacion mundial. Muchos pacientes con SII relacionan sus síntomas con la ingesta de alimentos. Existe ya evidencia científica sólida que ha demostrado que estas alteraciones orgánicas anteriormente mencionadas y/o los síntomas del SII pueden ser provocados por los alimentos. De igual manera, la eliminación de determinados alimentos de la dieta puede mejorar significativamente los síntomas de los pacientes con SII, especialmente cuando la diarrea, la distensión y el dolor abdominal son los síntomas predominantes. Sin embargo, puede ser difícil o imposible para los pacientes identificar con exactitud qué alimentos o componentes de los alimentos pueden ser los responsables de sus síntomas. Además, el acceso a dietistas expertas o fuentes de información fiables sobre dietas en la práctica clínica puede ser muy limitado y no existen pruebas diagnósticas validadas para el estudio de las intolerancias alimentarias. En la actualidad, la única manera de identificar los alimentos desencadenantes de síntomas en el SII es la eliminación y reintroducción de los mismos, como la dieta sin gluten y la dieta con bajo contenido en hidratos de carbono fermentables (FODMAP). La dieta FODMAP es un tipo de dieta de exclusión que consiste en reducir la ingesta de alimentos que contienen fructosa, lactosa, galactanos, polioles, fructanos y galacto-oligosacáridos [7, 9], para posteriormente reintroducirlos de forma incremental. Las reintroducciones de alimentos deben ser individuales y progresivas. En la dieta FODMAP, la reintroducción se hará por grupos de alimentos con alto contenido en los diferentes compuestos identificados como problemáticos. Esta estrategia tiene como

objetivo identificar qué alimentos y hasta qué cantidad de los mismos puede tolerar el paciente.

Aunque determinar el nivel FODMAP de un alimento es un proceso que aún se encuentra en vías de estudio y se ha comprobado que es más complejo de lo que se pensaba en un principio, ya que depende de la naturaleza del alimento, su proceso de elaboración u otras variables, se conocen algunas reglas que ayudan a determinar el nivel FODMAP de un alimento:

- Más de 300 mg de sorbitol o manitol
- Más de 4000 mg de lactosa
- Más de 300 mg de fructo-oligosacáridos o galacto-oligosacáridos

Si una ingesta no cumple ninguna de las condiciones anteriores, se considera segura desde el punto de vista FODMAP y se representa con un semáforo verde. Si cumple una de las condiciones, se debe tomar con precaución y se representa con un semáforo amarillo. Si por el contrario cumple dos o más condiciones, la ingesta se considera inadecuada y se representa con un semáforo rojo. Nótese que hay compuestos que se encuentran en el mismo grupo y no añaden un nivel de restricción más a la ingesta. Por ejemplo, si el alimento posee más de 300 mg de sorbitol y también más de 300 mg de manitol, solamente se está incumpliendo una condición (la primera) y por tanto el semáforo del alimento sería amarillo y no rojo [1].

## 1.2. Objetivos

El objetivo de este trabajo es diseñar y desarrollar una aplicación informática funcional que ayude a las personas a realizar un seguimiento y cálculo de la dieta FODMAP, además de realizar un estudio del estado del arte y conocer la disponibilidad de los datos nutricionales necesarios para elaborar este tipo de aplicaciones. A nivel funcional, se apunta a conseguir los siguientes objetivos concretos:

- La aplicación debe ajustar las recomendaciones de FODMAP en base a las preferencias o características del usuario. No debe limitarse a ser una simple tabla genérica.
- Sería positivo e innovador que la aplicación contase con el concepto de platos o menús, recogiendo y tratando como uno solo a un conjunto de alimentos o ingredientes.
- La aplicación debería contar con un escáner de código de barras que proporcionase algún tipo de información sobre alimentos comerciales que se encuentran en el súpermercado.
- La aplicación debería ser fácilmente mantenible y escalable, permitiendo un desarrollo ágil y la adición de nuevas características o actualizaciones de forma sencilla.

Al contrario que una aplicación empresarial, se trata de un proyecto con un grado de incertidumbre alto. El problema a tratar, las fuentes de datos, la lógica de negocio

[1] https://onlinelibrary.wiley.com/doi/full/10.1111/jgh.13698

o algunas de las tecnologías utilizadas, son puntos que tienen el mismo denominador común: un estado de inmadurez considerable. Por ello, se trata de un proyecto muy experimental y resulta fundamental tener en cuenta la escalabilidad, la limpieza y una buena arquitectura, puntos que se tratarán más adelante.

## 1.3. Antecedentes y estado actual del arte

Para obtener información sobre el estado del arte se consultó en los principales buscadores de recursos académicos utilizando la siguiente consulta de búsqueda:

```
TITLE-abs-KEY((((meal OR symptom) AND( logging OR tracking) AND ( crohn’s OR bowel ) )
OR ibs AND fodmap OR ( intolerances OR ( food AND allergies ) )
OR ( bowel OR food intolerance OR ibs OR allergies OR crohn’s ) )
AND ( app OR mobile application ) )
AND ( limit-TO ( language,"English" ) )
```

Tras analizar los resultados, se observa que desde el punto de vista médico se trata de un tema aún sin explotar, sobre el que no existe un excesivo número de estudios y los que existen son muy recientes. En España, concretamente, es un tema muy joven del que apenas se está comenzando a hablar en la medicina. Por ello, es lógico que desde el punto de vista tecnológico parece que aún no se hayan realizado grandes trabajos y apenas existan artículos o proyectos relacionados.

De las pocas aplicaciones que existen, se han analizado las principales que poseen un mayor número de descargas y la retroalimentación de los usuarios para extraer un cuadro de características comunes y valoradas o solicitadas (véase la Figura 1.1). La app FODMAP de Monash SÍ admite Personalización por usuario

| | FOODMAP by Monash | FOODMAP Helper | Low FOOD | My Food Intolerance List |
|---|---|---|---|---|
| OCR | No | No | No | No |
| Social | No | No | No | No |
| Escáner de código de barras | No | No | No | No |
| Personalización por usuario | Sí | Sí | Sí | No |
| Alimentos españoles | No | Algo | Algo | No |

Figura 1.1: Apps líderes en descargas y las características más solicitadas por los usuarios

Si bien toda característica añadida siempre es deseable y suma valor, debemos desglosar y priorizar correctamente aquellas que verdaderamente marcan la diferencia entre el posible éxito o fracaso de una aplicación. Así, del análisis funcional de las herramientas estudiadas junto con la opinión de sus usuarios en España, se extraen principalmente tres conclusiones:

- Prácticamente todas las apps tienen comentarios negativos sobre su interfaz. Parece que ninguna ha sabido acertar de lleno a la hora de ofrecer una interfaz usable, simple y cómoda.
- Dado que en España y Europa aún no se ha explotado este tema, las apps existentes no cuentan con una base de datos de alimentos orientadas a estas localizaciones, haciendo muy difícil a sus habitantes sacar provecho de las herramientas.
- La mayoría de aplicaciones existentes no ofrecen personalización al usuario y las que lo hacen apenas permiten ajustar algunas preferencias. Las aplicaciones no realizan un seguimiento del usuario ni realizan recomendaciones o poseen algún tipo de memoria para sus preferencias sino que se limitan a ser meros visualizadores de tablas estáticas.

## 1.4. Algunas definiciones

A lo largo de este proyecto se utiliza un gran abanico de tecnologías a todos los niveles implicados en una pila de desarrollo. Por ello, merece mencionar brevemente y sentar la base de algunos conceptos que cobrarán protagonismo a medida que avanza la lectura.

En diseño de software el *front-end* es la parte del software que interactúa con los usuarios y el *back-end* es la parte que procesa la entrada proveniente del *front-end*. La separación de un sistema software en estas dos partes es un tipo de abstracción que ayuda a mantener las funcionalidades del sistema separadas. En general el *front-end* es el responsable de recolectar los datos de entrada del usuario, que pueden ser de muchas y variadas formas, y los transforma adecuándolos a las especificaciones que demanda el *back-end* para poder procesarlos, devolviendo generalmente una respuesta que el *front-end* recibe y muestra al usuario de una forma entendible para este. En diseño web el *front-end* hace referencia a la visualización del usuario navegante por un lado, y por el otro, el *back-end* se refiere a la visualización del administrador del sitio. La conexión del *front-end* con el *back-end* es un tipo de interfaz (*interface*).

Una API (*Aplication Programming Interface*) [12, 13] no es nada más que una definición de cómo una pieza de software se puede comunicar con otra pieza de software, esto es, define qué formatos de datos y protocolos se utilizan para que dos piezas de software se comuniquen entre ellas. En términos técnicos, una API define la forma en que dos módulos de software que no se conocen entre sí se intercambian información. La API define un conjunto de protocolos, reglas y formato de los datos, creando un estándar que pueden usar para comunicarse entre sí diferentes sistemas.

REST (*Representational State Transfer*) es un estilo de arquitectura software utilizado a la hora de diseñar una API. REST tiene una serie de características que le han llevado a ser la arquitectura favorita de muchos desarrolladores hoy en día, tanto por motivos técnicos como de facilidad de implementación y uso. Si un sistema implementa correctamente las reglas de la arquitectura REST, se dice que este es RESTFUL.

REST es un conjunto de principios de diseño para construir servicios web escalables. Estos principios son los siguientes:

- Interfaz uniforme: la arquitectura está fuertemente desacoplada. Los módulos comparten una interfaz común que les permite comunicarse. Los recursos son identificados por una URI (*Uniform Resource Identifier*). Cada recurso tiene su propia URI,

pero los recursos reales son separados de su representación, es decir, el servidor no envía al cliente información sobre la base de datos que almacena un recurso, sino que se limita a enviar una representación del recurso, normalmente en formato JSON (*JavaScript Object Notation*).

- Cliente-servidor: el diseño basado en comunicación cliente-servidor ayuda a desacoplar la arquitectura. El servidor o el cliente no necesitan conocer el uno sobre el otro los detalles de implementación, tecnologías o lenguajes que utilizan.
- Sin estado (*stateless*): la inexistencia de estado es una de las claves del diseño RESTFUL. Significa que un servidor no necesita recordar el estado de la aplicación cliente. Toda la información relevante sobre el contexto es incluida en el mensaje que se intercambia.
- Caché: el servidor puede indicar al responder con un recurso, si dicho recurso se puede almacenar o no en la memoría Caché. En caso de serlo, puede acompañar la respuesta de un tiempo de vida del recurso. El cliente es responsable de utilizar este dato para saber si realizará una nueva petición o utilizará el recurso almacenado en la Caché.
- Sistema en capas: el principio de sistema dividido en capas es fundamental y permite la introducción de módulos que se encarguen de la seguridad o el balanceo de carga, entre otras tareas. Para el cliente, toda la arquitectura y funcionamiento del *back-end* es transparente.

En una API REST se trabaja con recursos. Un recurso es un concepto de negocio que puede ser identificado mediante una URI. Cualquier cosa susceptible de ser nombrada puede ser un recurso: un producto, una persona, la reserva de un hotel, etc. Normalmente, estos recursos son representados utilizando JSON. Para manipular y consultar estos recursos se utiliza el protocolo HTTP (*Hypertext Transfer Protocol*) con sus verbos POST, GET, PUT y DELETE, asociados respectivamente con las operaciones de creación, lectura, modificación y borrado. A este conjunto de operaciones básicas que se pueden realizar sobre los recursos se le conoce como CRUD (Create, Read, Update, Delete).

REST propone que las URIs deben asociarse al recurso y no a la acción sobre dicho recurso. Respetar esta regla produce APIs predecibles, escalables y fácilmente mantenibles. En una API bien diseñada, el desarrollador que la consume prácticamente no necesita consultar su documentación, porque su comportamiento es un estándar bien conocido. Deben evitarse URIs como getFoodByID o deleteFood. Colocar nombres de operaciones en las URIs es uno de los errores de principiante más comunes.

En la siguiente tabla de ejemplo, se muestra parte de la API del sistema desarrollado en este trabajo. Nótese como la URI representa un recurso, en este caso un alimento, y cómo es el método HTTP quien define la operación sobre el recurso.

| Verbo | URI | Descripción |
|---|---|---|
| POST | /foods | Crea y almacena un nuevo alimento en el sistema. La información del alimento a crear viaja en el cuerpo de la petición |
| GET | /foods | Obtiene todos los alimentos. |
| GET | /foods/:id | Obtiene el alimento con el id proporcionado. |
| PUT | /foods/:id | Modifica el alimento con el id proporcionado. La información a modificar viaja en el cuerpo de la petición. |
| DELETE | /foods/:id | Elimina el alimento con el id proporcionado. |

Un recurso, habitualmente se representa en JSON, pero esto no es obligatorio, ya que otra característica de las APIs REST es lo que se conoce como negociación de contenido, donde el cliente especifica en su petición el formato de datos preferido. Nótese que esto no implica que el servidor pueda satisfacer dicho requisito. Cuando se solicita un recurso, el servidor lo obtiene de la base de datos, normalmente almacenándolo en alguna estructura de datos del lenguaje que implemente y lo envía al cliente en un formato estándar independiente de dicho lenguaje. Este proceso se conoce como serialización. Un ejemplo de respuesta al solicitar un alimento, podría ser el siguiente:

```
food: {
    name: "pan blanco"
    lactose: 400,
    fos: 0,
    gos: 0,
    manitol: 0,
    maltitol: 0,
    sorbitol: 0,
    xilitol: 0,
    fodmap: {
        oligo: 0,
        lactose: 0,
        polyols: 0,
        level: 0
    }
}
```

Nótese que la serialización es transparente del sistema de persistencia utilizado. Tanto si se utiliza una base de datos SQL, NoSQL, en memoria o un simple fichero de texto como unidad de persistencia, la respuesta de la API será exactamente la misma. Sin embargo, en una base de datos NoSQL basada en colecciones como es el caso de MongoDB, utilizada en este proyecto, el formato de la información almacenada coincide exactamente con su representación, haciendo el proceso de serialización más simple e inmediato y también más natural para el desarrollador.

Respecto a las arquitecturas de back-end, actualmente se menciona mucho la “arquitectura de microservicios” la cual es un enfoque para desarrollar una aplicación software como una serie de servicios, cada uno ejecutándose de forma autónoma y comunicándose entre sí. Los microservicios poseen numerosas características atractivas, aunque no son todo ventajas.

En un entorno real, los microservicios presentan importantes retos y a veces es posible pecar de hacer sobreingeniería cuando realmente no es necesario. Es por ello que hay quien piensa que el nombre microservicio es engañoso, ya que hace pensar que se debe tratar de una pieza de software realmente pequeña, cuando en realidad es más importante el concepto de ser un servicio autocontenido y aislado, independientemente de su tamaño.

Respecto a la persistencia de datos, el término NoSQL se refiere a la denominación en inglés Not Only SQL. Plantea modelos de datos específicos de esquemas flexibles que se adaptan a los requisitos de las aplicaciones más modernas. Existen muchos tipos de bases de datos NoSQL, pero las más utilizadas suelen ser las orientadas a documentos. En una base de datos SQL o relacional, la información se estructura en tablas relacionadas entre sí, mientras que en una base de datos NoSQL la información se almacena en forma de

documentos que ya contienen toda la información necesaria sin necesidad de realizar uniones con otros fragmentos de información.

A lo largo de este proyecto también será muy importante seguir una metodología de desarrollo ágil. Una metodología de desarrollo es un marco o forma de trabajo que es usado para estructurar, planear y controlar el proceso de desarrollo en sistemas de información. En el caso concreto de las metodologías ágiles, son aquellas que permiten adaptar la forma de trabajo a las condiciones del proyecto, consiguiendo flexibilidad e inmediatez en la respuesta para amoldar el proyecto y su desarrollo a las circunstancias específicas del entorno. En el desarrollo moderno, donde es posible maquetar rápidamente un prototipo de aplicación estas metodologías han cobrado mucha importancia, ya que permiten dejar de lado el peso de la gestión o la planificación hasta un punto y centrarse en lo verdaderamente importante, que es el desarrollo y la consecución de los objetivos.

Otro punto a considerar en este proyecto es la virtualización basada en contenedores. En este enfoque, el kernel del sistema operativo se ejecuta sobre el nodo de hardware con varias máquinas virtuales (VM) invitadas aisladas, que están instaladas sobre el mismo. Los huéspedes aislados se denominan contenedores La principal diferencia con la virtualización tradicional, es que en la contenerización, la virtualización ocurre a nivel de proceso del sistema operativo anfitrión, con la consecuente mejora de rendimiento y recursos frente a la virtualización tradicional que requiere apoyarse en todo un sistema operativo virtual.

Por otro lado, de cara al desarrollo móvil, merece la pena mencionar el concepto de tecnologías móviles híbridas. Las aplicaciones híbridas, a diferencia de las nativas, son desarrolladas con tecnologías web y se ejecutan sobre una capa de abstracción a menudo denominado Webview. En cierto modo, es como si se ejecutaran dentro de un navegador web, el cual se apoya en alguna tecnología de bridge o puente para acceder a las funciones nativas del dispositivo.

Con el crecimiento de estas aplicaciones y el mundo de las aplicaciones web, las llamadas PWA o aplicaciones web progresivas han sufrido un incremento del protagonismo. Las PWA viven a medio camino entre las aplicaciones web, que se caracterizan por estar aisladas en un navegador sin acceso al sistema y las aplicaciones nativas. Así, las PWA presentan muchas ventajas de las aplicaciones nativas, como un mayor rendimiento, acceso al hardware del dispositivo o funcionamiento sin conexión, pero también las ventajas de las aplicaciones web, como la compatibilidad multiplataforma y una mayor facilidad y velocidad de desarrollo.

De cara a las arquitecturas software que se mencionan en este proyecto, merece mencionar las arquitecturas MVC y MVVM. La arquitectura Modelo-vista-controlador (MVC) es un patrón de arquitectura de software, que separa los datos y principalmente lo que es la lógica de negocio de una aplicación de su representación y el módulo encargado de gestionar los eventos y las comunicaciones. Para ello MVC propone la construcción de tres componentes distintos que son el modelo, la vista y el controlador, es decir, por un lado define componentes para la representación de la información, y por otro lado para la interacción del usuario. Este patrón de arquitectura de software se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones y su posterior mantenimiento. MVVM, por el contrario, significa Modelo Vista Vista-Modelo, porque en este patrón de diseño se separan los datos de la aplicación, la interfaz de usuario pero en vez de controlar manualmente los cambios en la vista o en los datos, estos se actualizan directamente cuando sucede un cambio en ellos, por ejemplo si la vista actualiza un dato que está presentando se actualiza el

modelo automáticamente y viceversa. Las principales diferencias entre MVC y MVVM son que en MVVM el Controller cambia a View-Model y hay un binder que sincroniza la información en vez de hacerlo un controlador «Controller» como sucede en MVC.

## 1.5. Metodología de desarrollo

En una primera fase se hizo una búsqueda bibliográfica lo mas exhaustiva posible considerando la posibilidad de hacerla usando la meotodología PRISMA. No hemos hecho un seguimiento riguroso de dicha metodología aunque ha inspirado nuestra forma de trabajo en la determinación de los trabajos de referencia.

Para el desarrollo de este trabajo se han utilizado Metodologías Ágiles con ciclos de reuniones cada dos semanas.

En un intento de seguir la metodología ágil hemos establecido con médicos especialistas en digestivo y nutrición intentando involucrarlos en el papel de *product owner*. Si bien no hemos logrado que - por razones de agenda - las reuniones con ellos tuvieran continuidad sus consejos y opinión han sido tenidos en cuenta en el diseño de la aplicación.

Se ha usado la herramienta de control de versiones git con GitHub para algunas tareas de coordinación. En especial las herramientas de gestión de incidencias y los paneles.

En la realización de este proyecto se ha tratado de mantener una cierta calidad en la solución desarrollada. Para ello, se ha seguido una serie de prácticas a la hora de diseñar e implementar. Si bien en el nivel de la presente titulación muchas de estas prácticas se sobreentienden, no está de más realizar algunos apuntes no tan obvios que comienzan a producir un código más cercano al que se utiliza en el mundo real y que pasan desapercibidos incluso para desarrolladores no tan nóveles.

#### Los objetos son microorganismos vivos

La programación orientada a objetos (en adelante POO) es un paradigma. Esto significa que es una forma de pensar diferente y no una mera técnica de implementación. Algo que muchos desarrolladores tardan años en aprender es que el hecho de utilizar objetos no implica que se esté haciendo POO. Se puede escribir código utilizando clases, herencia, polimorfismo y todas las herramientas asociadas a la POO y sin embargo estar escribiendo un código totalmente procedural.

