

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

Proyecto de Fin de Grado en Ingeniería Informática

**Desarrollo de aplicación web full-stack de  
 asesoramiento nutricional con   
sistemas basados en reglas**

Michael Laudrup

Dirigido por:

Curso: 2021-2022, Septiembre



**Desarrollo de aplicación web full-stack de**

**asesoramiento nutricional con**

**sistemas basados en reglas**

**Proyecto de Fin de Grado en Ingeniería Informática**

**de modalidad específica**

Realizado por: nombre alumno

Dirigido por: nombre tutor

Fecha de lectura y defensa: XX de Junio de 2022

Agradecimientos

Resumen

Este trabajo de fin de grado consiste en el desarrollo de una aplicación full stack con un sistema basado en reglas cuyo objeto es el de servir como asesor nutricional inteligente a las personas que quieren conseguir unos objetivos de forma física a través de una alimentación saludable, actuando principalmente como asesor nutricional. Además, dado que el dominio de negocio, es decir, la nutrición y dietética, no entran en el ámbito del grado en ingeniería informática, este trabajo de fin de grado se limitará al análisis de la parte más técnica del proyecto.

Como aspecto a reseñar este trabajo aborda la mayoría de las ramas de la informática, como, desarrollo de interfaces gráficas, usabilidad y accesibilidad, protocolos de comunicación, implementación de patrones arquitectónicos avanzados tanto en front como en back, ciberseguridad, inteligencia artificial, bases de datos, desarrollo por el lado del servidor basado en el patrón MVC , pruebas, testing, virtualización con docker y despliegue de aplicaciones, aderezándolo finalmente con un análisis de la situación actual del mundo profesional en el campo de desarrollo de aplicaciones full-stack.

Palabras clave

|  |  |
| --- | --- |
| Concepto | Defnición |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |
|  |  |

# Índice general

[Índice general 7](#_Toc115003906)

[Índice de tablas 9](#_Toc115003907)

[Índice de figuras 10](#_Toc115003908)

[Capítulo 1. Introducción, objetivos y estructura 11](#_Toc115003909)

[1.1 Introducción 11](#_Toc115003910)

[1.2 Objetivos 12](#_Toc115003911)

[1.2.1. Objetivos tecnológicos. 12](#_Toc115003912)

[1.2.2. Objetivos del dominio sobre el que se trabaja. 12](#_Toc115003913)

[1.2.3. Objetivos de contribución a la comunidad científica. 12](#_Toc115003914)

[1.3 Estructura del documento 13](#_Toc115003915)

[Capítulo 2. Estado del arte 14](#_Toc115003916)

[2.1 Introducción. 14](#_Toc115003917)

[2.2 ¿Qué es una aplicación web full stack? 14](#_Toc115003918)

[2.3 Ventajas de las aplicaciones web full stack 14](#_Toc115003919)

[2.4 Desventajas de las aplicaciones web full stack. 15](#_Toc115003920)

[2.5 Tendencias actuales de las aplicaciones web full-stack en el mundo profesional. 15](#_Toc115003921)

[2.6 Problemas detectados en el desarrollo de aplicaciones web y soluciones aportadas en este trabajo de fin de grado. 16](#_Toc115003922)

[2.6.1. Alto acoplamiento entre lado del cliente y servidor con patrones clásicos de MVC en el lado del servidor. 16](#_Toc115003923)

[2.6.2. Mala definición de arquitecturas de proyectos software. 18](#_Toc115003924)

[2.6.3. Componentes front-end altamente acoplados con peticiones al lado del clientes o fragmentación de los estados de una aplicación. 19](#_Toc115003925)

[2.6.4. Grandes anidaciones de sentencias condicionales 20](#_Toc115003926)

[2.6.5. Tabla resumen de inconvenientes y soluciones. 21](#_Toc115003927)

[Capítulo 3. Análisis 23](#_Toc115003928)

[3.1 Modelo de dominio 23](#_Toc115003929)

[3.2 Análisis de requisitos 23](#_Toc115003930)

[Capítulo 4. Diseño 25](#_Toc115003931)

[4.1 Introducción 25](#_Toc115003932)

[4.2 Diseño de base de datos. 25](#_Toc115003933)

[4.2.1. Entidades de la aplicación. 25](#_Toc115003934)

[4.2.2. Relaciones entre entidades. 27](#_Toc115003935)

[4.3 Diseño de sistema basado en reglas. 29](#_Toc115003936)

[4.4 Diseño arquitectónico del servidor. 30](#_Toc115003937)