> *Un objeto no es un almacén de datos. Un objeto es un organismo vivo, como tú y como yo. Y a un ser vivo no se le piden datos, se le pide que realice un trabajo.*
> *- Allen Holub -*

Reflexionar sobre estas palabras es el primer paso para comenzar a cambiar de paradigma y diseñar y escribir código realmente orientado a objetos. Es por ello que numerosos ingenieros de software reconocidos están en contra de los métodos setters, getters y constructores por defecto. Estos métodos exponen la implementación y el estado de la clase, violando el principio del paradigma. En el siguiente ejemplo se muestra un fragmento de código donde se crea un objeto y se le asignan unas propiedades.

```
Dog dog = new Dog();
dog.setName("Budy");
String color = dog.getColor(); //Null!
```

Pese a usar objetos, el código anterior es totalmente procedural. En la vida real, a un perro no se le puede asignar un color ni un número de patas. Esto nos lleva al concepto de **correcto por construcción**. Un perro nace con un color, y de la misma forma un objeto debería nacer con todas sus propiedades inicializadas. Esta forma de trabajar asegura que siempre se tendrá una instancia correctamente inicializada y librará el código de muchísimos errores.

```
Dog dog = new Dog("Buddy", "brown");
```

### Tipado fuerte es más que el uso de tipos primitivos

El uso de tipos aporta cierta seguridad, pero esto yace muy lejos de comprender la verdadera seguridad que nos puede ofrecer el tipado. Se debería trabajar siempre con entidades del modelo de negocio. Deberíamos deshacernos de los tipos primitivos tan pronto como sea posible. No siempre es posible cumplir esta regla, como normalmente ocurre en los controladores o la creación de entidades, pero debería respetarse el máximo tiempo posible. Un servicio o un DAO nunca deberían trabajar con tipos primitivos. Obsérvese el siguiente fragmento de código donde creamos una instancia de un perro y luego tratamos de localizar a su dueño mediante un servicio. Nótese que la segunda línea de código no sería necesaria, pero se supone que estaría en otro punto diferente del programa, donde desconocíesemos la variable owner. Aquí se ha decidido simplificar al máximo el ejemplo para destacar únicamente la esencia de la explicación.

```
Dog dog = new Dog("Buddy", "brwn", owner);
Owner owner = dogService.findOwnerByDogName("buddy");
```

Este código es problemático y de hecho generará errores difíciles de trazar. Para empezar, hemos cometido una errata escribiendo y no persistiremos correctamente el color del animal en base de datos, con los subsecuentes problemas de comportamiento en nuestra aplicación. Cuando busquemos y mostremos todos los perros marrones, esta entidad no aparecerá y no sabremos por qué. Además, tampoco conseguiremos encontrar al propietario de la mascota, ya que hemos escrito "buddy"en minúscula a la hora de usar el servicio, valor que no coincide con el que otorgamos en el constructor. En tan solo dos líneas de código hemos conseguido crear dos problemas. Esto podría solucionarse reescribiendo el código como sigue:

```
Dog dog = new Dog("Buddy", COLORS.BROWN, owner);
Owner owner = dogService.findOwnerByDog(dog);
```

Hemos reescrito el código utilizando entidades de nuestro modelo de negocio, siendo imposible que comentamos un error creando la primera entidad. Para poder utilizar el servicio para encontrar al dueño de la mascota, el código nos obliga primeramente a contar con una instancia de la clase perro; no hay posibilidad de cometer erratas. Pero, si internamente el servicio utiliza el nombre del perro para buscar al dueño ¿no podríamos aún obtener una excepción dentro del mismo si nos hubiésemos olvidado de asignar un nombre a la mascota?. No, al no contar con métodos setters ni constructores por defecto, el objeto "dog"es necesariamente correcto por construcción. Aquí sale a relucir la verdadera potencia del tipado: el propio código simplemente nos impide esribir código incorrecto. El compilador ya no es algo a lo que hay que enfrentarse, sino que se torna un aliado que trabaja para nosotros. Además, los entornos de desarrollo modernos son capaces de inferir y autocompletar el código cuando las opciones son evidentes. Un código

fuertemente tipado y correctamente diseñado prácticamente se escribe solo, a base de autocompletado.

# Capítulo 2

# Tecnologías empleadas

La oferta de tecnologías a la hora de realizar un proyecto hoy en día es realmente extensa. A menudo la elección de estas tecnologías para formar la pila de desarrollo está vagamente justificada y el mayor argumento suele ser la familiaridad de los desarrolladores con las herramientas escogidas. Esto, de hecho, no es un argumento menor, ya que en un entorno real, adquirir los conocimientos suficientes sobre una serie de tecnologías como para crear un producto listo para la explotación necesita una gran cantidad de tiempo por parte del equipo de desarrollo y por ende también supone un importante costo económico. Sin embargo, sí que deberían conocerse los puntos débiles y fuertes de las principales opciones y debería realizarse un análisis para evitar problemas graves en el futuro. En este capítulo se describen las tecnologías empleadas para la realización de este proyecto y la justificación de la elección realizada.

## 2.1. Virtualización mediante Docker

Este proyecto utiliza tecnología de contenerización con Docker, tanto como herramienta de desarrollo como de despliegue. Docker posee numerosas ventajas. En el caso de utilizarse como herramienta de desarrollo:

- Solo se tiene que programar la aplicación una sola vez. Dado que una app en Docker se ejecuta dentro de un contenedor, y el contenedor se puede ejecutar en cualquier sistema operativo que tenga Docker instalado, no se tiene que programar y configurar el software para diferentes tipos de plataformas hardware o sistemas operativos en los que tiene que poder ejecutarse. Solo tienes que programarlo una sola vez para Docker.
- Se obtiene mayor modularidad. El desarrollo con contenedores es ideal para un enfoque basado en microservicios para el diseño de aplicaciones. Bajo este modelo, las aplicaciones complejas se dividen en unidades más discretas y pequeñas. Por ejemplo, y sin llegar a cambiar la arquitectura tradicional de la app, la base de datos quizás se ejecute en un contenedor mientras que el front-end de la misma se ejecuta en otro distinto. Este enfoque hace que la aplicación sea modular, reduciendo la complejidad de tener que mantener y actualizar la aplicación, dado que un error o un cambio relacionado con una parte de la app no requiere que se revise la aplicación completa.

Docker también se utiliza como parte del proyecto a la hora de desplegar:

- Las instancias de Docker son más ligeras. Para desplegar una app como imagen de una máquina virtual, lo más probable es que se necesite incluir un sistema operativo entero en la imagen. Con un contenedor, solo la app y unas cuantas capas de base tienen que ir dentro del contenedor. Esto se traduce en un proceso de montaje más sencillo, y, encima, la capacidad de poder alojar muchos más contenedores en un servidor físico único. Además, arrancan mucho más rápido (en centésimas de segundo) y es posible lanzarlos y cerrarlos automáticamente según las necesidades.
- Los contenedores son ideales y vienen prácticamente de la mano del diseño basado en microservicios, como es el caso de la aplicación presentada. Permite a la aplicación realizar un balanceo de carga o una tolerancia a fallos bajo demanda, lo cual es muy importante de cara a los costes y el rendimiento [**?** ].

## 2.2. Back-end en NodeJS

NodeJS es un entorno de ejecución para JavaScript construido con el motor de JavaScript V8 de Chrome [4]. Las propiedades más característica de Node son su arquitectura orientada a eventos y sobre todo, su diseño basado en un único hilo de ejecución con E/S no bloqueante. La arquitectura de Node consiste en un bucle de eventos que continuamente se encuentra a la espera de ejecución de instrucciones. Coloquialmente se suele conocer como el hilo principal de ejecución. En la Figura 2.1 se muestra la arquitectura básica de NodeJS [15].

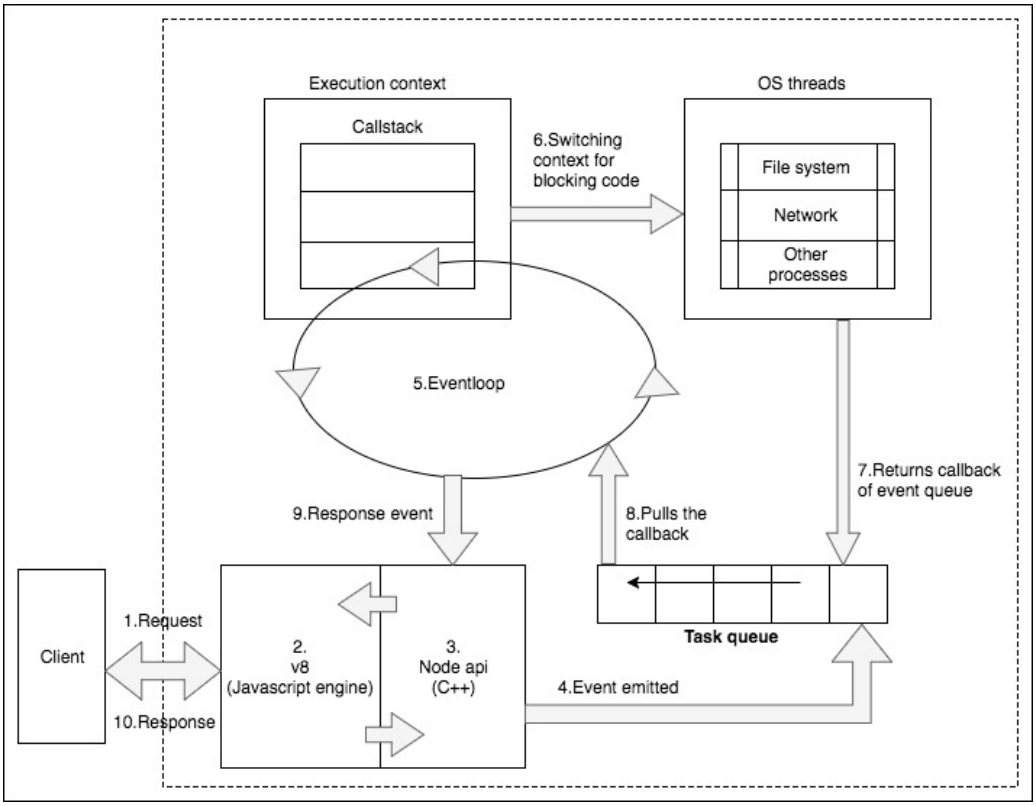


Figura 2.1: Arquitectura básica de NodeJS

Node dispone de una cola de tareas donde almacena los manejadores de las operaciones que utilizan la E/S. Cuando Node encuentra una de estas funciones bloqueantes, no la ejecuta en un segundo thread en paralelo, sino que la envía a la cola de eventos. Cuando

el bucle de eventos se queda sin instrucciones que ejecutar o dicho coloquialmente el hilo principal termina, Node comienza a extraer y ejecutar instrucciones pendientes de la cola de eventos hasta que el hilo principal vuelva a recibir alguna tarea. Este diseño consigue que a pesar de contar con un solo hilo la CPU esté siempre ocupada y permite un gran equilibrio entre rendimiento y coste de recursos.

En una aplicación tradicional que no haga uso de la E/S, la ejecución de un caso de uso es como se muestra en la Figura 2.2

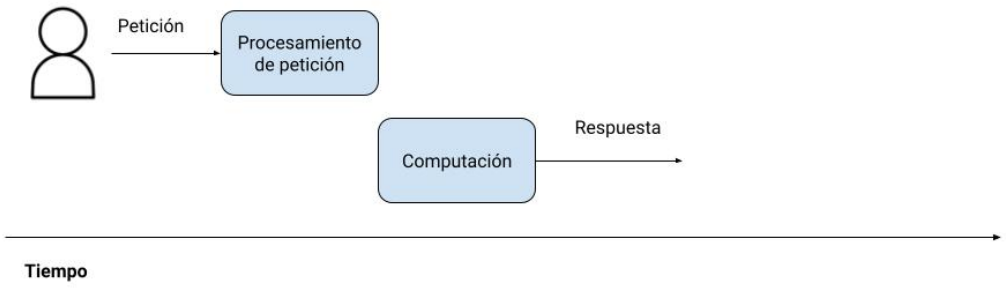


Figura 2.2: Flujo de ejecución de una aplicación sin uso de E/S.

Sin embargo, las aplicaciones web en su inmensa mayoría, y en general cualquier aplicación que utilice operaciones de E/S no funcionan de esa forma. Normalmente, hay un punto donde la aplicación debe consultar a una unidad de persistencia, sea del tipo que sea. El flujo de una aplicación web empresarial típica siempre suele ser el mismo: el usuario envía una request para activar un caso de uso, el servidor procesa la petición, y se hace una operación de lectura o escritura en base de datos. En medio de esto hay algo de computación para la lógica de negocio, pero no suele ser significativa. Esto quiere decir que el 90 del tiempo de ejecución del caso de uso suele ser tiempo de espera por la respuesta de la base de datos. Esto se refleja en la Figura 2.3

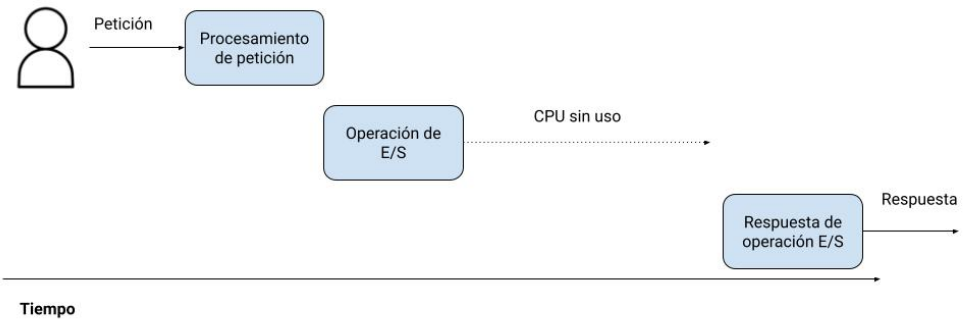


Figura 2.3: Flujo de ejecución normal en las aplicaciones web.

En la Figura 2.4 se muestra como gestiona un servidor Java las peticiones de los usuarios. El servlet crea un hilo para cada request, con el consecuente coste computacional y de memoria. Sin embargo, la CPU no está en uso la mayor parte del tiempo y el programa

se limita a esperar respuesta de la E/S. Pese a ello se consume una gran cantidad de memoria y recursos al crear cada hilo [8].

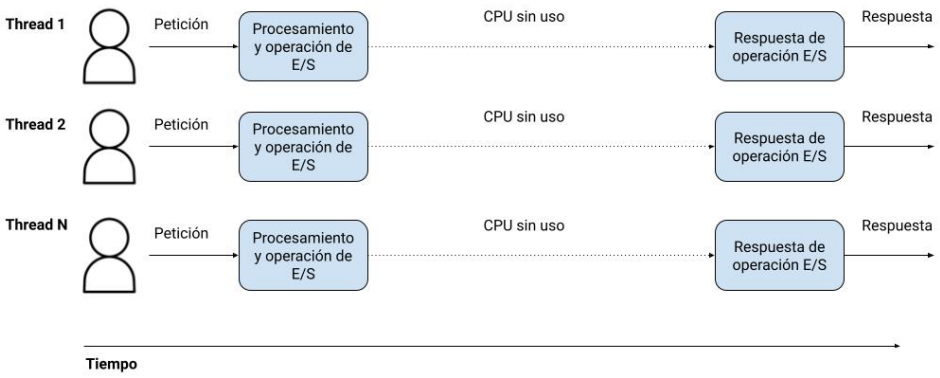


Figura 2.4: Flujo de ejecución en un servidor multihilo.

NodeJS sin embargo, logra los mismos resultados con un consumo de recursos infinitamente menor gracias a que aprovecha el tiempo sin uso de la CPU procesando otras peticiones, como se muestra en la Figura 2.5.

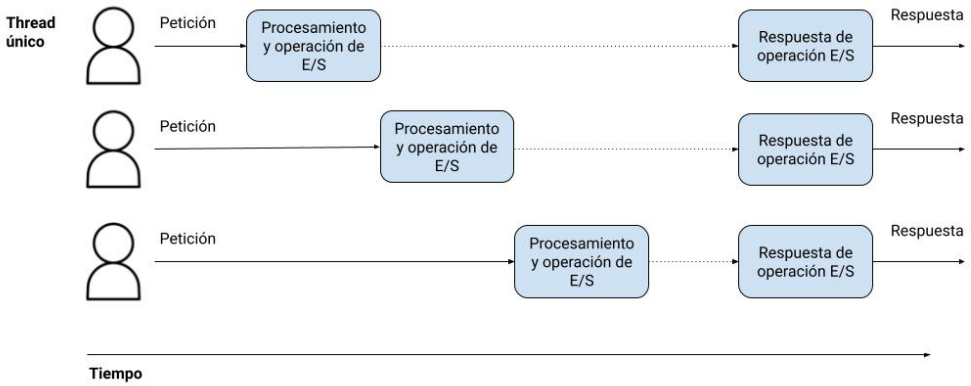


Figura 2.5: Flujo de ejecución en NodeJS.

No debería utilizarse NodeJS si el servidor realizará una lógica de negocio compleja que impleque el uso frecuente de algoritmos de cierto orden de complejidad, ya que la arquitectura monothread es gravemente penalizada en estas circunstancias. Otro aspecto a tener en cuenta es que no se aprovecharán todos los núcleos de la máquina, aunque ya existen técnicas y frameworks para sacarle partido a estas arquitecturas con Node, siempre que se esté dispuesto a pagar el incremento de complejidad.

No debería utilizarse un servidor multihilo y apostar por Node cuando la aplicación posee una lógica de negocio muy ligera o inexistente, y la prioridad es despachar el mayor número de peticiones posible rápidamente.

La aplicación desarrollada implementa gran parte de la lógica de negocio sobre el lado del cliente, haciendo el servidor únicamente de puente entre el mismo y la base de datos. Dicho de otra forma, el papel del servidor es únicamente exponer al cliente la base de datos. Relacionando esto con lo explicado en el 1, resulta fácil ver que estamos ante un caso de uso ideal para el uso de Node.

Node posee numerosos frameworks y herramientas para ayudar en el desarrollo de aplicaciones de todo tipo. Para el presente caso, se ha optado por un framework de microservicios llamado Moleculer.

### Moleculer

MoleculerJS es un framework progresivo de microservicios para NodeJS [3]. Moleculer implementa de serie la estructura básica y comunicaciones de una arquitectura de microservicios, de forma que el desarrollador no tenga que preocuparse por cuestiones tan concretas propias de la arquitectura. Algunas de las características que Moleculer implementa de serie son:

- Soporte para arquitectura orientada a eventos de forma monolítica, de microservicios o mixta.
- Balanceo de carga con varios algoritmos predeterminados como Round Robin, uso de CPU, latencia, entre otros. Compatibilidad con múltiples protocolos de comunicación (TCP, NATS, MQTT, Kafa, entre otros.)
- Soporte para middlewares.
- API Gateway.
- Segurización y autorización.
- Herramientas para medición de métricas y logging.
- Despliegue contenerizado con K8s y Docker Compose.

En la Figura 2.6 se muestra el esquema estándar de una arquitectura básica orientada a microservicios.

Aunque existen variantes, este tipo de arquitecturas suelen tener las siguientes partes básicas:

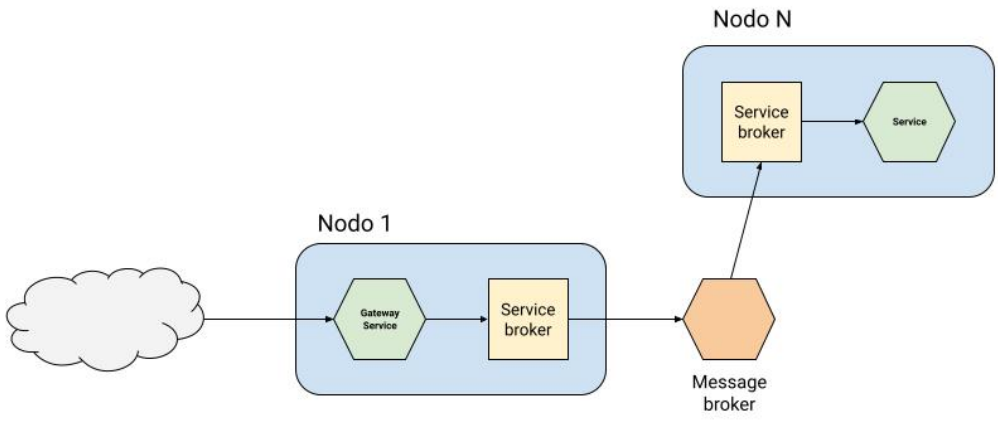


Figura 2.6: Partes principales de una arquitectura de microservicios

- Un API Gateway que balancea la carga y gestiona el tráfico de entrada y salida al sistema, pasando por la securización y autorización entre otras funciones. Moleculer utiliza Traefik.
- Un service broker, encargado de hacer de puerta de entrada y salida entre servicios. El service broker de Moleculer es un módulo relativamente complejo y se encarga de todo lo relativo a las comunicaciones y el contexto del servicio. Incorpora serializadores, validadores, registro de métricas, entre otros. Cada nodo posee su propio service broker.
- Un bus de comunicaciones o message broker, encargado de comunicar los microservicios entre sí de forma eficiente. Por defecto, Moleculer utiliza NATS como protocolo de transporte, aunque es posible configurar muchos otros.
- Uno o más servicios que implementan las funcionalidades de la aplicación.

Moleculer ya implementa de serie los elementos vistos anteriormente (y muchos más) con un gran número de posibles configuraciones. El desarrollador únicamente debe preocuparse de implementar la lógica de negocio en los servicios que desee, y si un servicio desea comunicarse con otro para obtener un dato u ordenarle hacer una tarea, a nivel de implementación esto se hace mediante eventos, de la misma forma en que se trabaja con la famosa clase EventEmitter de NodeJS. Que el método al que se llama pertenezca al mismo servicio, a otro servicio diferente, que el servicio sea local o bien que existan múltiples réplicas del mismo en diferentes localizaciones físicas es totalmente transparente para el desarrollador.

Por todo ello, Moleculer evita reinventar la rueda y partir de una base sólida para comenzar a desarrollar el proyecto sin introducir errores humanos que siempre pueden surgir en las implementaciones manuales.

## 2.3. Persistencia de datos con MongoDB

Debido a la naturaleza y tecnologías del proyecto se ha pensado que una base de datos no relacional (MongoDB) es la mejor opción. MongoDB es una base de datos orientada a documentos: los datos se organizan en documentos, que son estructuras JSON sin un esquema predefinido. Esto dota a MongoDB de una gran flexibilidad y adaptabilidad muy apropiada en los desarrollos ágiles y cambiantes.Las bases de datos documentales almacenan cada uno de los registros y los datos asociados en un solo documento. Cada documento contiene datos semiestructurados que pueden ser consultados con diferentes herramientas de análisis. Estas bases de datos ofrecen mucha flexibilidad, escritura rápida, y consultas rápidas gracias a su gran capacidad de indexación [11].

Sin embargo, existe un enorme número de proyectos que han arrancado utilizando MongoDB en su capa de persistencia y que se han visto obligados a abandonar esta tecnología. Esto no quiere decir que MongoDB sea una mala opción en absoluto, sino simplemente que debemos estar seguros de elegir con criterio y no únicamente por moda, algo que pasa con mucha frecuencia en el mundo del software.

MongoDB es adecuado cuando se utilizan entidades dinámicas sin relación entre ellas o con una relación extremadamente débil, donde la aplicación será la que manejará la entrada y salida de datos. Como se verá más adelante en este presente trabajo, el modelo

de datos necesario es claramente no relacional, actuando como un mero repositorio. Esto es un caso óptimo para el uso de esta tecnología.

Cuando se tiene un modelo de datos relacional, el uso de MongoDB suele traer múltiples complicaciones que casi siempre terminan en desastre. MongoDB no está pensado para hacer "joins", si bien pueden hacerse manualmente en caso de necesidad, aunque es una operación muy ineficiente. En el caso, por ejemplo, de una red social, un usuario publica posts y tiene amigos, que son a su vez también usuarios. Usando colecciones JSON de MongoDB, esto produciría un esquema con dependencias circulares muy ineficiente. Además, si un usuario cambiase de nombre, habría que recorrer y actualizar estos nombres uno por uno en todas las colecciones y documentos donde apareciese.

En base a lo expuesto, se ha dcidido utilizar MongoDB para este proyecto por las siguientes razones:

- Versatilidad. La capacidad de adaptación es uno de los puntos más fuertes las bases de datos NoSQL. La posibilidad de cambios en el "esquema"en tiempo real o la flexibilidad en el modelo de datos les permite responder y adaptarse muy rápidamente a un entorno muy dinámico y cambiante. Esta característica resulta ideal para el presente proyecto donde se ha desarrollado e investigado al mismo tiempo realizando pequeñas iteraciones sobre todo el stack de desarrollo.

- Crecimiento horizontal. Estas bases de datos son altamente escalables en horizontal y suelen venir de la mano de los microservicios. Adoptan muy bien la división del problema en pequeñas instancias de almacenamiento y una comunicación entre las mismas de forma reactiva, logrando el balanceo de carga de la aplicación. Esto se detallará más en el Capítulo 3.

- Naturaleza de los datos. Por supuesto, si nuestro modelo de datos es claramente relacional una base de datos NoSQL puede ser contraproducente. En el caso del presente proyecto, el modelo de datos de muchos de los problemas no es relacional sino que es una suerte de repositorio. Esto encaja perfectamente con el concepto de documentos y colecciones que ofrece MongoDB.

## 2.4. Front-end (app móvil)

### 2.4.1. Ionic

En cuanto al desarrollo de la aplicación móvil, se ha optado por Ionic debido a las siguientes razones:

- La reusabilidad de su código es prácticamente total, ahorrando la necesidad de realizar versiones o adaptaciones del código para cada dispositivo.

- Está basado en frameworks de desarrollo web ya existentes, por lo que si se conocen estos frameworks con anterioridad no se necesita casi invertir tiempo para su aprendizaje. Además, la velocidad de desarrollo es mayor que con otras tecnologías [14].

Los detractores de Ionic argumentan -no sin parte de razón- que al estar basado en web views su rendimiento es menor que el de una aplicación nativa. Sin embargo, para el caso concreto de este desarrollo encontramos al menos dos justificaciones a esta argumentación:

| | React Native | Flutter | Xamarin | Ionic |
|---|---|---|---|---|
| Lenguaje | Javascript | Dart | C# | Web |
| Rendimiento | Casi nativo | Casi nativo | Casi nativo | Moderado |
| Interfaz | Componentes nativos | Componentes nativos | Componentes nativos | Web |
| Comunidad | Muy activa y grande | Poco popular | Moderadamente popular | Bastante popular |
| Reusabilidad | 90% del código | 75% del código | 98% del código | 98% del código |
| Curva de aprendizaje | Fácil. Bastante fácil si se conoce React | Elevada sobre todo si no se conoce | Algo lenta | Muy fácil |
| Aplicaciones | Facebook, Instagram | Alibaba, Google Ads | Olo, MRW | JustWatch, Diesel |

Tabla 2.1: Principales tecnologías utilizadas para el desarrollo de apps móviles junto con las características más valorables

- La aplicación a desarrollar no realizará grandes cómputos ni visualizaciones de forma local y el dispositivo será un mero visualizador que consumirá servicios externos. En estos casos el rendimiento de Ionic es indistinguible de una aplicación nativa.
- Aún si existiera cierta demanda de recursos, el rendimiento de Ionic ha mejorado notablemente con el nuevo bridge, Capacitor, que ha desplazado al anterior Apache Cordova. Este bridge optimiza mucho los recursos del web view logrando una sensación que está muy cerca de la que puede producir una aplicación nativa en la mayoría de los casos.
- Otro factor que aunque sea un beneficio indirecto es importante mencionar, es que con cada versión de Ionic no solo esta tecnología ha mejorado sino que los dispositivos móviles son cada vez más potentes, paliando la penalización del uso de un webview. A fecha de realización de este trabajo, el rendimiento de una aplicación con Ionic Capacitor en un dispositivo de gama media, es indistinguible del que proporciona una aplicación nativa, siempre que no se exija una alta demanda de recursos y procesamiento en tiempo real, como podría ser en el caso de un videojuego.

En el Cuadro 2.1 se resumen las principales tecnologías utilizadas para el desarrollo de apps móviles junto con las características más valoradas

**Capacitor**

Las aplicaciones en Ionic se ejecutan dentro de un web view. Esto significa que la aplicación no es más que un navegador embebido con contenido web y como tal, no puede acceder a los recursos nativos del dispositivo. Tan pronto como necesitemos utilizar la cámara de fotos, acceder a los ficheros del sistema o cualquier tipo de funcionalidad por

parte del host, necesitaremos una tecnología de bridge o puente. En los primeros años de Ionic esta tecnología era Cordova, propiedad de Apache. Sin embargo, Capacitor se está imponiendo como digno sucesor.

- Capacitor está creado por el mismo equipo que Ionic. Esto significa que Ionic ahora controla más espectro del E2E (de extremo a extremo). Ha sido diseñado y desarrollado pensando únicamente en Ionic, haciendo hincapié en su otpimización mediante una mejor gestión de los recursos nativos. Esto significa que el rendimiento de una aplicación con Capacitor es ligeramente mayor que con Cordova.

- Capacitor está bien mantenido y tiene una comunidad grande y muy activa, a diferencia de Cordova que nunca solucionó algunos bugs.

- Soporte para PWA. Capacitor fue concebido con la idea de dar soporte desde el principio a las PWA (Progressive Web Applications). Está optimizado y diseñado para el desarrollo de este tipo de aplicaciones cada vez más demandadas.

- Cordova funciona como una capa de abstracción que genera y utiliza los ficheros de código nativo por el desarrollador. Cuando se requiere implementar o personalizar manualmente una funcionalidad nativa, con Cordova este proceso es más complicado. Capacitor, sin embargo, está preparado para que los desarrolladores del lenguaje nativo implementen interfaces o lógica de negocio y luego pueda ser expuesta cómodamente en forma de API para ser usada por los desarrolladores web.

- Un detalle importante, Cordova utiliza como navegador embebido para iOs el componente UIWebView, deprecado por Apple a principios del año 2020 en favor del nuevo WKWebView. Las aplicaciones creadas con Cordova son muy complicadas en la actualidad de subir a la AppleStore y requieren que los desarrolladores se adentren en el código nativo y realicen manualmente este cambio [1].

**Funcionamiento de un plugin de Capacitor**

Los plugins de Capacitor son utilizados para llamar a código nativo desde el webview, es decir, la aplicación Ionic. La lógica subyacente es muy simple y en cuestión de minutos es posible comprender y crear el plugin que se necesite si es que no lo ha hecho ya la comunidad.

Un plugin de Capacitor no es más que un trozo de código nativo que se dispara mediante una llamada desde Javascript. Capacitor se encarga de realizar toda la lógica subyacente e incluso del paso o devolución de datos y parámetros entre ambas tecnologías, por lo que el desarrollador no debe preocuparse de estos aspectos. En la práctica, un plugin de Capacitor es un directorio de un proyecto Node, ya que a diferencia de Cordova, Capacitor se instala como dependencia local de forma autocontenida. De forma muy resumida, los tres elementos más importantes son:

- Un fichero de definición plugin.xml que define una serie de metainformación como el nombre del plugin, su repositorio, tecnologías soportadas, etc...

- Un fichero Typescript que define una interfaz, que no es más que los métodos que el plugin ofrece, con sus parámetros de entrada y salida.

- Directorios /android /ios y /web que contienen el código nativo que se lanzará al llamar a las funciones definidas en la interfaz de Javascript. No es obligatorio que todos estos directorios existan, pudiendo crearse un plugin únicamente para una tecnología.

### 2.4.2. TypeScript

Hoy en día es imposible cuestionar la potencia y utilidad de Javascript. Sin embargo, este lenguaje que comenzó casi como algo anecdótico, pese a haber evolucionado en una dirección muy correcta en la última década, sigue presentando importantes deficiencias que lo hacen altamente peligroso para un proyecto estable y escalable.

TypeScript pretende solucionar las deficiencias de Javascript. El tipado y la compilación son en realidad poderosos aliados a la hora de realizar un proyecto robusto y escalable en el tiempo [6].

- La seguridad de tipos evita toda una serie de errores y problemas que son un famoso quebradero de cabeza en Javascript. Si el código está fuertemente tipado y nuestro diseño e implementaciones son correctas, es muy difícil que nos equivoquemos enviando o recibiendo valores erróneos. Por ejemplo, la signatura de una función que recibe una clase de nuestro modelo es mucho más segura que una que reciba un tipo primitivo. El compilador simplemente nos impedirá continuar ahorrándonos un error que podría ser muy difícil de detectar en tiempo de compilación.

- El tipado y la declaración de clases e interfaces promueven una utilidad muy potente que muchos programadores ignoran: el IDE escribe el código por nosotros. Si el código está correctamente diseñado y la jerarquía de clases, interfaces y signaturas de los métodos es correcta, el código prácticamente es escrito a base de autocompletado. Las refactorizaciones son también por lo tanto muy sencillas de aplicar. Esto ayuda no solo a incrementar la velocidad de desarrollo sino que evita escribir código erróneo.

#### Angular

Angular es un conocido framework de front-end que permite crear SPAs reactivas. Angular posee una arquitectura MVVM (Modelo Vista - Vista Modelo), donde parte de la programación que antiguamente recaía en el servidor ahora se encuentra del lado del cliente [14]. Su paradigma declarativo y reactivo aporta una mayor velocidad de desarrollo, logrando de forma casi automática comportamientos que antes requerían un gran número de líneas de código. De igual forma, su filosofía de componentes autosuficientes y reutilizables no solo aporta mayor velocidad de desarrollo, sino que con su propio motor de renderizado logra interfaces con la ilusión de un tiempo de respuesta instantáneo. En realidad, más allá de las ventajas que puede ofrecer Angular, su elección no es voluntaria, ya que este framework es el que implementa Ionic y por tanto el que se usa para desarrollar aplicaciones con esta tecnología. Es cierto que en las últimas versiones de Ionic es posible utilizar Vue y React. No obstante, su uso aún no está demasiado extendido y las comunidades son pequeñas. Angular, por el contrario, lleva muchos años siendo usado, la comunidad es grande, experimentada y activa y la mayoría de los problemas técnicos que puedan surgir del desarrollo ya han sido cuestionados o resueltos, así como posibles bugs conocidos. Si bien la combinación de Ionic con Vue o React resulta muy

prometedora, a día de hoy se encuentra en un estado demasiado inexplorado que podría atascar o comprometer un proyecto donde el tiempo es limitado.

### VueJS

Esta tecnología se ha utilizado para elaborar un sencillo panel de control del back-end, que permite hacer un CRUD con la base de datos a travéz de una interfaz web. Al igual que Angular, Vue es un framework de front-end de tipo MVVM. A diferencia de Angular, Vue es un framework progresivo, lo que nos permite introducirlo únicamente en aquellas partes o para aquellas funcionalidades que necesitemos. Vue también es mucho más ligero y rápido, y su curva de aprendizaje es significativamente menor. Tanto React como Vue (que no Angular) ofrecen un rendimiento comparable en los casos de uso más comunes, con Vue normalmente un poco por delante debido a su implementación más ligera del DOM virtual.

En Vue, las dependencias de un componente se rastrean automáticamente durante su renderizado, por lo que el sistema sabe con precisión qué componentes deben volver a renderizarse cuando cambia el estado [5].

Vue permite un desarrollo muy ágil y ligero, movito por el que se ha utilizado para elaborar un veloz prototipo muy ágil de un panel de control.

# Capítulo 3

# Diseño del sistema

Una vez analizada la oferta de tecnologías disponible, se procede a realizar el diseño del sistema. En realidad, estos dos pasos se retroalimentan entre sí, y la decisión en uno de ellos puede afectar al otro y viceversa. En este capítulo se hará un diseño del sistema sin entrar aún en su implementación. Se diseñará una arquitectura del sistema y de software tanto a nivel de cliente como de servidor, así como del modelo de datos necesario. En definitiva, se trata de realizar un papel de analista y trasladar el problema a un sistema informático. Esta fase de análisis es importante porque sentará las bases del proyecto, le permitirá escalar o definirá la facilidad o el coste con que se enfrentará a los cambios, entre otras características.

## 3.1. Arquitectura del sistema

Una de las primeras decisiones a las que hubo que hacer frente fue si optar por una arquitectura standalone o local, o bien una arquitectura con un enfoque cliente-servidor, en cualquiera de sus posibles formas de implementación. En las siguientes páginas se analizan los pros y contras de cada una y qué motivó la decisión adoptada.

#### Local frente a cliente-servidor

Una de las primeras decisiones que se planteó de cara a la arquitectura del proyecto fue si debería tratarse de una aplicación local o bien se necesitaría algún tipo de conexión en la red. Para ello, se analizaron los pros y contras de cada opción

- Una aplicación puramente local es mucho más fácil y sencilla de desarrollar. La velocidad de respuesta del dispositivo es mayor al estar los datos almacenados localmente y no depender de la conexión de red. No hay que preocuparse por variables que surgen cuando se habla de este tipo de arquitecturas como sincronización, optimizar tiempos de espera o la seguridad. De cara a los costes, tampoco es necesario mantener una infraestructura de servicios que respalde al cliente.
- Una aplicación cliente-servidor es más compleja no solamente porque se necesite hacer dos desarrollos y sea necesario diseñar una arquitectura, sino porque se introduce una serie de problemas propios de este tipo de este tipo de sistemas, tales como errores o latencias en la conexión, seguridad, sincronización. Muchos de estos problemas más que una solución, suelen requerir una estrategia que minimice

sus efectos. Como ventajas, este tipo de arquitectura ofrece más dinamismo. Entre otras muchas posibilidades, pueden actualizarse o añadirse alimentos en tiempo real, funciones sociales o de gamificación, recopilar datos y crear una base de conocimiento compartido que podría servir, por ejemplo, para recomendar alimentos u ofrecer fuentes de datos que harían la aplicación demasiado pesada si estuviesen almacenados localmente. En definitiva, este tipo de diseño ofrece más juego y potencial que la versión local.

Dado que ambas opciones poseen puntos a favor y en contra, se ha optado por una solución híbrida que intente combinar lo mejor de ambos mundos: La aplicación poseerá una arquitectura cliente-servidor que le permitirá tener un mayor potencial y ofrecer las funciones mencionadas anteriormente. Sin embargo, también contará con mecanismos de sincronización y de funcionamiento en modo degradado. Los datos personales del usuario se almacenarán localmente, siendo casi innecesarias políticas de seguridad y protección en la red.

### Monolito frente a Microservicios

Una arquitectura de microservicios es un enfoque para desarrollar una aplicación software como una serie de pequeños servicios, cada uno ejecutándose de forma autónoma y comunicándose entre sí. Con los microservicios, las aplicaciones se dividen en sus elementos más pequeños e independientes entre sí. A diferencia del enfoque tradicional y monolítico de las aplicaciones, en el que todo se compila en una sola pieza, los microservicios son elementos independientes que funcionan en conjunto para llevar a cabo las mismas tareas.

Sin embargo, no todo son ventajas: uno de los problemas más complicados de manejar en los microservicios es la consistencia de la persistencia de datos. Esto es conocido coloquialmente como el lado oscuro de los microservicios. Para solucionar este problema existe el denominado patrón SAGA, aunque su implementación no es trivial.

Aunque se separen las bases de datos en trozos de negocio independientes, normalmente es frecuente que un servicio requiera cierta información de otro. En la Figura 3.1 de ejemplo, se muestra una arquitectura de microservicios simple que podría representar una plataforma de vídeo online, la cual dispone de un servicio de usuarios y otro de vídeo, donde este último está sufriendo una sobrecarga. Uno de los datos que devuelve el servicio de usuarios (videos_created) requiere de la consulta al servicio de vídeo. Esto se traduce en que el intento de separar ambos servicios para balancear la carga ha sido en vano y el comportamiento será igual que el de un monolito.

La solución a este problema reside en dar la vuelta al enfoque clásico e introducir un sistema de colas como podría ser Kafka. Al añadir un vídeo al sistema, se publica un evento al cual el servicio de usuarios está suscrito, incrementando en una unidad el contador del campo videos_created. Esto puede añadir diversos retos de implementación y de consistencia eventual.

Este ejemplo solo pretende ilustrar que los microservicios no son la solución a todos los problemas. A veces, en la ingeniería informática y el mundo del desarrollo se persigue utilizar la última tecnología de moda sin hacer un análisis de las necesidades reales del proyecto. Un diseño monolítoco o microlítico no es en absoluto obsoleto y posee muchas ventajas, como una gran velocidad de comunicaciones, facilidad de implementación, testeo, despliegue y depuración, simplicidad del proyecto a nivel técnico e inexistencia de los problemas mencionados anteriormente.

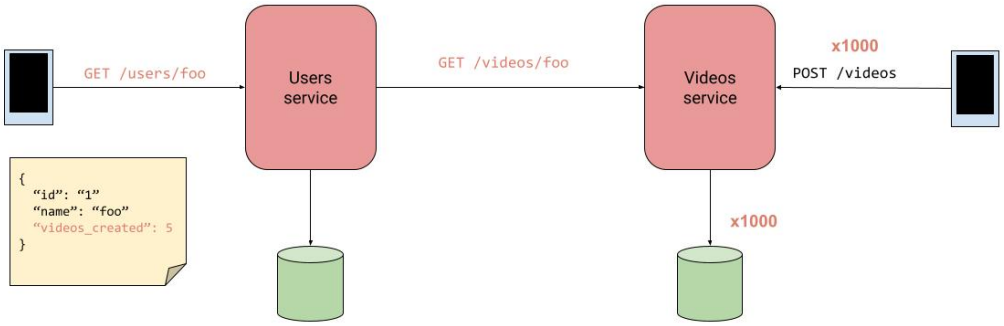


Figura 3.1: El servicio de vídeo sufre una sobrecarga y por tanto el servicio de usuarios también se resiente al depender de su respuesta.

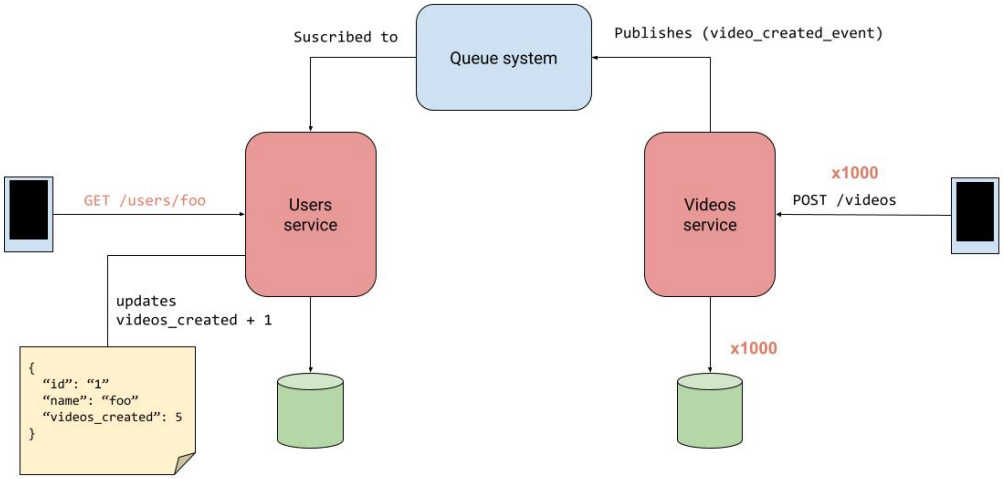


Figura 3.2: Esquema de microservicios del servidor (no se muestran todos)

Sintetizando y haciendo un balance de pros y contras:

- Debido a la reducción de los ciclos de desarrollo, una arquitectura de microservicios permite que la implementación y las actualizaciones se realicen más rápidamente.
- A medida que crece la demanda de ciertos servicios, es posible realizar implementaciones en distintos servidores e infraestructuras para satisfacer sus necesidades.
- Si estos servicios independientes están bien diseñados, no tienen que afectar a los demás. Esto significa que si una parte falla, no afecta a toda la aplicación, a diferencia del modelo de aplicaciones monolíticas.

- Debido a que las aplicaciones basadas en microservicios son más modulares y más pequeñas que las aplicaciones monolíticas tradicionales, ya no es necesario preocuparse por su implementación. Se requiere más coordinación, pero las ventajas son enormes.
- Debido al uso de API políglotas, los desarrolladores tienen la libertad de elegir los mejores lenguajes y tecnologías para la función que se necesita.

Desventajas y desafíos:

- Se debe invertir tiempo en identificar las dependencias entre los servicios. Y debe estar atento, porque cuando se termina un diseño, puede surgir la necesidad de muchos otros debido a esas dependencias.
- Las pruebas de integración, así como las pruebas finales, pueden tornarse más complejas e importantes que nunca.
- Cuando se actualicen los sistemas a las nuevas versiones, se debe tener en cuenta que se corre el riesgo de anular la compatibilidad con las versiones anteriores. Se puede diseñar en función de una lógica condicional para manejar este problema, pero se torna una tarea engorrosa. Otra opción es implementar múltiples versiones en vivo para distintos clientes, pero esto puede ser más complejo durante el mantenimiento y la gestión.
- La depuración remota se hace complicada y no funcionará cuando tenemos decenas o cientos de servicios. Desgraciadamente, no hay una única respuesta sobre cómo realizar la depuración en este momento.

Para realizar este proyecto, después de un análisis se ha optado sin embargo por un enfoque basado en microservicios debido a las siguientes razones:

- Al guardar las preferencias del usuario en el dispositivo local, las bases de datos son solo repositorios de consulta y no necesitan transaccionalidad, sorteando uno de los grandes problemas de los microservicios. En el caso de que el proyecto escalase y fuese necesaria más adelante, podría desarrollarse y acoplarse cuando estuviese lista.
- Se busca un desarrollo ágil e iterativo, implementado y experimentando con tecnologías y funcionalidades cada semana. Con los microservicios es posible realizar esto muy fácilmente sin tocar el resto del sistema y la evolución es sencilla por naturaleza.
- En un entorno real, la aplicación, por su naturaleza, presentaría picos de carga momentáneos y por el contrario habría otras franjas de tiempo donde el tráfico sería mucho menor. Los microservicios de cara a coste en la nube permitirían desplegar y mantener justo los recursos que se necesitan en cada momento.
- Por último y no menos importante, se busca descubrir y comprender una nueva tecnología emergente muy relacionada con la filosofía de la presente titulación.

## 3.2. Diseño del back-end

El back-end se ha diseñado siguiendo una arquitectura basada en microservicios. Como suele ocurrir en este tipo de diseños, la puerta de entrada es un balanceador de carga que distribuye las peticiones a los diferentes nodos tal y como se muestra en la imagen. Este esquema permite acoplar funcionalidades con mucha facilidad sin alterar el resto del sistema o incluso sin que este sea detenido. A continuación se describen los servicios:

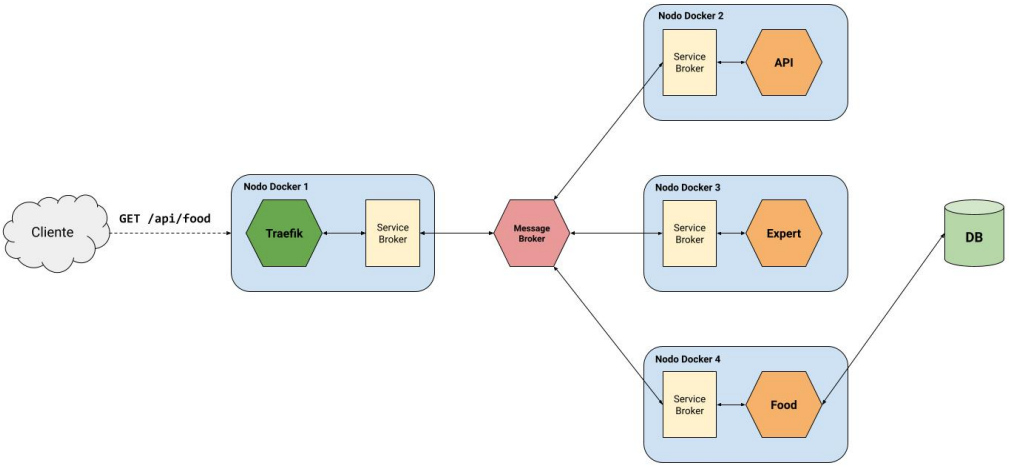


Figura 3.3: Esquema de microservicios del servidor (no se muestran todos)

### Expert

Este servicio es el encargado de determinar el nivel FODMAP de un alimento dada su composición. Determinará el nivel del semáforo y qué compuestos poseen niveles FODMAP elevados.

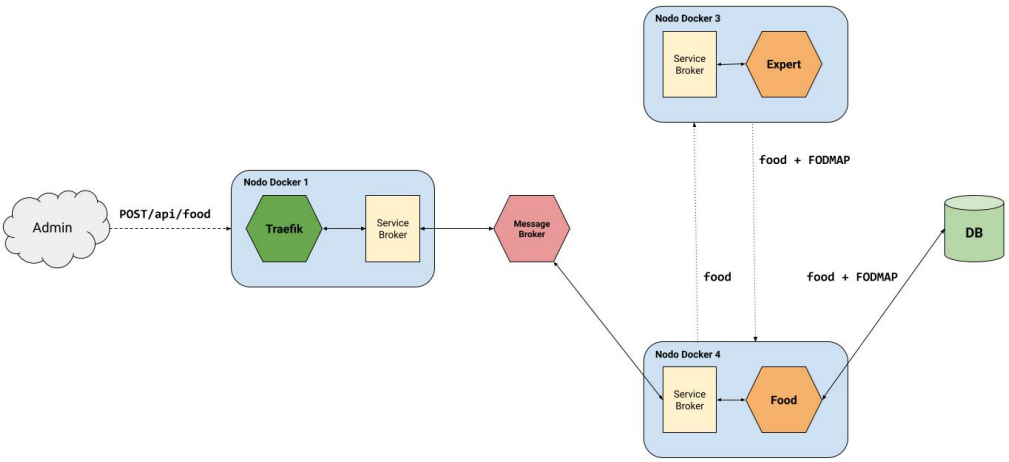


Figura 3.4: Al insertar un alimento, este se envía primero al microservicio Expert, encargado de calcular sus niveles fodmap. Una vez el alimento regresa de vuelta con toda la información completa, el servicio de alimentos lo almacena en la base de datos.

**API**

Es el servicio de API gateway encargado de administrar la API y en definitiva toda la entrada y comunicación externa con el back-end. Autoriza, autentica y expone rutas y servicios. Lo proporciona el propio framework.

**OpenFoodFacts**

Se encarga de devolver la información de OpenFoodFacts relativa a un alimento si se proporciona su código de barras. Esta información indica si el alimento es ultraprocesado o contiene sustancias alérgenas o poco saludables.

**Sync**

Permite conocer la metainformación de la base de datos del repositorio de alimentos. Es utilizado por el servicio de sincronización del cliente para conocer la versión actual de la base de datos y sincronizarse si es necesario.

**Food**

Es el servicio que permite obtener o manipular la información relativa a los alimentos. En definitiva, hacer un CRUD con el repositorio de alimentos. Nótese que los clientes solo tienen permiso para consultar y descargar los alimentos, no para modificar o añadir, lo cual queda del lado del servidor para el administrador responsable.

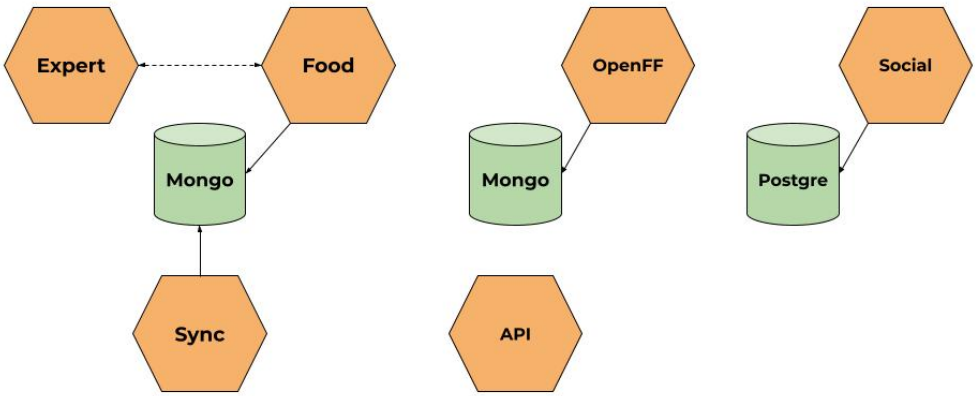


Figura 3.5: Resumen de los microservicios simplificados, donde solo se muestra su interacción entre ellos y las bases de datos del sistema. El servicio Social finalmente no se implementó. En el Capítulo 5 se describe cuál era su cometido planificado.

### 3.2.1. Diseño del modelo de datos

De cara a la persistencia y manipulación de datos, el primer paso ha sido identificar las entidades que intervienen en la solución a desarrollar. Ya en una primera aproximación parecía intuirse que el modelo de datos podía ser débilmente relacionado, por lo que se

decidió explotar esta característica para aprovechar la agilidad de desarrollo de MongoDB, pero teniendo cuidado de realizar relaciones que pudiesen volver esta tecnología en contra.

### Food

Parece muy simple ver un alimento como un documento JSON que en principio no debería guardar relación alguna con ninguna otra entidad. Este modelo cuenta con la información del nombre del alimento y sus cantidades de compuestos fodmap. Además, cuenta a su vez con un objeto anidado llamado fodmap. Este objeto representa los niveles fodmap del alimento y sus campos se calculan en base a sus compuestos siguiendo unas reglas lógicas. Por ejemplo, si la lactosa supera la cantidad de "4000", el campo lactosa del objecto fodmap cambiará su valor a 1. Además, el campo level ofrece un recuento precalculado de cuántos campos de este objeto están fijados al valor 1. En definitiva, level representa el valor del semáforo FODMAP, siendo 0 equivalente a verde, 1 a amarillo y 2 o más a rojo.

```
food: {
    name: "pan blanco"
    lactose: 400,
    fos: 0,
    gos: 0,
    manitol: 0,
    maltitol: 0,
    sorbitol: 0,
    xilitol: 0,
    fodmap: {
        oligo: 0,
        lactose: 0,
        polyols: 0,
        level: 0
    }
}
```

### Tolerance

El modelo para representar las tolerancias del usuario no es muy diferente y consiste simplemente en la lista de los compuestos fodmap y un valor que va de 0 a 10, indicando los valores más altos un nivel mayor de tolerancia a dicho compuesto. El hecho de que los diferentes objetos del modelo de datos compartan estructura y nombre de las claves hace que la implementación sea muy sencilla y limpia de realizar.

```
tolerance: {
    lactose: 0,
    fos: 0,
    gos: 0,
    manitol: 0,
    maltitol: 0,
    sorbitol: 0,
    xilitol: 0,
}
```

#### Plate

Un plato es una agrupación de alimentos que son tratados por el usuario como uno solo. Esto permite mejorar la experiencia de usuario al permitirle manipular conjuntos de ingredientes frecuentes como uno solo. Un plato está formado por un nombre y un conjunto de ingredientes. Estos ingredientes, a su vez son un objeto formado por el identificador del alimento y la cantidad del mismo. Ciñéndonos al estilo de diseño de MongoDB, el plato debería contener los alimentos por los que está formado y no simplemente una colección de los identificadores de estos alimentos. Esto recuerda a un modelo relacional y como se explicó en la introducción de este trabajo, no suele ser una buena práctica. Sin embargo, MongoDB sí admite que pueden existir pequeñas relaciones entre entidades, siempre que no sea la norma, y en este caso nos aporta un buen beneficio: si se ha creado un plato y el valor FODMAP de los ingredientes que lo conforman cambia, también lo harán automáticamente los valores del plato, ya que existe una única instancia para cada alimento en toda la aplicación y el resto son referencias a las mismas. Este patrón de diseño es útil en el caso que nos ocupa, ya que los alimentos cambiarán su valor FODMAP continuamente según se ingieran o se ajusten las tolerancias. En ese caso, sería necesario recorrer todos los platos alterando el valor de los objetos que contienen. Se trata, por tanto, de un caso acotado que deliberadamente se ha decidido implementar de esta forma al considerarse que los beneficios que aporta son mayores que los efectos colaterales.

```
plate: {
    name: "hamburguesa",
    ingredients: [
        {
            food_id: "1" //carne
            amount: 200
        },
        {
            food_id: "2" //pan
            amount: 300
        }
    ]
}
```

## 3.3. Diseño y arquitectura de la aplicación

En esta sección se describe el diseño de la aplicación a nivel de software, la organización del proyecto y sus componentes principales. Siguiendo los fundamentos de diseño y estilo planteados al inicio de esta memoria, se ha optado por una arquitectura fuertemente tipada y bien estructurada, organizada en capas, como se refleja en la Figura 3.6. Existe un único punto de entrada de datos en crudo a la aplicación, los cuales son inmediatamente convertidos a entidades del modelo. Este es el motivo por el que en la figura se ha separado el servicio de API del resto de servicios, ya que además de realizar sus funciones, implementa implícitamente una suerte de patrón factoría [10].

#### Sincronización

Para mejorar el tiempo de respuesta, ahorrar datos de red y ofrecer una mínima funcionalidad sin conexión, se ha diseñado un módulo de sincronización que solamente

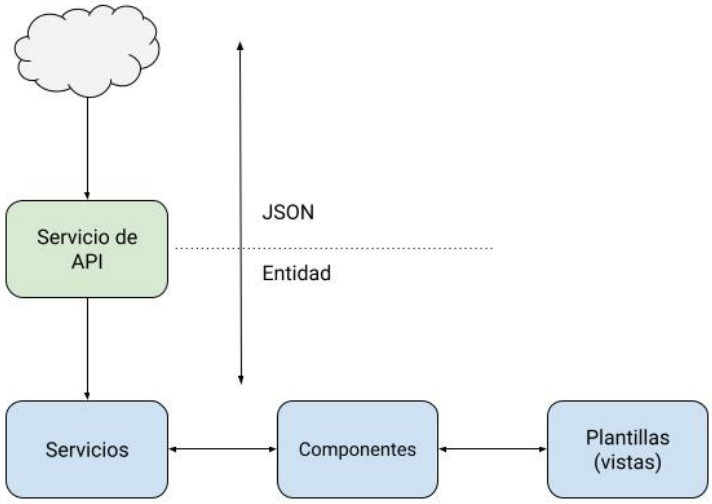


Figura 3.6: Capas básicas de la arquitectura de la aplicación.

descargará los alimentos si estos datos no están alineados con el repositorio. Una vez el repositorio de alimentos sea estable, no es un elemento que se actualice a diario. Podrían pasar incluso meses hasta que el equipo médico añadiese nuevos alimentos, por ello no es necesario que el dispositivo los descargue cada vez que inicia la aplicación. Así, el cliente almacenará los alimentos descargados localmente junto con un id de versión del repositorio. Cada vez que la aplicación inicia, el cliente solamente consultará si su versión coincide con la versión actual del repositorio remoto, evitando la descarga en caso afirmativo.

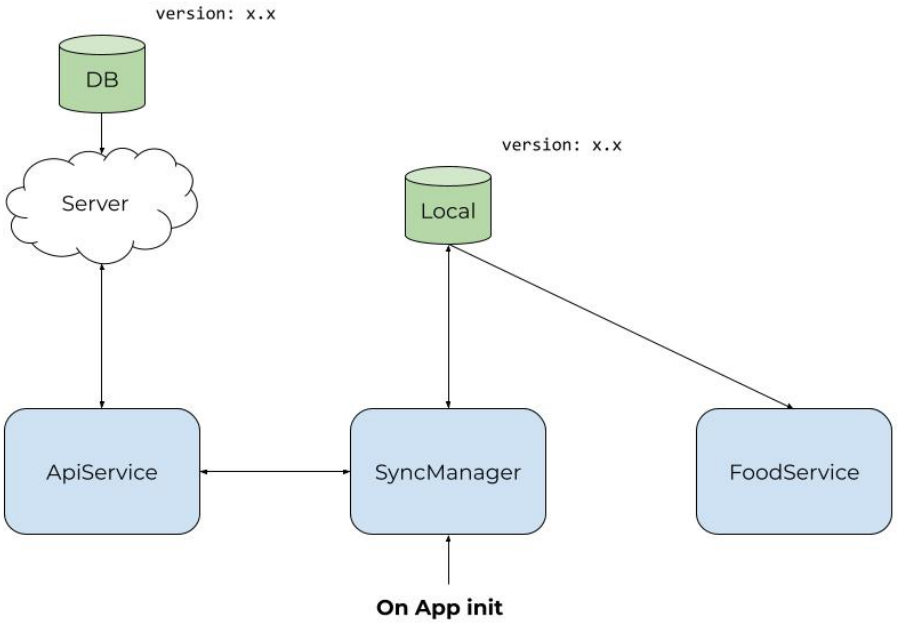


Figura 3.7: Diagrama simplificado del Syncronization Manager.

En la figura anterior se muestra la estrategia de sincronización diseñada. Nótese que estamos en el lado del cliente y la palabra servicio aquí hace referencia simplemente a un servicio de Ionic como clase o parte de código que realiza una función y no a microservicios del servidor. Como se observa, el servicio de alimentos, que es la clase encargada de proporcionar dichos alimentos como si de un DAO se tratase, solo sabe obtenerlos desde el almacenamiento local y es allí donde espera encontrarlos. Es la clase SyncManager quien vela porque ese almacenamiento local contenga datos. Para hacer esto, consulta al servidor la versión de la base de datos y descarga el repositorio de alimentos en caso necesario. Esta comunicación con el servidor se realiza a través de otra clase servicio diseñada para tal fin. En la siguiente figura se muestra un diagrama con mayor nivel de detalle del funcionamiento de esta sección del software.

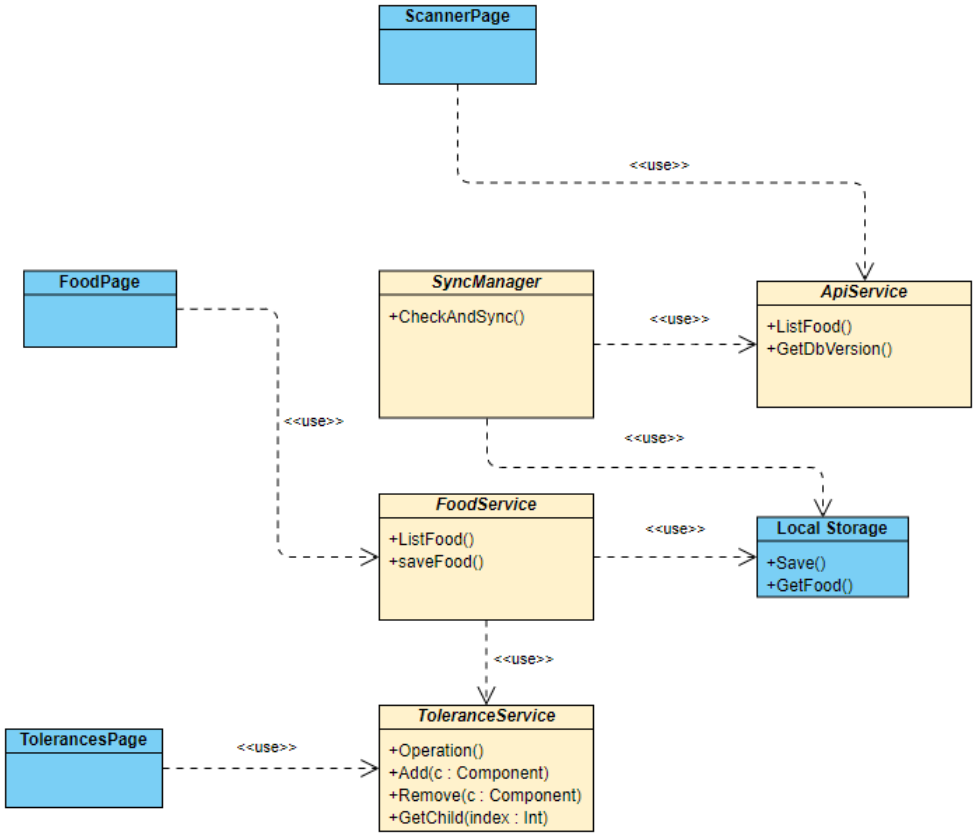


Figura 3.8: Diagrama UML del Syncronization Manager

### Estructura del proyecto

Un proyecto en Ionic/Angular puede tener varias estructuras de acuerdo a la metodología de desarrollo [2]. Para el caso de este proyecto se ha utilizado un diseño muy modular basado en casos de uso, que permita un fuerte desacoplamiento y un desarrollo iterativo sin afectar a las partes ya escritas.

### components

Aquí se encuentran componentes de Ionic. Los componentes son elementos visuales y con comportamiento propio que luego pueden ser reutilizados.

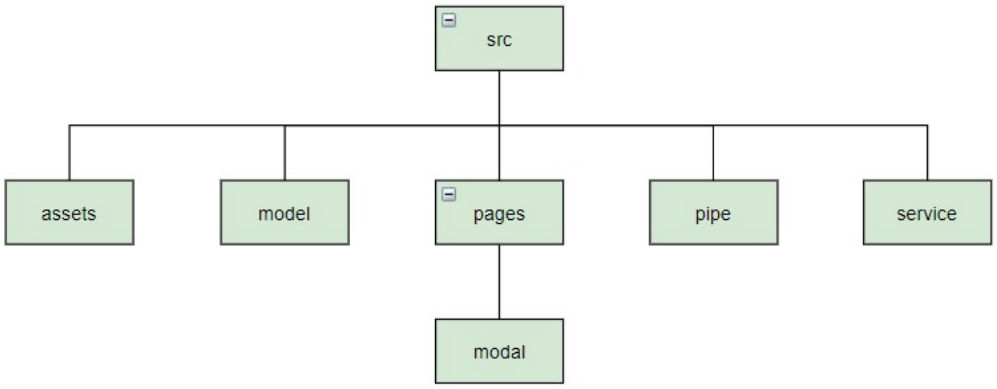


Figura 3.9: Estructura básica de un proyecto Ionic (Angular)

**model**

Almacena los modelos TypeScript de las diferentes clases propias diseñadas, como puede ser la clase alimento o la clase tolerancia. Estas clases vendrían a representar lo que se suele conocer como entidades.

**assets**

Contiene ficheros estáticos como variables de CSS globales, ficheros de internacionalización, grandes textos legales que quieran mostrarse en la aplicación, etc...

**service**

En este directorio se encuentran los servicios que se desarrollan. Un servicio, de forma muy simplificada puede verse como un trozo de código reutilizable en otras partes del código. Los servicios contienen lógica de negocio o código que es útil y frecuente en otra partes de la aplicación.

**pages**

Bajo esta ruta se encuentran las diferentes páginas y vistas. Algunos desarrolladores llaman a esta ruta Story porque cada subdirectorio contenido suele representar un caso de uso.

**pipes**

Contiene los pipes, funciones de Angular que son utilizadas para visualizar el código correctamente en las plantillas de forma limpia y eficiente.

### 3.3.1. Interfaz de usuario

Para la interfaz de usuario se ha buscado la facilidad de uso basándose en diferentes reglas de buenas prácticas de diseño y experiencia de usuario. Se ha optado por un diseño sidebar de material design que ofrece al usuario una rápida navegación y conocimiento del lugar de la aplicación en que se encuentra.

El primer paso ha sido diseñar una paleta de colores para la aplicación. Una vez elegida se procede a crear un tema que se aplicará de forma global. Esto no solo ahorra tiempo de desarrollo sino que le da a la aplicación una apariencia consistente y mejora la experiencia de usuario. Una vez pensados los tonos principales del tema, se puede utilizar una herramienta de generación de paletas que será de ayuda a la hora de elegir acertadamente el tono exacto del resto de colores.

La elección de los colores debería guardar relación con la temática de la aplicación. Una aplicación relacionada con la nutrición o la salud debe transmitir bienestar, sentimiento de naturaleza y otras sensaciones calmadas y positivas. Por ello, se ha optado por una paleta de tonos pastel donde los colores primarios serán tonos suaves de blanco, verde y azul como se aprecia en la Figura 3.10.


Figura 3.10: Propuesta de paleta de colores para la aplicación.

Para previsualizar el aspecto de la aplicación se utilizó una herramienta de elaboración de wireframes. El objetivo era reducir al máximo la complejidad de la interfaz y hacerla lo más simple y minimalista posible, minimizando la navegación o el número de clicks para llegar de un sitio a otro.

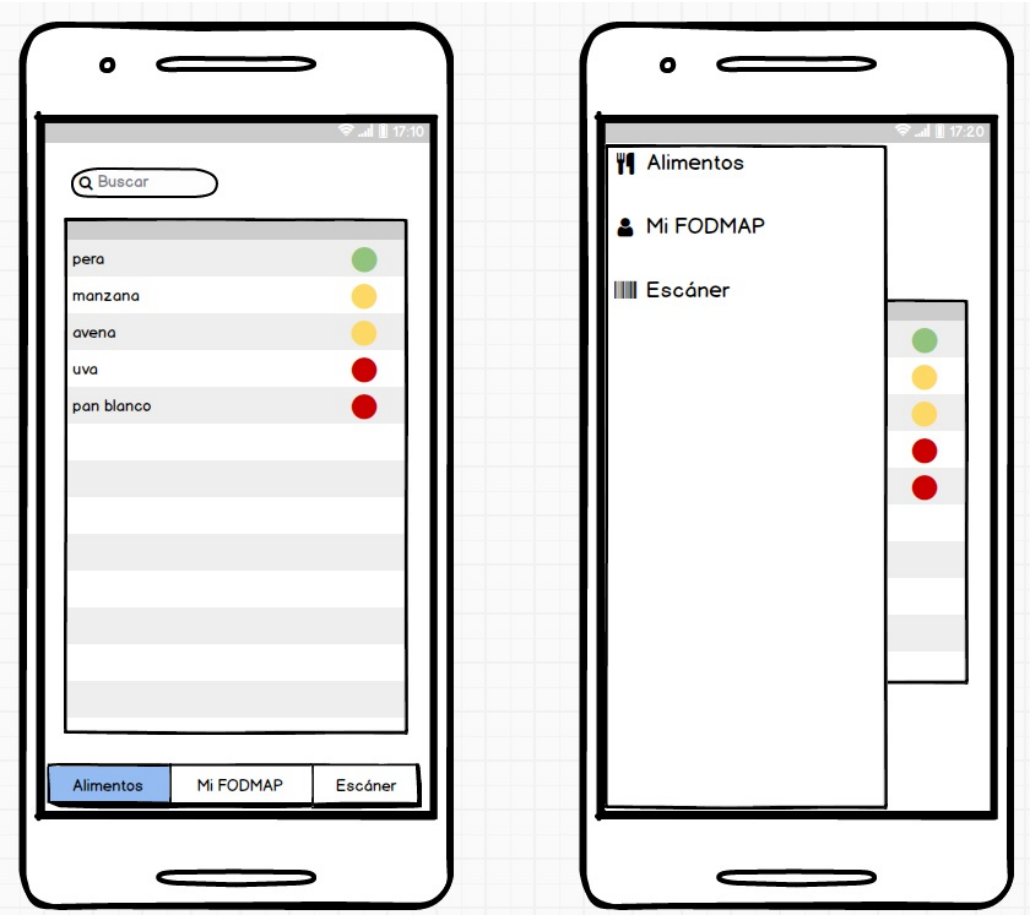


Figura 3.11: Mockup de la navegación entre pantallas. Se decidió prevenir y pasar de un menú basado en tabs (izquierda) a un típico menú lateral desplegable (derecha). El motivo es que se realizaría un desarrollo iterativo desconociendo de antemano el número exacto de pantallas y funcionalidades. El diseño basado en tabs limita de antemano el número máximo de pantallas, necesitando luego rehacer la aplicación si se excede dicho número.


Figura 3.12: Algunos mockups que se realizaron durante la fase de diseño.

# Capítulo 4

# Implementación

En este capítulo se describirá el sistema desde un menor nivel de abstracción y se mostrarán algunos puntos de interés. No se mostrará y explicará el código del proyecto en su totalidad sino únicamente aquellos fragmentos más destacables que permitan esbozar una fotografía general del sistema. En el anexo de esta memoria se adjunta el repositorio que contiene todo el código fuente del proyecto junto con el tag del último commit realizado en fecha válida para la entrega del presente trabajo.

## 4.1. Implementación del back-end

A la hora de implementar el back-end se ha utilizado el CLI de Moleculer para crear el proyecto. Esto ya proporciona los ficheros de configuración y estructura básica del proyecto, por lo que únicamente debemos preocuparnos de comenzar a desarrollar.

La primera toma de contacto consistió en desarrollar un microservicio que permitiese hacer un CRUD sencillo con la base de datos. Para ello, precisamente la primera cosa que debe ocurrir es conectar con dicha base de datos. Moleculer ofrece unos métodos predefinidos que representan el ciclo de vida de los servicios, lo cual resulta un lugar muy adecuado para esta primera tarea, como se aprecia en la Figura 4.1. En la sección methods, se encuentran todos los métodos relacionados con el ciclo de vida de un servicio de Moleculer. El nombre de los mismos es tan intuitivo que no requieren explicación. Dentro de esta sección también es posible escribir todos los métodos personalizados que se desee para utilizarlos en otro punto del código y mantener así una limpieza y estructura adecuada. Este es el caso del método propio connectToDatabase(), que se encarga de realizar dicha conexión y se invoca al crearse el servicio. Nótese que al utilizar un await en el primer paso del ciclo de vida del servicio, este no entrará en el método started() y por tanto no arrancará hasta que el primer paso se haya completado. La variable mongoose no es más que un objeto declarado al principio de la clase y a la que esta tendrá acceso desde todos sus métodos.

Una vez se logra conectar de manera exitosa con la base de datos, se puede pasar a implementar los primeros endpoints: listar todos los alimentos y obtener un alimento concreto. En Moleculer, los endpoints se encuentran dentro del objeto actions, y a su vez dentro de un objeto con un nombre personalizado, aunque es buena práctica que coincida con el verbo HTTP que se utiliza. Nótese que el path es relativo al nombre del servicio, es decir, al encontrarnos en el servicio food, el path “/“ ya se publicará como “/food“, y si se especificase como path “/bar“ el endpoint estaría disponible bajo “/food/bar“. Ahora tan solo queda escribir el código del handler, que es la función que se disparará cuando un


Figura 4.1: El método created es ideal para tareas de inicialización.

cliente llame al endpoint. El handler recibe opcionalmente un parámetro context. En el context, entre otras cosas, se almacenan todos los parámetros enviados por el cliente, vengan estos en la cabecera, la URL o el body de la petición. Lo único que es necesario hacer es utilizar el ORM de Mongoose para devolver la respuesta necesaria. En la Figura 4.2 se puede apreciar la simplicidad del código necesario para realizar estas tareas, y en la Figura 4.3 se detalla el modelo de Mongoose para un alimento.

El caso de añadir un alimento (Figura 4.4) es más interesante al mostrar como se interactúa con otros microservicios en Moleculer. Antes de explicar la implementación cabe recordar los pasos de este caso de uso: cuando se recibe un alimento, únicamente viene con sus ingredientes pero no con su valor FODMAP. Este valor FODMAP lo rellena el servidor en función de los ingredientes del alimento. En este caso, esta tarea la realiza otro microservicio. Como se puede observar, tras recibir el alimento, a través del contexto, se lanza un evento con un nombre personalizado y un dato que será el mensaje, en este caso el alimento. Otro microservicio (el que hace de sistema de reglas), se encarga de modificar el mensaje y devolverlo para que finalmente pueda ser insertado en la base de datos. Estas líneas de código aunque muy simples, merecen tres menciones:

- A pesar del uso de la palabra evento, nótese que esto no implica asincronía y el código se ejecuta de forma síncrona y ordenada.
- El mensaje actúa como parámetro de entrada y salida, por lo que no es necesario igualarlo al valor devuelto por la llamada.
- La arquitectura del sistema que subyace debajo es transparente para el programador. Si realizamos un despliegue monolítico o contamos con decenas de réplicas en

```
module.exports = {
    name: "food",

    settings: {},

    dependencies: [],

    actions: {

        list: {
            rest: {
                method: "GET",
                path: "/"
            },
            async handler() {
                return await Food.find();
            }
        },


        get: {
            rest: {
                method: "GET",
                path: "/:id"
            },
            async handler(ctx) {
                try {
                    return await Food.findOne({ _id: ctx.params.id });
                } catch(e) {
                    return e;
                }
            }
```

Figura 4.2: Método que devuelve los alimentos.

```
'use strict'

const mongoose = require('mongoose');


const FoodSchema = new mongoose.Schema({
    name: {type: String},
    lactose: {type: Number},
    fos: {type: Number},
    gos: {type: Number},
    manitol: {type: Number},
    maltitol: {type: Number},
    sorbitol: {type: Number},
    xilitol: {type: Number},
    fodmap: {
        polyols: {type: Number},
        lactose: {type: Number},
        oligo: {type: Number},
        level: {type: Number}
    }
});


module.exports = mongoose.model("Food", FoodSchema);
```

Figura 4.3: Modelo de Mongoose para un alimento.

contenedores balanceadas y en distintos lugares físicos es irrelevante a nivel de implementación.

Como se muestra en la Figura 4.6, las reglas resultaron ser muchísimo más simples de lo esperado, por lo que se descartaron otras aproximaciones más complicadas y se

```
44      post: {
45          rest: {
46              method: "POST",
47              path: "/"
48          },
49          async handler(ctx) {
50              const food = new Food(ctx.params)
51              ctx.emit("foodrepository.post", food);
52              await food.save();
53
54              const setting = await Setting.findOne();
55              setting.version = parseInt(setting.version) + 1;
56              await setting.save();
57
58              return food;
59          }
60      },
```

Figura 4.4: Método llamado al insertar un alimento. Véase como se actualiza la versión de la base de datos.

```
17      events: {
18          "foodrepository.post" (food) {
19              food.fodmap = fact;
20              return this.calculateFodmapLevel(food)
21          }
22      },
23
24
```

Figura 4.5: El servicio Expert, reacciona al evento del servicio de API y devuelve el alimento añadiendo la información relativa al valor FODMAP.

decidió realizar una rápida implementación para abordar cuanto antes el desarrollo del cliente.

El resto de servicios siguen exactamente la misma lógica y estructura, por lo que mencionarlos tan solo mostraría de forma repetida las mismas líneas de código una y otra vez. Sin embargo, hay un servicio que es más interesante y es generado por el propio Moleculer: el servicio de API. Como se aprecia en la Figura 4.7, este servicio dispone de multitud de configuraciones y es el punto de entrada al servidor. En este servicio se configura enrutamiento, listas de control de acceso, autenticación y autorización, ruta de contenido estático y muchas otras opciones, todo de una forma bastante inmediata e intuitiva.

Existe una regla de oro que siempre es necesario cumplir y dice que el servidor jamás debe creer lo que dice el cliente. Por supuesto, debería comprobarse y sanearse toda entrada de datos que llegue desde clientes externos, entre otras muchas medidas de seguridad. Sin embargo, como se mencionará al final de este trabajo, ni siquiera era una certeza que se terminase optando por el enfoque cliente-servidor hasta poseer mayores resultados, por lo que no se decidió implementar aún la capa de seguridad necesaria para poder avanzar en otros aspectos del proyecto. Más aún cuando debido al diseño modular de la arquitectura, esta sería muy fácil de añadir llegado el caso independientemente del estado del proyecto.

```
25      methods: {
26
27          calculateFodmapLevel(food) {
28              food.fodmap = fact;
29
30
31              if (food.gos >= 300 || food.fos >= 300) {
32                  food.fodmap.oligo = true;
33                  food.fodmap.level++;
34              }
35
36              if (food.lactose >= 4000) {
37                  food.fodmap.lactose = true;
38                  food.fodmap.level++;
39              }
40
41              if (food.manitol >= 300 || food.sorbitol >= 300) {
42                  food.fodmap.polyols = true;
43                  food.fodmap.level++;
44              }
45              return food;
46          }
47
```

Figura 4.6: Lógica para determinar el nivel FODMAP de un alimento.

```
12      name: "api",
13      mixins: [ApiGateway],
14
15      // More info about settings: https://moleculer.services/docs/0.14/moleculer-web.html
16      settings: {
17          // Exposed port
18          port: process.env.PORT || 3000,
19
20          // Exposed IP
21          ip: "0.0.0.0",
22
23          // Global Express middlewares. More info: https://moleculer.services/docs/0.14/moleculer-web.html#Middlewares
24          use: [],
25
26          cors: {
27              // Configures the Access-Control-Allow-Origin CORS header.
28              origin: "*",
29              // Configures the Access-Control-Allow-Methods CORS header.
30              methods: ["GET", "PATCH", "POST", "PUT", "DELETE"],
31          },
32
33          routes: [
34              {
35                  path: "/api",
36
37                  whitelist: [
38                      "**"
39                  ],
```

Figura 4.7: Aspecto general del microservicio de API creado automáticamente por Moleculer.

### Implementación del panel de control

Como pequeño valor añadido, para no manipular la base de datos directamente o a través de llamadas REST, se implementó un pequeño panel de control para hacer el CRUD. Esto permitiría que una persona sin conocimientos técnicos pudiese administrar la aplicación. El código de este front-end hecho en Vue no tiene nada mencionable y puede consultarse en el repositorio del proyecto. En la Figura 4.8 se observa el panel y la acción de realizar botón derecho sobre un alimento. Este panel se desarrolló completamente aparte en Vue, y simplemente se compiló y se situó en el directorio de ficheros estáticos de Moleculer bajo la ruta raíz.

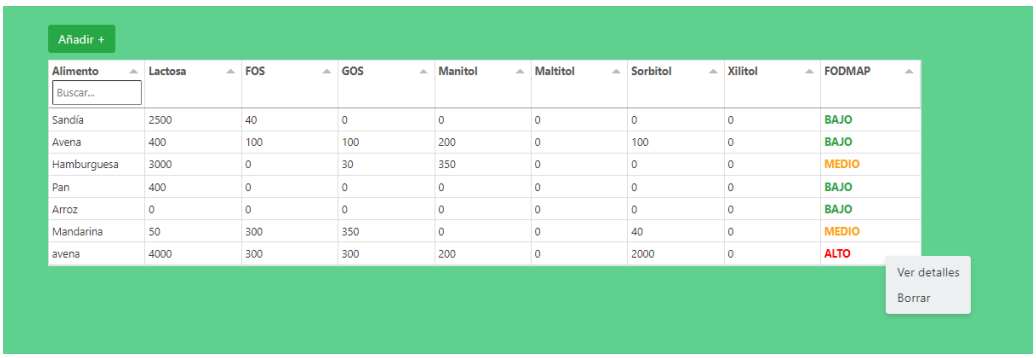


Figura 4.8: Panel de administración creado en VueJS.

### Despliegue monolítico

A pesar de utilizar microservicios a nivel de implementación, el despliegue en sí puede realizarse de forma monolítica en un único equipo. Esto es lo primero que se ha probado para asegurarse de que todo funciona correctamente. Moleculer integra un sistema de logs que hace que sea muy simple controlar todo lo que ocurre. En la Figura 4.9 se muestra el arranque del servidor en modo monolítico. Obsérvese como se registran todos los servicios necesarios.

```
[2021-06-02T21:32:02.091Z] INFO  desktop-1thcqpb-479584/API: Register route to '/admin'
[2021-06-02T21:32:02.091Z] INFO  desktop-1thcqpb-479584/API:   Registered 1 middlewares.
[2021-06-02T21:32:02.091Z] INFO  desktop-1thcqpb-479584/API:
[2021-06-02T21:32:02.590Z] INFO  desktop-1thcqpb-479584/REGISTRY: '$node' service is registered.
[2021-06-02T21:32:02.593Z] INFO  desktop-1thcqpb-479584/REGISTRY: 'expert' service is registered.
[2021-06-02T21:32:02.594Z] INFO  desktop-1thcqpb-479584/REGISTRY: 'openfoodfacts' service is registered.
[2021-06-02T21:32:02.595Z] INFO  desktop-1thcqpb-479584/$NODE: Service '$node' started.
[2021-06-02T21:32:02.595Z] INFO  desktop-1thcqpb-479584/EXPERT: Service 'expert' started.
[2021-06-02T21:32:02.596Z] INFO  desktop-1thcqpb-479584/REGISTRY: 'food' service is registered.
[2021-06-02T21:32:02.597Z] INFO  desktop-1thcqpb-479584/REGISTRY: 'sync' service is registered.
[2021-06-02T21:32:02.598Z] INFO  desktop-1thcqpb-479584/OPENFOODFACTS: Service 'openfoodfacts' started.
[2021-06-02T21:32:02.599Z] INFO  desktop-1thcqpb-479584/FOOD: Service 'food' started.
[2021-06-02T21:32:02.600Z] INFO  desktop-1thcqpb-479584/SYNC: Service 'sync' started.
[2021-06-02T21:32:02.601Z] INFO  desktop-1thcqpb-479584/METRICS: Prometheus metric reporter listening on http://0.0.0.0:3030/metrics address.
[2021-06-02T21:32:02.603Z] INFO  desktop-1thcqpb-479584/API: API Gateway listening on http://localhost:3000
[2021-06-02T21:32:02.605Z] INFO  desktop-1thcqpb-479584/REGISTRY: 'api' service is registered.
[2021-06-02T21:32:02.607Z] INFO  desktop-1thcqpb-479584/API: Service 'api' started.
[2021-06-02T21:32:02.616Z] INFO  desktop-1thcqpb-479584/BROKER: ✓ ServiceBroker with 6 service(s) is started successfully in 30ms.
mol $ [2021-06-02T21:32:03.309Z] INFO  desktop-1thcqpb-479584/API: ♻ Generate aliases for '/api' route...
[2021-06-02T21:32:03.311Z] INFO  desktop-1thcqpb-479584/API:      GET /api/openfoodfacts/:id => openfoodfacts.get
[2021-06-02T21:32:03.312Z] INFO  desktop-1thcqpb-479584/API:      GET /api/food => food.list
[2021-06-02T21:32:03.312Z] INFO  desktop-1thcqpb-479584/API:      GET /api/food/:id => food.get
[2021-06-02T21:32:03.313Z] INFO  desktop-1thcqpb-479584/API:     POST /api/food => food.post
[2021-06-02T21:32:03.314Z] INFO  desktop-1thcqpb-479584/API:    PATCH /api/food/:id => food.patch
[2021-06-02T21:32:03.315Z] INFO  desktop-1thcqpb-479584/API:   DELETE /api/food/:id => food.delete
[2021-06-02T21:32:03.316Z] INFO  desktop-1thcqpb-479584/API:      GET /api/sync => sync.get
[2021-06-02T21:32:03.317Z] INFO  desktop-1thcqpb-479584/API:      GET /api/api/list-aliases => api.listAliases
```

Figura 4.9: Traza del servidor arrancando.

Con el servidor en marcha, es hora de realizar la primera prueba e intentar insertar un alimento en la base de datos con algunos valores de prueba. En la Figura 4.10 se muestra la llamada realizada junto con la respuesta y en la Figura 4.11 el log del servidor. El servidor rellena el alimento con los campos FODMAP adecuadamente y lo guarda en la base de datos. Además, tal como se pensó en la fase de diseño, el servidor aumenta la versión de la base de datos cada vez que añade o modifica un alimento.

### Despliegue contenerizado

Para realizar un despliegue distribuido y contenerizado, Moleculer ofrece un fichero de Docker-compose parcialmente relleno y con una plantilla que tan solo hay que copiar y rellenar adecuadamente con los servicios que se ha creado, como se observa en la Figura 4.12. De la misma forma también ofrece un fichero para desplegar con K8s, aunque en este caso se ha optado por la primera tecnología. Una vez rellenado este fichero, Moleculer ofrece un script de npm que realiza todo el trabajo por nosotros. Tras

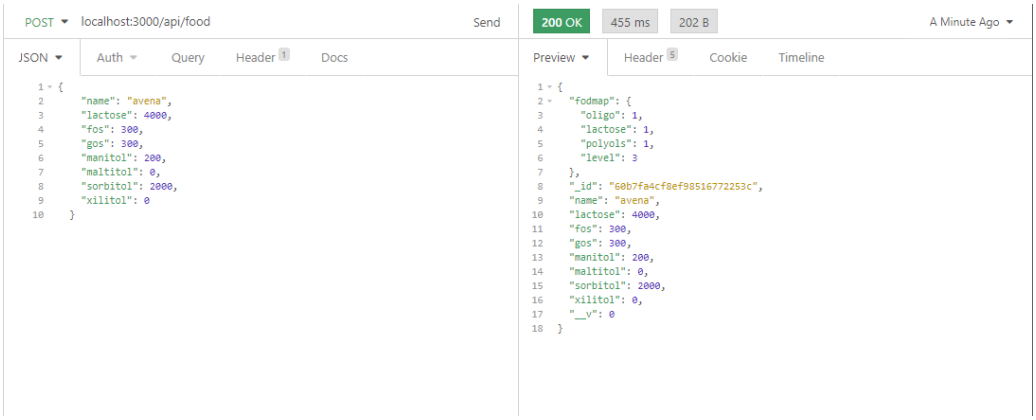


Figura 4.10: Envío y respuesta de la petición para insertar un alimento.

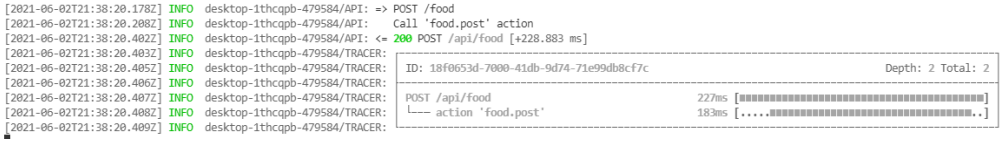


Figura 4.11: Logs del servidor al recibir la petición de insertar un alimento

ejecutar dicho comando, Moleculer creará el clúster de contenedores que luego podrá ser desplegado cuando se desee, como se observa en la Figura 4.13

```yaml
version: "3.3"

services:

  api:
    build:
      context: .
    image: micro
    env_file: docker-compose.env
    environment:
      SERVICES: api
      PORT: 3000
    depends_on:
      - nats
    labels:
      - "traefik.enable=true"
      - "traefik.http.routers.api-gw.rule=PathPrefix(`/`)"
      - "traefik.http.services.api-gw.loadbalancer.server.port=3000"
    networks:
      - internal

  expert:
    build:
      context: .
    image: micro
    container_name: expert
    env_file: docker-compose.env
    environment:
      SERVICES: expert
    depends_on:
      - nats
    networks:
      - internal

  food:
    build:
      context: .
    image: micro
    container_name: food
    env_file: docker-compose.env
    environment:
      SERVICES: food
    depends_on:
      - nats
    networks:
      - internal
```

Figura 4.12: Fichero de Docker Compose.

Si se accede al panel de control en el navegador que proporciona Moleculer, puede verse los detalles y características del cluster creado, como se muestra en las Figura 4.14, la Figura 4.15 y la Figura 4.16


Figura 4.13: Despliegue con Docker.

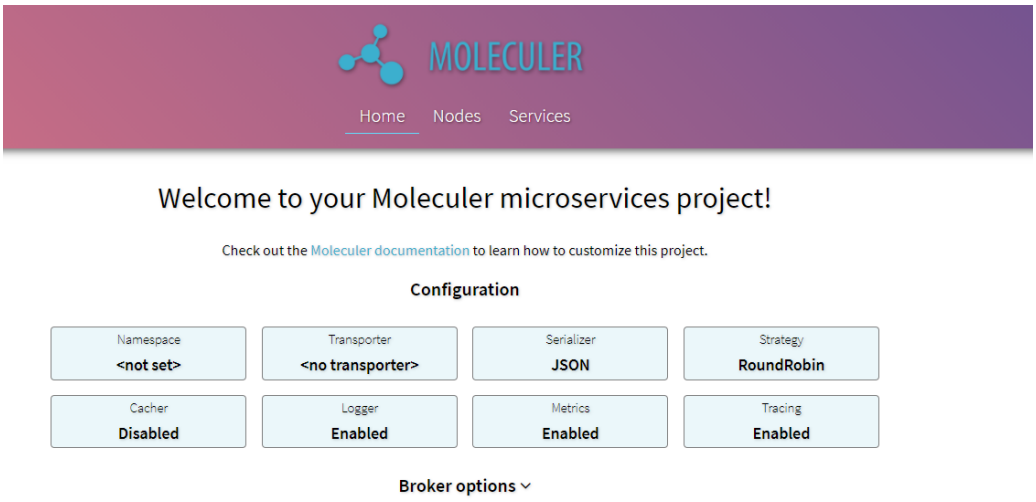


Figura 4.14: Panel de administración de Moleculer: vista de las configuraciones.

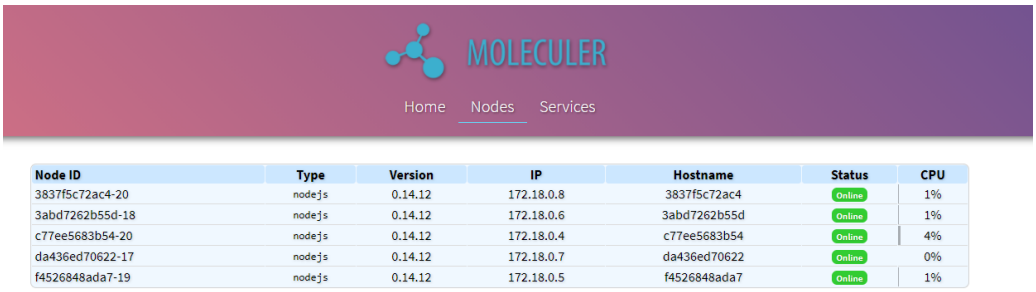


Figura 4.15: Panel de administración de Moleculer: vista de los nodos.

## 4.2. Implementación del front-end

Una vez se tiene los servicios de back-end preparados, se puede comenzar a desarrollar el cliente. El código del cliente es relativamente extenso y mucha parte del mismo son

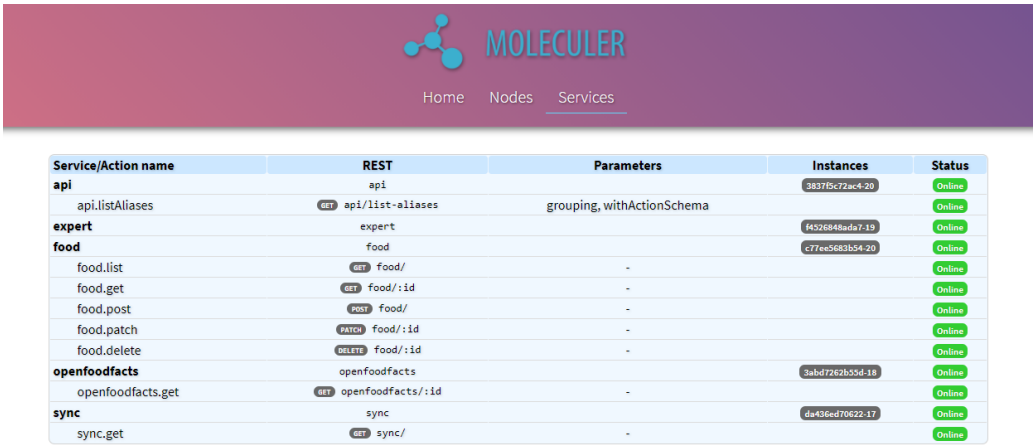


Figura 4.16: Panel de administración de Moleculer: vista de los servicios.

características técnicas de Ionic o Angular, por lo que en esta sección se hará mención únicamente a las partes más destacables. A continuación se describirá con algo más de detalle la pantalla de entrada a la aplicación, que es la de listar los alimentos. Esta pantalla hace uso de servicios, modales y el sistema de sincronización, por lo que es un excelente ejemplo para mostrar el aspecto general que se repite a lo largo de todo el código de la aplicación, por lo que para el resto de pantallas y funcionalidades no será necesario sino mencionar los puntos más destacables.

Cabe mencionar que ionic posee un CLI que ayuda en el desarrollo, por lo que gran parte del código es creado por esta herramienta para acelerar la velocidad de implementación y reducir los errores. Por ejemplo, para crear esta primera página food dentro del directorio pages basta con ejecutar:

```
ionic g page pages/food
```

Esto creará todos los ficheros y componentes necesarios para una página y también la añadirá al routing de la aplicación. De forma parecida es posible crear componentes y servicios. El elemento de Ionic que está compuesto por más ficheros es la página. Los servicios y componentes no poseen todos los ficheros que componen la primera, por lo que es el mejor ejemplo para dar una breve explicación de cómo se distribuye la lógica en estos componentes. Atendiendo a la Figura 4.17, se cuenta con los siguientes ficheros:

- un fichero de routing donde se especifica la ruta de la página y se importa el componente de routing. El CLI de Ionic lo crea y rellena automáticamente por nosotros, por lo que no es significante.
- un fichero de módulo donde se importan y agrupan los diferentes componentes externos utilizados. Los módulos de Angular permiten agrupar un conjunto de código utilizado en un ámbito concreto de la aplicación, o una funcionalidad específica. El fichero de módulos aunque no es necesario, permite el lazy loading si desea utilizarse, cargando los componentes importados bajo demanda.
- Un fichero .page.html donde se escribe el código de la vista.
- Un fichero .page.ts donde se escribe el código Typescript que usa la vista.
- Un fichero .page.scss donde se escribe el código CSS (en Sass).

- Un fichero .spec.ts para tests unitarios.

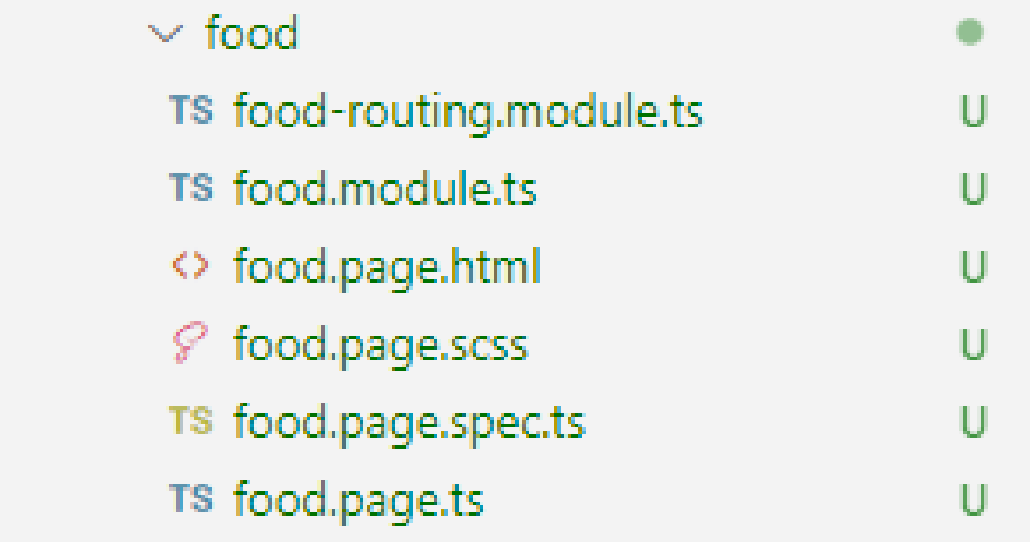


Figura 4.17: Ficheros que componen un componente página.

La combinación de los componentes prefabricados que ofrece Ionic junto con las ventajas de Angular, hace que escribir la vista sea trivial, como se aprecia en la Figura 4.18. Tan solo basta con crear una lista que contendrá ítems que serán los alimentos descargados por la API. En función del nivel FODMAP, se añade un icono de color que representa el semáforo.

```
<ion-header>
  <ion-toolbar>
    <ion-buttons slot="start">
      <ion-menu-button></ion-menu-button>
    </ion-buttons>
    <ion-title>Alimentos</ion-title>
  </ion-toolbar>
</ion-header>

<ion-content>

  <ion-list>
    <ion-item *ngFor="let result of results" >
      <ion-label (click)="openModal(result)">{{result.name}}</ion-label>
      <ion-icon slot="end" name="create" (click)="addToDiary(result)" color="secondary"></ion-icon>
      <ion-icon *ngIf="result.fodmap.level <= 0" name="ellipse" class="good"></ion-icon>
      <ion-icon *ngIf="result.fodmap.level == 1" name="ellipse" class="mid"></ion-icon>
      <ion-icon *ngIf="result.fodmap.level >= 2" name="ellipse" class="danger"></ion-icon>
    </ion-item>
  </ion-list>

</ion-content>
```

Figura 4.18: Código de la vista principal que lista los alimentos.

El código Typescript resulta algo más interesante, aunque también es muy sencillo. En la Figura 4.19 se muestra la primera mitad del mismo y los primeros métodos. El constructor, como se observa, debería utilizarse únicamente para inyectar los servicios a utilizar. En el método ngOnInit(), que se dispara cuando el componente es creado, se hace una llamada al servicio de sincronización y se obtiene los alimentos. Para comprender estas líneas de código es necesario tener en cuenta un detalle: el código ngOnInit() solo se llama una vez cuando el componente es creado ya que este es cacheado. Es por ello que en realidad solo se llamará una vez al inicio de la aplicación al servicio de sincronización.

```typescript
export class FoodPage implements OnInit {

  results: Food[];

  constructor(private foodService: Foodservice,
              private syncManagerService: SyncmanagerService,
              private diaryService: DiaryService,
              private modalController: ModalController,
              private toastController: ToastController) { }


  async ngOnInit() {
    await this.syncManagerService.checkAndsync();
    await this.getFood();
  }

  async presentToast(foodName: string) {
    const toast = await this.toastController.create({
      message: `${foodName} añadido al diario`,
      duration: 2000
    });
    toast.present();
  }

  async getFood() {
    this.results = await this.foodService.listFood();
  }
```

Figura 4.19: Código Typescript de la vista para mostrar los alimentos.

Si se desea que un código se llame siempre que se entre en una vista, hay otro método del ciclo de vida de Ionic (que no de Angular) que es ionViewWillEnter().

Más adelante se analizará la implementación de los servicios utilizados. Por ahora, continuaremos analizando este componente. Se observa la función openModal, que abre una ventana con los detalles de un alimento al pinchar sobre él. Un modal no es más que otra página que puede recibir un parámetro de entrada. El código de este componente es prácticamente inexistente y no es digno de mención.

Por último, se observa el método addToDiary(), que se dispara cuando se elige ingerir un alimento y es el responsable de anotarlo en el diario. El método presentToast() muestra un pequeño aviso para dar feedback al usuario de que la acción se ha realizado.

```typescript
  async addToDiary() {
    this.diaryService.saveFood(this.food, this.amount);
    this.presentToast(this.food.name)
    this.modalController.dismiss();
  }


  async presentToast(foodName: string) {
    const toast = await this.toastController.create({
      message: `${foodName} añadido al diario`,
      duration: 2000
    });
    toast.present();
  }
```

Figura 4.20: Código Typescript de la vista para mostrar los alimentos.

Analicemos ahora con más detalle los servicios que son utilizados no solo en este componente sino en muchos otros a lo largo de la aplicación. Para ello, comenzaremos con el servicio de API, que es el punto de entrada de los datos al sistema. Como se observa en la Figura 4.21, este servicio realiza las llamadas HTTP a la API y mapea la respuesta a entidades de nuestro modelo de negocio. En el método getFoodFacts puede observarse que se ha comentado temporalmente la llamada al servidor propio para realizar llamadas

directas a la API pública de OpenFoodFacts. El motivo de esto ya se explicó en el 3. Como se mencionaba en la introducción de este trabajo, el servicio de API no solo sirve de punto de entrada de los datos a la aplicación, sino que además realiza la importantísima labor de pasar de datos en crudo a entidades del modelo utilizado.

```
11  export class ApiService {
12
13    baseUrl: string;
14
15    constructor(private httpClient : HttpClient) {
16      this.baseUrl = `http://192.168.0.108:3000/api`;
17    }
18
19
20    public listFood(): Observable<Food[]> {
21      return this.httpClient.get<Food[]>(`${this.baseUrl}/food`)
22        .pipe(map(foodList => {
23          return foodList.map(food => new Food(food));
24        }));
25    }
26
27
28    public getDbVersion(): Observable<Setting> {
29      return this.httpClient.get<Setting>(`${this.baseUrl}/sync`)
30        .pipe(map(setting => {
31          return new Setting(setting);
32        }));
33    }
34
35    public getFoodFacts(barcode: string): Observable<any> {
36      //return this.httpClient.get(`${this.baseUrl}/openfoodfacts/${barcode}`)
37      return this.httpClient.get(`https://world.openfoodfacts.org/api/v0/product/${barcode}`)
38    }
39
```

Figura 4.21: Código del servicio de API

El servicio de sincronización hace uso del servicio de API visto anteriormente y de otros que se explicarán más adelante. Como se puede observar en la Figura 4.22, se limita a cotejar si las versiones locales y en base de datos del repositorio de alimentos coinciden (o si existe el almacenamiento local). En caso de que esto no sea así, se descarga los alimentos y los almacena localmente junto con el nuevo número de versión.

```
11  export class SyncmanagerService {
12
13    constructor(private apiService: ApiService,
14                private foodService: Foodservice) { }
15
16
17    async checkAndsync() {
18      const setting = await this.apiService.getDbVersion().toPromise();
19      const localSetting = new Setting(JSON.parse((await Storage.get({ key: 'setting' })).value));
20
21      if (setting.version != localSetting.version) {
22        console.log('Sincronizando...')
23        await Storage.set({
24          key: 'setting',
25          value: JSON.stringify(setting)
26        });
27
28
29        return new Promise((res, rej) => {
30          this.apiService.listFood().subscribe(async foodList => {
31            this.foodService.saveFood(foodList);
32            res(null);
33          })
34        }).then(food => {return});
35      }
36    }
37  }
```

Figura 4.22: Código del servicio de sincronización.

Llegamos por fin, al servicio de alimentos, como se aprecia en la Figura 4.23, que es el utilizado originalmente en la vista de listar alimentos. Este servicio devuelve los alimentos cotejando su semáforo FODMAP con los niveles de tolerancia según las preferencias del usuario y los alimentos que figuran en su diario. Por ello, la hora de obtener los alimentos en una vista, este es el servicio que debería utilizarse y no el de API, que se encuentra en un nivel de abstracción menor. El código es muy simple y prácticamente no necesita explicación. Obsérvese como antes de devolver el alimento este se contrasta con el servicio de tolerancias.

```
14
15    async listFood(): Promise<Food[]> {
16      let foodList: Food[] = JSON.parse((await Storage.get({ key: 'food' })).value);
17      foodList.map(async food => {
18        return new Food(await this.toleranceService.getToleranceToFood(food));
19      });
20      return foodList;
21    }
22
23    async saveFood(foodList: Food[]) {
24      await Storage.set({
25        key: 'food',
26        value: JSON.stringify(foodList)
27      })
28    }
29
30    async getFood(id: string, amount: number = 100): Promise<Food> {
31      let foodList: Food[] = JSON.parse((await Storage.get({ key: 'food' })).value);
32      let food: Food = foodList.find(food => food._id == id);
33      return new Food(await this.toleranceService.getToleranceToFood(food, amount))
34    }
35  }
```

Figura 4.23: Código del servicio de alimentos.

El servicio de tolerancias, como se aprecia en la Figura 4.24, es el encargado de almacenar y cargar las tolerancias introducidas por el usuario, pero también de contrastar los ingredientes de los alimentos según sus ingredientes. Aquí no se muestra el código en su totalidad porque siempre tiene el mismo aspecto. Básicamente se trata de comparar si ciertos campos superan un umbral establecido para activar el campo FODMAP correspondiente en consecuencia.

```
21
22    async getToleranceToFood(food: Food, amount: number = 100): Promise<Food> {
23
24      let tolerance = new Tolerance(JSON.parse((await Storage.get({ key: 'tolerance' })).value));
25      let fodmapLevel: FodmapLevel = JSON.parse((await Storage.get({key: 'fodmapLevel'})).value)
26      if (fodmapLevel == null) {
27        fodmapLevel = new FodmapLevel();
28      }
29
30        let umbral = tolerance.lactose * 600 - fodmapLevel.lactose
31        if (food.lactose > 0) {
32          if (food.lactose * (amount / 100 )> umbral) {
33            if (food.fodmap.lactose == 0) {
34              food.fodmap.lactose = 1;
35              food.fodmap.level += 1;
36            }
37          } else {
38            if (food.fodmap.lactose == 1) {
39              food.fodmap.lactose = 0;
40              food.fodmap.level -= 1;
41            }
42          }
43      }
44
```

Figura 4.24: Código del servicio de tolerancias.

Existen algunos otros servicios como el de platos (Figura 4.25) que no merecen ninguna mención especial. Como se puede observar, una vez que se conoce la arquitectura y el

esquema de trabajo de la aplicación, el código resulta bastante simple y repetitivo conceptualmente.

```
18  async savePlate(plate: Plate) {
19
20    let plateList: Plate[] = JSON.parse((await Storage.get({ key: 'plates' })).value);
21
22    if (plateList != null) {
23      plateList.push(plate);
24    }
25    else {
26      plateList = [];
27      plateList.push(plate)
28    }
29
30    await Storage.set({
31      key: 'plates',
32      value: JSON.stringify(plateList)
33    })
34  }
```

Figura 4.25: Código del servicio de platos.

Con Ionic, todo es mucho más sencillo de lo que se puede imaginar. En la Figura 4.26 se muestra el código relativo a escanear un alimento y obtener su información. La llamada al servicio barcodeScanner ya se encarga de abrir la cámara del dispositivo y detectar un código de barras automáticamente. La numeración del código de barras viene en la respuesta asíncrona que es utilizada para llamar al servicio de OpenFoodFacts. No se necesita absolutamente nada más.

```
77  scan() {
78    this.barcodeScanner.scan()
79      .then(data => {
80        this.apiService.getFoodFacts(data.text)
81          .subscribe(res => {
82            this.result = res;
83            this.nutrients = this.result.product.nutrient_levels;
84            this.nutrientKeys = Object.keys(this.nutrients);
85            this.allergens = this.result.product.allergens.split(',');
86            if (this.allergens.length > 0) {
87              this.hasAllergens = true;
88            }
89            this.additives = this.result.product.additives_tags;
90          })
91      })
92  }
93
```

Figura 4.26: Código del servicio de Escáner.

En relación a la vista, hay un detalle que podría ser interesante de mencionar y es el caso de la creación de un pipe. Si se observa la Figura 4.27, se puede ver cómo en las líneas 55 y 56 se hace uso de un pipe personalizado. Esto es porque en la vista, deseamos mostrar los campos del objeto recibido con un formato correcto: es decir, puede que el texto de este valor key sea, por ejemplo, “fats“, pero nosotros queremos mostrarlo como “Grasas“. La forma más correcta y eficiente de transformar datos en la vista es a través de pipes, y aunque Ionic proporciona algunos predefinidos muy útiles, siempre podemos crear nuestros propios pipes.

El código de un pipe es muy intuitivo, como se observa en la Figura 4.28. Basta con crear un componente Pipe con la ayuda del CLI de Ionic, que ya nos generará buena parte del código necesario. Un pipe posee un método transform que recibe una entrada (lo que precede al pipe, en nuestro caso “key“) y una serie de parámetros opcionales que en nuestro caso no son necesarios. A continuación, tan solo debe implementarse la lógica deseada y el valor de retorno para cada caso. Este pipe, de hecho, es una aproximación

```
52
53                  <ion-col size="12">
54                    <ion-row *ngFor="let key of nutrientKeys">
55                      <ion-col size="6">{{ key | translate }}</ion-col>
56                      <ion-col size="6">{{ nutrients[key] | translate }}</ion-col>
57                    </ion-row>
58
```

Figura 4.27: Código de la vista del resultado del escáner. Nótese el pipe usado para formatear el texto.

muy sencilla a cómo funciona la internacionalización en Ionic. En un caso real, el pipe consulta las variables de texto declaradas en un fichero JSON y aplica la traducción según la opción almacenada en el sistema. Como este caso de uso era muy sencillo, se decidió posponer esta aproximación más elaborada.

```
1   import { Pipe, PipeTransform } from '@angular/core';
2
3   @Pipe({
4     name: 'translate'
5   })
6   export class TranslatePipe implements PipeTransform {
7
8     transform(value: string, arg): string {
9
10      switch(value){
11
12        case 'fat':
13          return 'Grasas'
14
15        case 'salt':
16          return 'Sal'
17
18        case 'saturated-fat':
19          return 'Grasas saturadas'
```

Figura 4.28: Código necesario para crear un pipe.

El resto del código de la aplicación siempre tiene el mismo aspecto: vistas que se apoyan en los servicios reutilizables. Una vez se asimilan los conceptos básicos de Ionic y Angular, el aspecto de un proyecto por grande que sea es siempre el mismo. Normalmente, en este tipo de aplicaciones orientadas a casos de uso y basadas en arquitecturas MCV o MVVM, los primeros pasos de desarrollo son más lentos al ser necesario crear modelos, entidades o servicios, pero una vez que se tiene el modelo de negocio correctamente modelado y se habla únicamente de crear nuevos casos de uso, la velocidad de desarrollo es considerable, además de fiable.

## 4.3. Aspecto final de la aplicación

Para desarrollar la aplicación se ha utilizado el navegador Chrome con las herramientas de depuración incorporadas. No obstante, después de crear cada funcionalidad es una buena práctica realizar pruebas en un dispositivo nativo, ya sea emulado o real. En las tecnologías híbridas como Ionic o React Native, el aspecto final de la aplicación en un dispositivo nativo no siempre es idéntico al observado en la versión web, aunque esto justamente en Ionic ocurre en muy raros casos. Las capturas de pantalla adjuntas en este capítulo han sido obtenidas de la versión web para una mayor comodidad, pero su

apariencia no guarda diferencia alguna con la versión móvil ejecutada en un dispositivo real.

**Nota importante:** los valores de los diferentes compuestos FODMAP de los alimentos así como el semáforo calculado por la aplicación que se mostrará en las siguientes imágenes no son reales. Estos valores han sido generados aleatoriamente para la realización de pruebas, encontrándose los verdaderos en libros recomendados por personal médico y de la salud y que deben ser revisados antes de formar parte de un entorno real.

La primera pantalla que aparece al abrir la aplicación es la lista de alimentos y su semáforo FODMAP, tal como se aprecia en la Figura 4.29. En la parte superior izquierda, puede desplegarse el menú para navegar a las diferentes pantallas, tal como se muestra en la Figura 4.30. A continuación, siguiendo el orden de este menú se hará un recorrido de todas las funcionalidades.

### La lista de alimentos

La lista de alimentos es la pantalla principal de la aplicación y permite tener una rápida visión general de los alimentos que se puede ingerir. Si se selecciona un alimento, se entra en la vista de detalles. Esta pantalla tiene dos funciones: por un lado, informar con más detales de los motivos del semáforo FODMAP del alimento. Esta información varía dinámicamente si se desplaza el selector de cantidad, tal como se aprecia en la Figura 4.31. También encontramos un botón .Añadir al diario". Esto permite marcar que se ha ingerido el alimento y quedará así registrado en el diario de comidas junto con la cantidad. El contenido del diario influye en el semáforo de los alimentos, como se mostrará más adelante.

### Los platos

Los platos permiten agrupar ingredientes para un uso más cómodo de la aplicación. En la Figura 4.32 se muestra de izquierda a derecha, el caso de uso completo de crear un plato y ver sus detalles. Al seleccionar en el menú la opción "Mis platos", se llega a la primera pantalla, donde se muestra la lista de platos existentes o se invita al usuario a crearlos si no existen. Si se opta por crear un plato, se llega a la segunda pantalla de la imagen. En esta pantalla aparecen todos los ingredientes y seleccionando los mismos se pueden ir marcando o desmarcando para conformar el plato. El selector de la parte inferior ajusta la cantidad de cada ingrediente, afectando siempre al último seleccionado. Una vez se tiene seleccionados todos los alimentos en las cantidades deseadas y se le ha otorgado un nombre al plato, se habilita el botón Çrearz ahora el plato aparecerá en la pantalla principal como se muestra. En esta vista encontramos el semáforo FODMAP del plato y un botón que permite, al igual que en el caso de los alimentos, anotarlo como ingerido y enviarlo al diario. Si se selecciona el plato, se llega a la última pantalla de la imagen, donde se muestran sus detalles y se da la posibilidad de eliminar el plato.

### El escáner

El escáner de alimentos tiene como objetivo intentar dar cierta orientación sobre alimentos comerciales que se encuentran en tiendas y supermercados. No es posible saber la cantidad de compuestos FODMAP que estos alimentos contienen, pero sí al menos si los contiene. Además, proporciona otra información útil para la salud. Al entrar en esta opción desde el menú de la aplicación, automáticamente se abre la cámara del

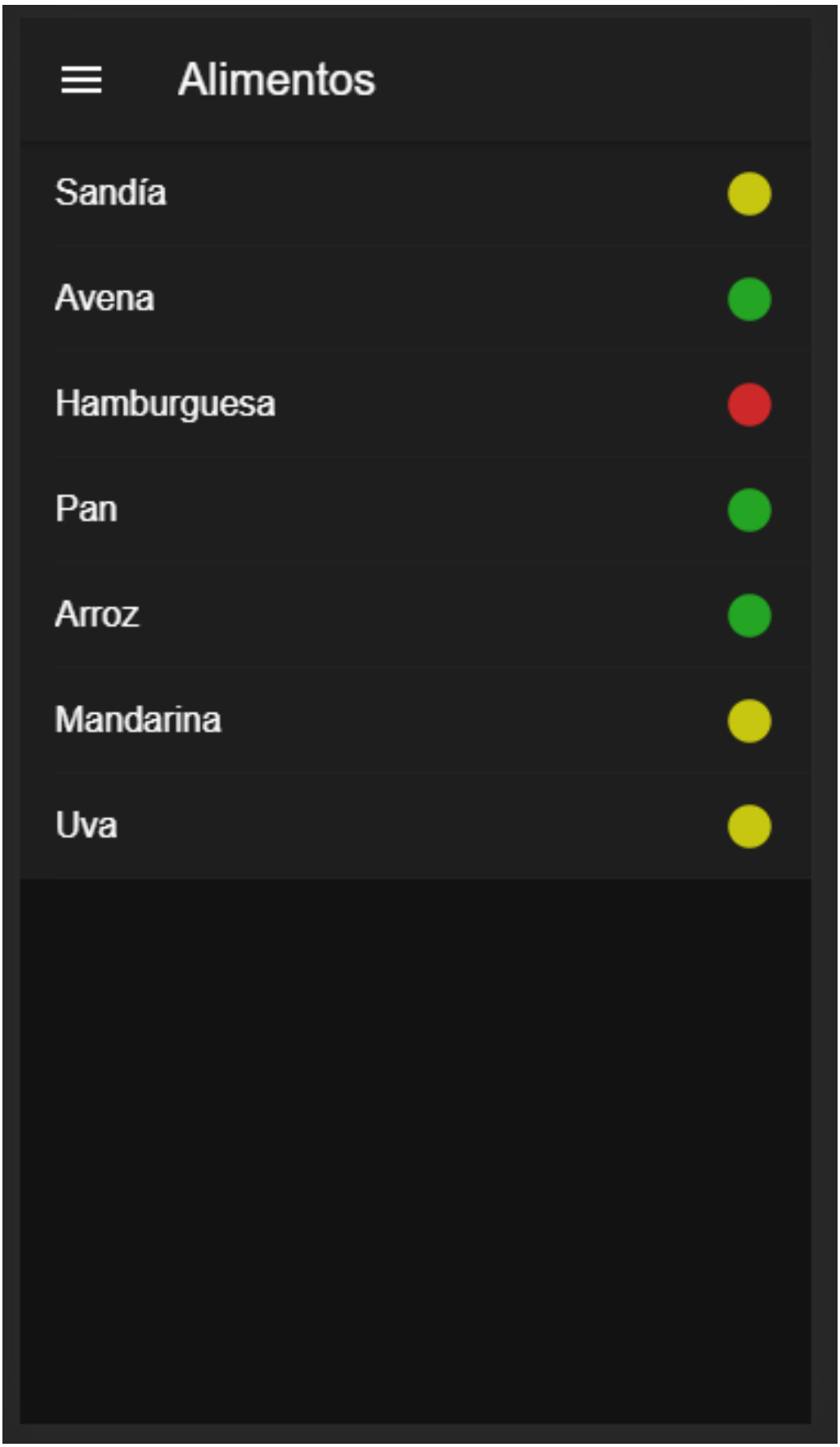


Figura 4.29: Lista de alimentos con su valor FODMAP.

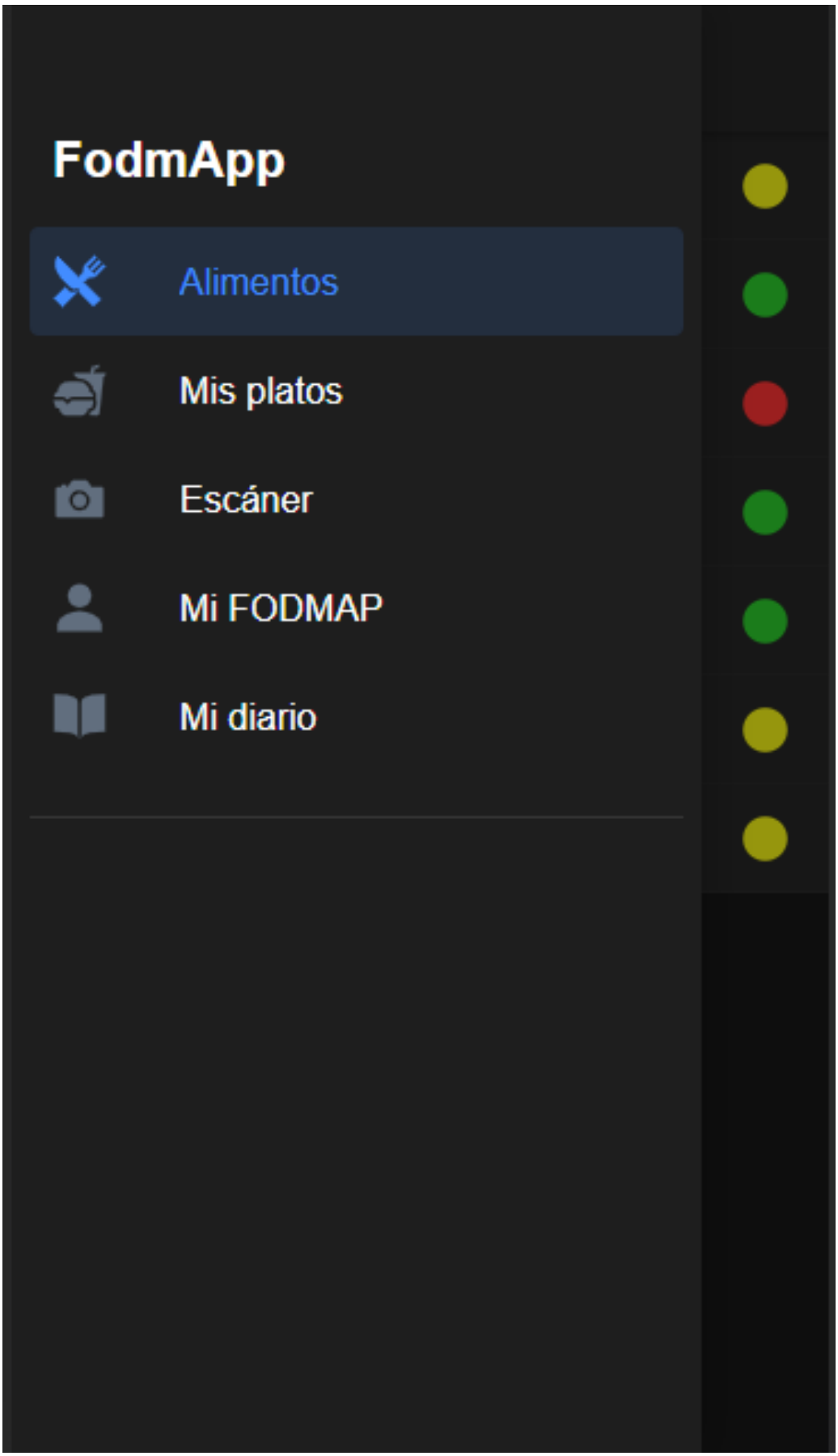


Figura 4.30: Menú de la aplicación.

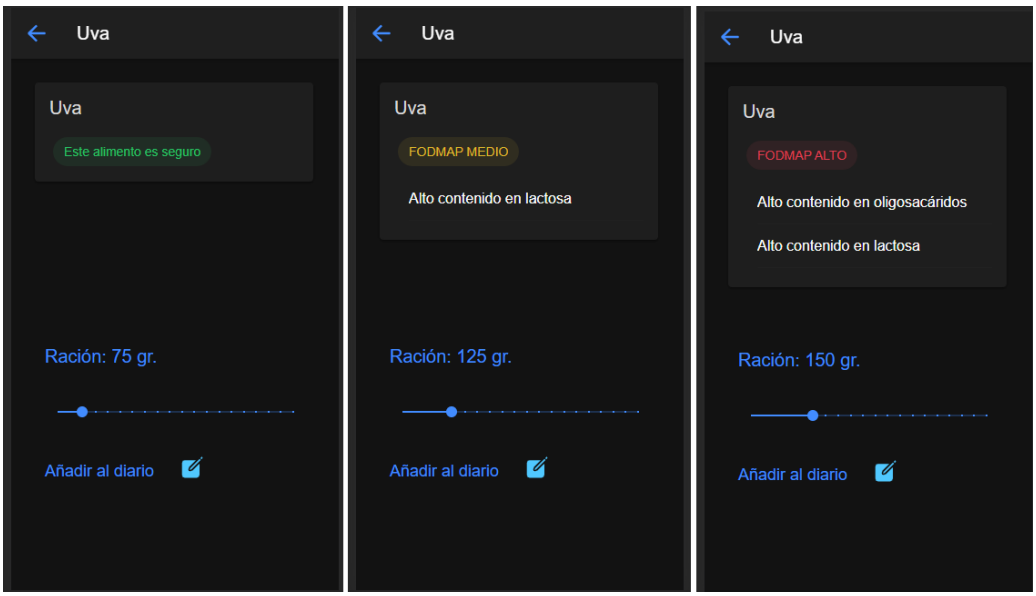


Figura 4.31: Vista de detalles de un alimento.

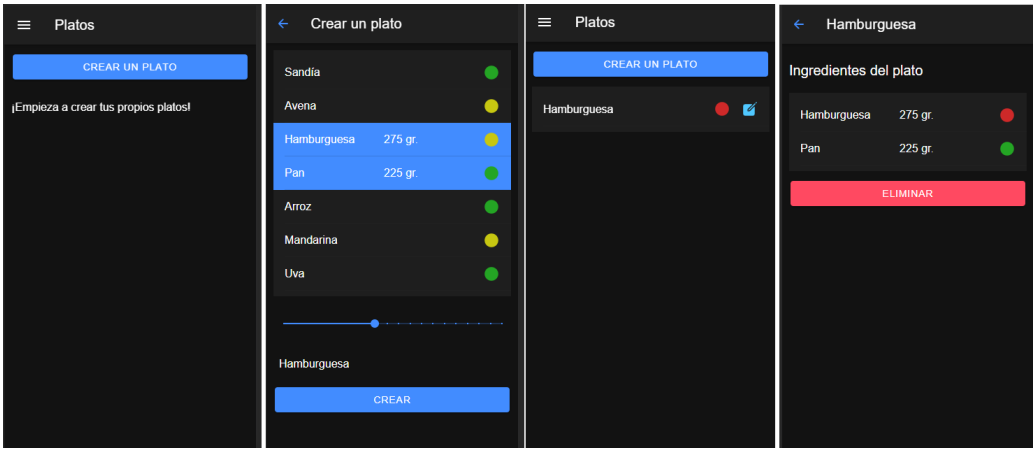


Figura 4.32: Creación de un plato.

dispositivo. No es necesario tomar ninguna fotografía: tan pronto como se detecte un código de barras se mostrará el resultado del procesamiento tal como se aprecia en la Figura 4.33.

### Las tolerancias

Desde la opción "Mis tolerancias"es posible ajustar las tolerancias a los diferentes compuestos FODMAP. Esta pantalla no requiere de gran explicación. Su uso es simple e intuitivo.

### El diario

El diario registra todos los alimentos que se han comido durante el día y en qué cantidad. El usuario debe limpiar el diario cada día o cuando lo considere oportuno desde su última digestión. En la Figura 4.35 se observa un ejemplo de caso de uso. Primero, desde la lista básica de alimentos, tal como se mostró al inicio de esta sección, se ingieren 100 gramos de avena. Luego, se acude a la vista de platos y se ingiere también un plato que se ha creado con varias frutas. Si en este momento se acude al diario, el resultado

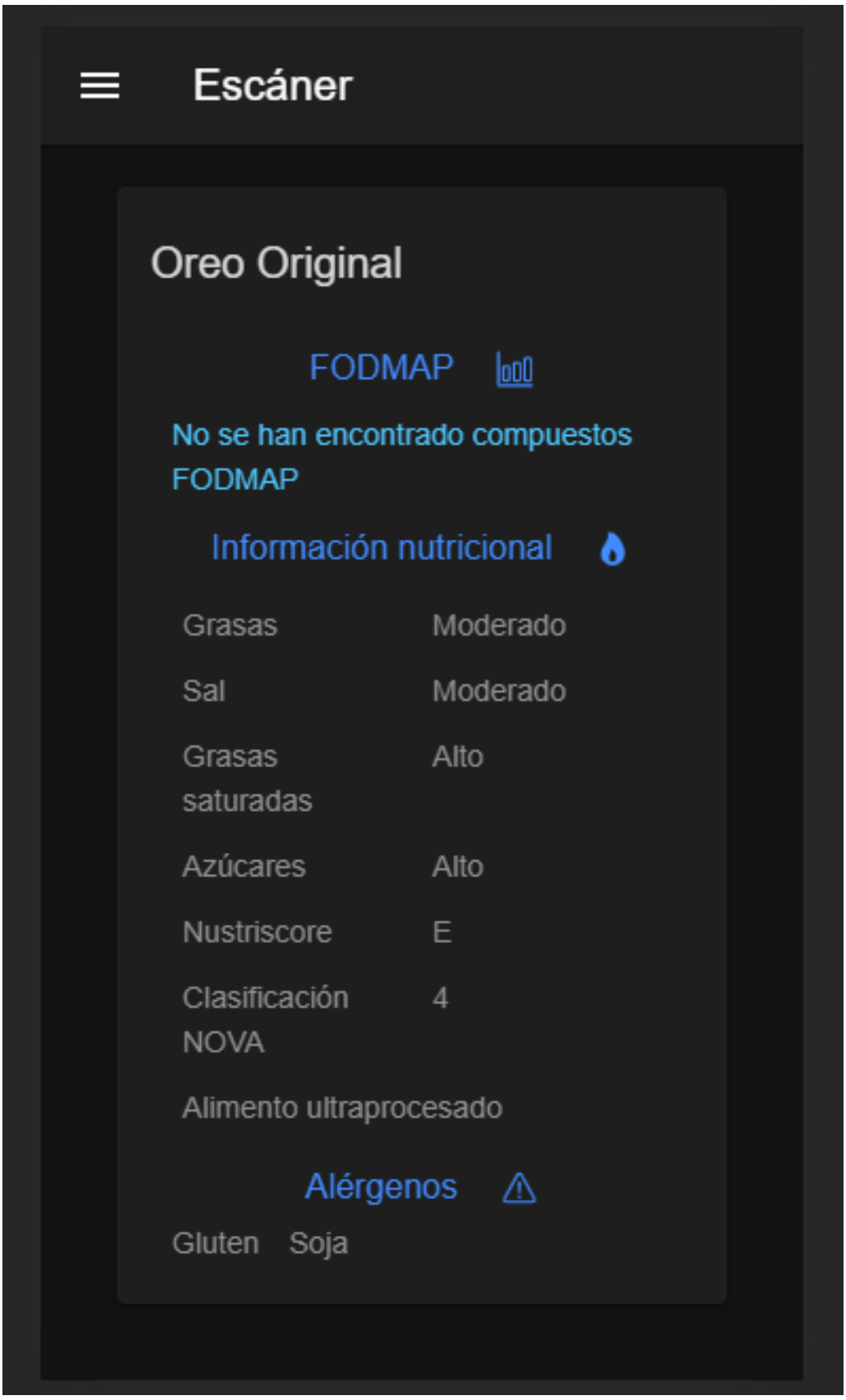


Figura 4.33: Resultado del escáner.

será la imagen de la derecha, donde se puede ver cada ingrediente ingerido, su cantidad y la hora de la ingesta.

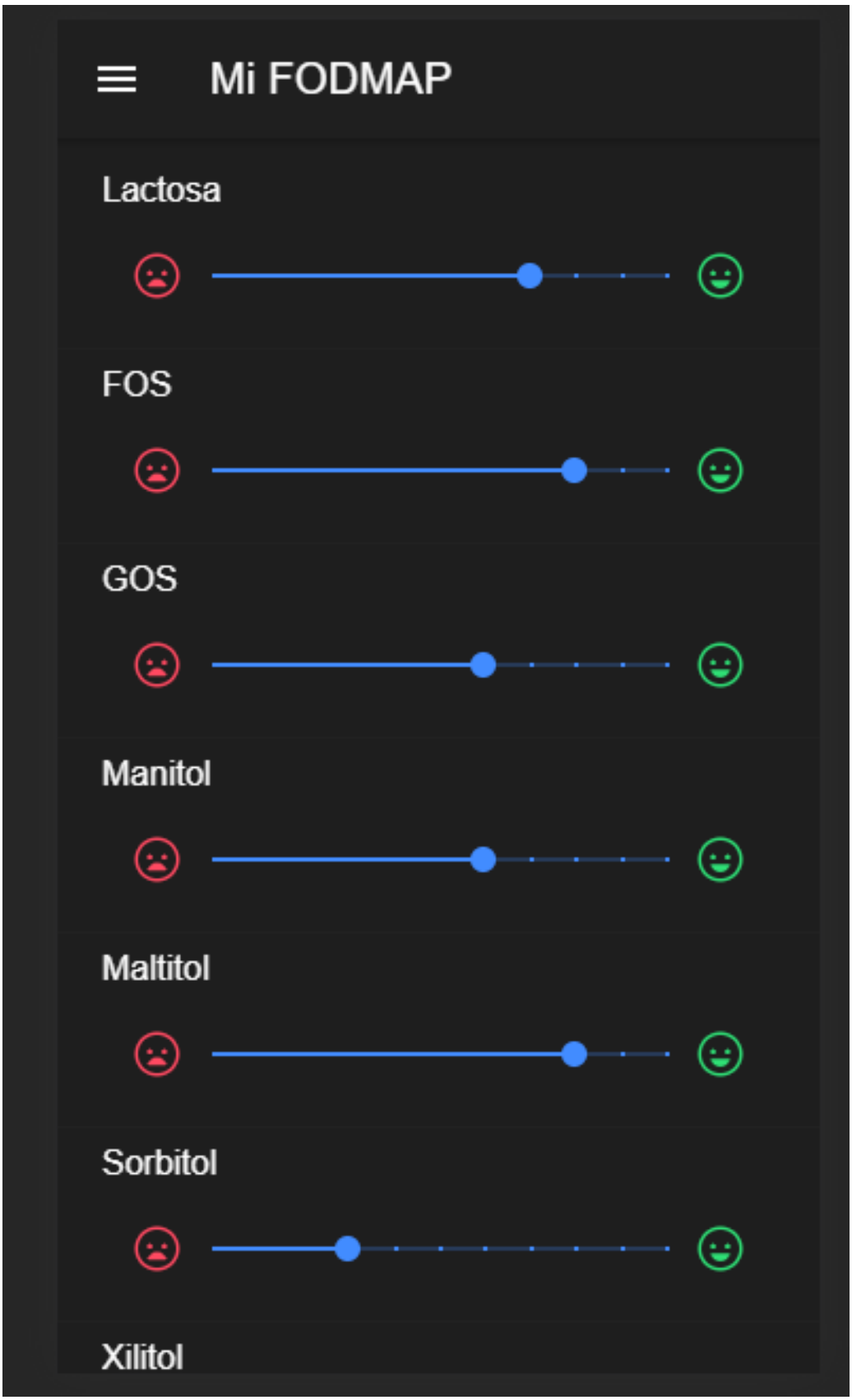


Figura 4.34: Vista de ajuste de las tolerancias.

**Semáforo dinámico de la aplicación**

Algo que es importante comprender es el comportamiento dinámico de la aplicación respecto a los valores FODMAP. El semáforo FODMAP depende en todo momento de

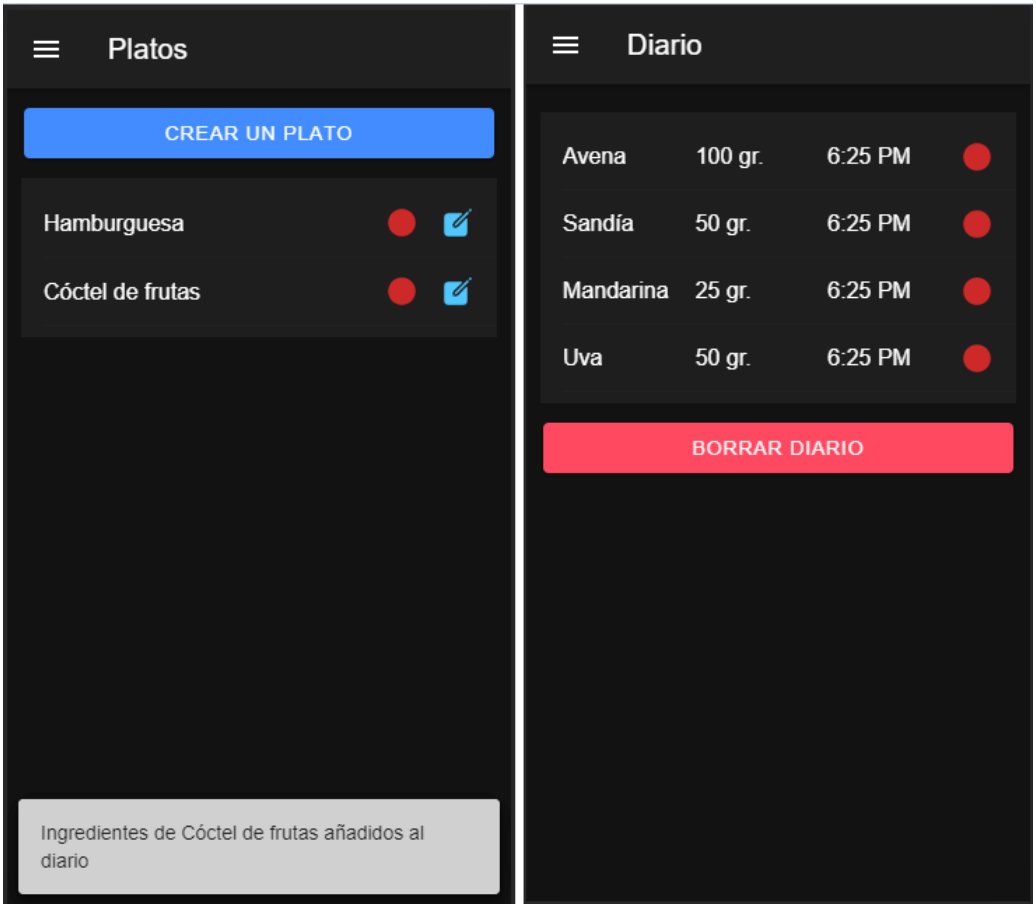


Figura 4.35: Diario de la aplicación. Nótese que en el diario solo constan ingredientes y no platos.

los alimentos ingeridos y su cantidad y de las tolerancias del usuario. Esto funciona en todo momento y no únicamente en las pantallas principales. Por ejemplo, si se está seleccionando ingredientes para crear un plato, el semáforo FODMAP de estos ingredientes dependerá en tiempo real de las tolerancias y el contenido del diario. En la Figura 4.36 se muestra, de izquierda a derecha un ejemplo de esta característica. En primer lugar, se observa la lista de alimentos con su semáforo FODMAP. Como se aprecia en la segunda pantalla, 100 gramos de uva es por ahora una ingesta segura, así que se realiza dicha ingesta. Ahora, en la lista, la uva aparecerá con semáforo FODMAP en rojo, y si volvemos a entrar a sus detalles (tercera pantalla) vemos que 100 gramos de uva ya no son una ingesta permitida y de hecho tan solo 25 gramos presentan contenido alto en oligosacáridos, siendo su semáforo amarillo. En la última pantalla, se observa los detalles del plato Çóctel de frutas"que se creo con anterioridad. Este plato ahora posee semáforo rojo debido a que cuenta con 50 gramos de uva, valor que en la situación actual es prohibitivo. Si se limpian las entradas del diario o se ajustan las tolerancias, este valor volvería a cambiar.

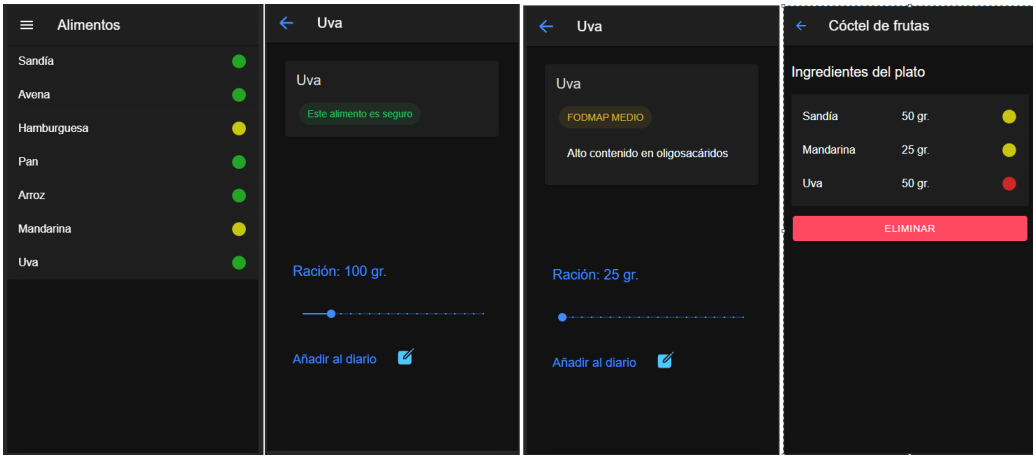


Figura 4.36: Comportamiento dinámico del semáforo FODMAP.

# Capítulo 5

## Conclusiones y líneas futuras

Durante la realización de este proyecto, si algo ha predominado a diferentes niveles es la incertidumbre. Por un lado, gran parte de las tecnologías utilizadas eran desconocidas y nuevas. El problema de utilizar tecnologías demasiado recientes es que no existen garantías de que cubran todos los futuros casos de uso o necesidades que puedan surgir. Por otra parte, también existía la incertidumbre de la obtención de los datos o siquiera de si sería posible obtener los mismos. De igual forma, al desconocer el modelo de negocio no se tenía garantías de que el modelo de datos elegido y la lógica aplicada fueran exactamente los correctos. Durante la realización de este proyecto se ha seguido por tanto una metodología muy ágil y participativa, y cada semana se desconocía que nuevas ideas o retos surgirían la siguiente.

Sin embargo, todo esto no es malo y de hecho forma parte del proceso de desarrollo de software, mucho más cuando no se trata de un software de naturaleza empresarial cuyos conceptos y diseños ya se encuentran muy maduros y evolucionados, sino de un proyecto innovador que intenta abrirse paso entre muchos interrogantes. Después de décadas de intentos frustrados por las mejores compañías, ha quedado patente que el desarrollo de software es muy difícil de planificar. Esto es un hecho y hay dos formas de afrontarlo: insistir en el camino de cumplir con una planificación exacta e intentar un desarrollo en cascada que funcione a la primera o intentar aceptar y fluir con esta naturaleza cambiante del software y diseñar y desarrollar en consecuencia. Normalmente, los proyectos que se basan en la segunda corriente de pensamiento suelen tener mayor garantía de éxito.

En mi opinión, un software de calidad no es aquel que no necesita cambios, porque en el mundo real esto es sencillamente una utopía. Un software de calidad es aquel que responde rápidamente a cambios y adaptaciones , es fácil de construir y evolucionar sin producir efectos colaterales. Durante el desarrollo de este proyecto, esta ha sido la filosofía que se ha intentado seguir. Por ejemplo, en el estado actual del proyecto a fecha de la elaboración de esta memoria, es posible plantearse si el enfoque cliente-servidor ha resultado ser estrictamente necesario y si podría ser posible pasar a una aplicación standalone única. ¿Representa esto el fracaso o la poca calidad del proyecto? En absoluto: forma parte del proceso natural, sobre todo en una aplicación tan experimental. La calidad del proyecto quedaría mejorado por el coste o los efectos colaterales de implementar este cambio. De hecho, si se quisiera llevar a cabo, sería tan sencillo como reescribir únicamente el servicio de API para que los datos en lugar de mediante una solicitud a un servidor, se obtuviesen, por ejemplo, de un fichero local. El resto de la aplicación no se enteraría del cambio. En mi opinión, este es el verdadero indicador de calidad del software, siendo preferible aquel que debe cambiar una decena de veces pero lo hace

con suma facilidad que el que solo cambia una vez pero siendo destruir o readaptar gran parte del mismo.

A pesar del enorme recorrido que supone desarrollar un software hasta el punto de ser apto para su explotación y que este proyecto aún necesitaría en dicho caso, no es aventurado decir que hasta este punto, el proyecto ha conseguido sus objetivos, los cuales fueron descritos en el Capítulo 1:

- Se pretendía lograr una aplicación que al contrario de muchas existentes, no fuese una mera tabla de valores, ajustándose dinámicamente a las tolerancias del usuario y a lo que ha comido a lo largo del día, objetivo que ha sido cumplido satisfactoriamente.
- Se ha introducido el concepto de plato con el que ninguna aplicación cuenta a fecha de realización de este trabajo, permitiendo agrupar y consumir múltiples alimentos frecuentes de una sola vez, mejorando la comodidad del usuario y dejando la puerta abierta a potenciales mejoras como podría ser un manual de recetas.
- Se ha dotado a la aplicación de un escáner de código de barras, algo con lo que tampoco cuenta ninguna aplicación hasta la fecha y que muchos usuarios demandan. La base de datos aún no posee información para dar las cantidades exactas de compuestos FODMAP, pero desde OpenFoodFacts se está trabajando en ello debido al auge de esta dieta. Implementar este cambio el día que la información estuviese disponible no sería costoso y precisamente este era el último objetivo que se había planteado al inicio de este trabajo.
- La aplicación se ha desarrollado siguiendo una arquitectura muy modularizada que le permite ser modificada o escalada con seguridad y simplicidad, pudiendo adaptarse rápidamente a cambios o novedades.

Como posibles líneas futuras del proyecto si se dispusiera de más tiempo, las divido en dos categorías: las de carácter técnico y las de carácter funcional.

A nivel técnico:

- Implementar mecanismos de securización y autorización si se decide continuar por un enfoque cliente-servidor. Dado que los clientes únicamente consultarán los alimentos pero no los modificarán o añadirán nuevos, estando esta funcionalidad reservada para el administrador, puede directamente no exponerse estas rutas al exterior desde un nivel de capa de red más bajo sin necesidad de llegar a la aplicación.
- Refactorizar el código y sanar la deuda técnica que se ha acumulado en algunos puntos para avanzar en la entrega del producto. Por ejemplo, el diario debería disponer de su propio servicio para obtener y guardar alimentos.
- Versionar el código adecuadamente utilizando versionado semántico. Separar entornos de desarrollo y producción, comenzar a diseñar y utilizar una estrategia de branching, así como en herramientas de testing, integración y despliegue continuo antes de que el proyecto escale demasiado.
- Utilizar el plugin nativo de MySQL que ofrece Ionic en lugar del almacenamiento simple basado en clave-valor, mucho más ineficiente y que obliga a escribir más código al no poder realizar consultas.

A nivel funcional:

- Implementar un asistente de seguimiento personalizado. La dieta FODMAP suele realizarse por semanas donde al principio se prohíben todos los compuestos y gradualmente se van subiendo las tolerancias a los mismos. De esta forma se llega a un grado de tolerancia óptimo que evita las molestias de esta afección pero otorgando al paciente una mayor calidad de vida al restringir los alimentos justos y en su justa medida. Un asistente virtual haría un seguimiento al usuario preguntándole por sus síntomas, cómo se encuentra, recordándole que debe ajustar las tolerancias, etc...
- Gamificar la dieta FODMAP y la dieta saludable. Podría plantearse una serie de logros o puntos que motivasen al usuario a conseguir sus objetivos como hacen muchas aplicaciones o videojuegos. Por ejemplo, medallas que se desbloqueasen y coleccionasen al conseguir distintos logros, como podría ser ingerir menos de cierta cantidad de azúcar o grasas durante una semana.
- Crear funciones sociales que conecte a los usuarios y permitan crear una comunidad que retroalimente la aplicación. Esto abriría la puerta a utilizar herramientas de análisis de datos y permitiría, por ejemplo, ayudar a recomendar alimentos sobre los que no se tiene datos. Por ejemplo, al utilizar el escáner de alimentos, se desconocen las cantidades de los compuestos FODMAP, pero con una comunidad lo suficientemente grande se le podría indicar al usuario que otras personas con tolerancias similares a la suya han reportado sentirse mal o no notar molestias tras consumirlo.

En conclusión, ha sido un proyecto muy interesante y ameno de realizar en el que además han intervenido todos o muchos niveles de la pila de desarrollo necesaria para crear un producto, temática precisamente propia de la titulación a la que se opta con la presentación de esta memoria. Como primer prototipo, podría decirse que el proyecto ha sido un éxito, ya que se han cumplido todos los objetivos propuestos en el planteamiento y además de la forma en que se pretendían cumplir. En un caso real para preparar una aplicación comercial, esta fase habría sido la equivalente a un spike y ahora, tras analizar este primer prototipo, debería tomarse una serie de decisiones sobre el camino por el que finalmente se optará para comenzar a trabajar en un desarrollo más orientado a una explotación real.

# Capítulo 6

## Summary and Conclusions

During the realization of this project, if something has predominated at different levels, it is uncertainty. On the one hand, a large part of the technologies used were unknown and new. The problem with using technologies that are too recent is that there is no guarantee that they will cover all future use cases or needs that may arise. On the other hand, there was also the uncertainty of obtaining the data or even whether it would be possible to obtain them. Similarly, by not knowing the business model, there was no guarantee that the chosen data model and the applied logic were exactly correct. During the execution of this project, therefore, a very agile and participatory methodology has been followed, and each week it was unknown what new ideas or challenges would arise the next.

However, all this is not bad and in fact it is part of the software development process, much more when it is not a business software whose concepts and designs are already very mature and evolved, but an innovative project that try to break through many questions. After decades of frustrated attempts by the best companies, it has become patent that software development is very difficult to plan. This is a fact and there are two ways to deal with it: sticking to the path of attemping to do a cascade development, or trying to embrace and flow with this changing nature of software and design and develop accordingly. Typically, projects that are based on the second stream of thought tend to have a greater guarantee of success.

In my opinion, quality software is not one that does not need changes, because in the real world this is simply a utopia. Quality software is one that responds quickly to changes and adaptations, it is easy to build and evolve without producing collateral effects. During the development of this project, this has been the philosophy that has been tried to follow. For example, in the current state of the project as of the writing of this report, it is possible to consider whether the client-server approach has turned out to be strictly necessary and whether it might be possible to move to a single standalone application. Does this represent the failure or poor quality of the project? Not at all: it is part of the natural process, especially in such an experimental application. The quality of the project would be improved due to the cost or the collateral effects of implementing this change. In fact, if we wanted to implement this change, it would be as simple as rewriting only the API service so that the data, instead of a request to a server, would be obtained, for example, from a local file. The rest of the app wouldn’t know about this change. In my opinion, this is the true indicator of software quality, being preferable the one that changes a dozen times but does it so easily than the one that only changes once but making the developerdestroy or re-adapt a large part of it.

Despite the enormous journey involved in developing software to the point of being suitable for exploitation and that this project would still need in that case, it is not risky

to say that up to this point, the project has achieved its objectives, which were described in Chapter 1:

- It was intended to achieve an application that, unlike many existing ones, was not a mere table of values, dynamically adjusting to the user's tolerances and what they have eaten throughout the day, a goal that has been satisfactorily accomplished.
- The concept of a plate has been introduced with which no application has as of the date of this work, allowing to group and consume multiple frequent foods at one time, improving user comfort and leaving the door open to potential improvements as could be a recipe manual.
- The application has been equipped with a barcode scanner, something that no application has to date and that many users demand. The database does not yet have information to give the exact amounts of FODMAP compounds, but OpenFoodFacts is working on it due to the rise of this diet. Implementing this change the day the information was available would not be expensive and this was precisely the last objective that had been set at the beginning of this work.
- The application has been developed following a highly modularized architecture that allows it to be modified or scaled with security and simplicity, being able to adapt quickly to changes or novelties.

As for the possible future lines of the project, we divide them according two categories: technical and functional.

From a technical point of view:

- Implement security and authorization mechanisms if you decide to continue with a client-server approach. Since customers will only consult the food but will not modify or add new ones, this functionality being reserved for the administrator, these routes may not be directly exposed to the outside from a lower network layer level without having to reach the application.
- Refactor the code and heal the technical debt that has accumulated at some points to advance the delivery of the product. For example, the newspaper should have its own service for obtaining and storing food.
- Versioning the code appropriately using semantic versioning. Separate development and production environments, start designing and using a branching strategy, as well as testing tools, integration and continuous deployment before the project scales too much.
- Use the native MySQL plugin that Ionic offers instead of simple key-value-based storage, which is much more inefficient and forces you to write more code as you cannot perform queries.

From a functional point of view:

- Implement a custom tracking wizard. The FODMAP diet is usually carried out for weeks where at the beginning all compounds are prohibited and tolerances are gradually increased to them. In this way, an optimal degree of tolerance is reached

that avoids the discomfort of this condition but granting the patient a higher quality of life by restricting the right food and the right amount. A virtual assistant would follow up with the user, asking about their symptoms, how they are doing, reminding them to adjust tolerances, etc ...

- Gamify the FODMAP diet and the healthy diet. A series of achievements or points could be raised that motivate the user to achieve their goals as many applications or video games do. For example, medals that were unlocked and collected by achieving different achievements, such as eating less than a certain amount of sugar or fat during a week.

- Create social functions that connect users and allow the creation of a community that gives feedback on the application. This would open the door to using data analysis tools and would allow, for example, to help recommend foods for which there is no data. For example, when using the food scanner, the amounts of the FODMAP compounds are unknown, but with a large enough community it could be indicated to the user that other people with tolerances similar to yours have reported feeling bad or not noticing discomfort after consume it.

In conclusion, it has been a very interesting and enjoyable project to carry out in which all or many levels of the development stack necessary to create a product have also intervened, which is the main theme of the current degree that is trying to be obtained with this work. As a first prototype, it could be said that the project has been a success, since all the objectives proposed in the approach have been met and in addition to the way in which they were intended to be met. In a real case to prepare a commercial application, this phase would have been the equivalent of a spike and now, after analyzing this first prototype, a series of decisions should be made about the path that will finally be chosen to start working on a development more oriented to a real exploitation.

# Apéndice A

## Repositorio del proyecto

Se adjunta la dirección del repositorio Github que contiene el proyecto junto con el tag del último commit, que garantiza el estado del mismo en fecha previa al límite de entrega del presente trabajo.

`https://github.com/PAL-ULL/tfm-2021-alexrcas #81896d`

# Bibliografía


[1] Capacitor documentación oficial. `https://capacitorjs.com/docs`. Accessed: 2021-04-04.

[2] Ionic documentación oficial. `https://ionicframework.com/docs`. Accessed: 2021-04-15.

[3] MoleculerJS documentación oficial. `https://moleculer.services/docs/0.14/`. Accessed: 2021-03-01.

[4] NodeJS documentación oficial. `https://nodejs.org/es/docs`. Accessed: 2021-05-10.

[5] VueJS documentación oficial. `https://vuejs.org/v2/guide`. Accessed: 2021-05-01.

[6] Kamran Ayub. *Introduction to TypeScript*. Packt Publishing, 2017.

[7] Jacqueline S Barrett. How to institute the low-fodmap diet. *Journal of gastroenterology and hepatology*, 32:8–10, 2017.

[8] Bryan Basham, Bert Bates, and Kathy Sierra. *Head First Servlets and JSP, 2nd Edition*. O'Reilly Media, Inc, 2008.

[9] J. Bermúdez Alonso. La dieta fodmap para el colon irritable. *I Congreso Internacional Virtual de Nutrición Clínica Práctica*, 2018.

[10] Evan Burchard. *Refactoring JavaScript*. O'Reilly Media, Inc, 2017.

[11] Kristina Chodorow. *MongoDB: The Definitive Guide*. O'Reilly Media, Inc, 2010.

[12] Brajesh De. *API Management: An Architect's Guide to Developing and Managing APIs for Your Organization*. Apress, 2017.

[13] Fernando Doglio. *REST API Development with Node.js : Manage and Understand the Full Capabilities of Successful REST Development*. Apress, 2018.

[14] Jeremy Wilken. *Ionic in Action: Hybrid Mobile Apps with Ionic and AngularJS*. Manning Publications, 2015.

[15] Jim Wilson. *Node.js 8 the Right Way*. Pragmatic Bookshelf, 2018.