[4.5 Diseño de sistema de autenticación y autorización (Ciberseguridad). 31](#_Toc115003938)

[4.6 Contratos de comunicación servidor-cliente 32](#_Toc115003939)

[4.7 Diseño arquitectónico del front-end (lado cliente). 32](#_Toc115003940)

[4.8 Conclusiones. 32](#_Toc115003941)

[Capítulo 5. Implementación 33](#_Toc115003942)

[5.1 Introducción 33](#_Toc115003943)

[5.2 Stack tecnológico del proyecto. 33](#_Toc115003944)

[5.3 Implementación de arquitectura back-end. 35](#_Toc115003945)

[5.4 Implementación de base de datos. 35](#_Toc115003946)

[5.4.1. Entidades de la relación. 35](#_Toc115003947)

[5.5 Implementación de sistema basado en reglas. 36](#_Toc115003948)

[5.6 Implementación de sistema de autenticación y autorización. 36](#_Toc115003949)

[Capítulo 6. Pruebas 36](#_Toc115003950)

[6.1 Introducción. 36](#_Toc115003951)

[6.2 Pruebas unitarias 36](#_Toc115003952)

[6.3 Pruebas de integración 36](#_Toc115003953)

[6.4 Pruebas de validación 36](#_Toc115003954)

[6.5 Conclusiones 36](#_Toc115003955)

[Capítulo 7. Cronograma y costes del proyecto 37](#_Toc115003956)

[Capítulo 8. Conclusiones y trabajos futuros 38](#_Toc115003957)

[8.1.1. Un aspecto fundamental en la ingeniería: La abstracción. 38](#_Toc115003958)

[Bibliografía 39](#_Toc115003959)

[Glosario 40](#_Toc115003960)

[Anexos 41](#_Toc115003961)

[I. Manual de la aplicación 41](#_Toc115003962)

[II. Datos origen para simulación y pruebas 41](#_Toc115003963)

# Índice de tablas

[Tabla 1 Entidad usuario 15](#_Toc113809709)

# Índice de figuras

# Introducción, objetivos y estructura

## Introducción

El objetivo inicial de este trabajo de fin de grado era el desarrollo de una aplicación web full stack con ingeniería de sistema basado en reglas aplicada en el dominio de negocio de la dietética y la nutrición, aunque a grosso modo a nivel práctico y tecnológico se ha implementado este objetivo, es verdad que debido a que el desarrollo de este trabajo ha sido paralelo a mi desarrollo en el mundo profesional, me he dado cuenta de que este trabajo de fin de grado puede cubrir una necesidad mucho más importante, la cual se expone en los siguiente párrafos.

Una de las dificultades con las que se puede encontrar un alumno del grado de ingeniería informática al salir de la facultad, es la falta de conocimiento o destreza para poder abordar el mundo profesional, es por ello que este trabajo de fin de grado es una aportación más en el intento de acercar más a los estudiantes o cualquier persona con inquietud en el mundo de la tecnología al mundo del desarrollo software del mundo laboral, con un ejemplo práctico de una aplicación web full stack durante sus diferentes fases, acompañada de comentarios de como se procesan estas fases en el mundo profesional.

Además, a nivel tecnológico este proyecto es muy rico dado que abarca todas las temáticas fundamentales del grado en ingeniería informática, como son:

* **Programación:** Desarrollo web con MEAN stack, es decir, MongoDB, Express, Angular y Nodejs.
* **Pruebas y testing:** Pruebas manuales de peticiones a back-end con POSTMAN
* **Redes:** Protocolos de comunicación entre lado cliente y lado servidor, con peticiones HTTP con verbos como GET, PATCH, POST, PUT, DELETE
* **Ingeniería de reglas:** Implementación práctica de un sistema basado en reglas.
* **Ciberseguridad:** Implementación de sistema de autorización y autenticación con Json Web Token.
* **Usabilidad y accesibilidad:** Experiencia de usuario en aplicaciones web.
* **Virtualización y despliegue**

Con todo lo anteriormente expuesto, tras leer e investigar este trabajo de fin de grado se enriquece al lector con una visión mucho más práctica no solo de como se desarrolla una aplicación sino de muchas implementaciones prácticas de los anteriores temas expuestos, por lo que sirve de complemento perfecto para interiorizar conceptos y entrar al mundo laboral con una visión mucho más completa de lo que se va a encontrar el individuo. A continuación, se exponen de manera más precisa los objetivos de este trabajo de este fin de grado.

## Objetivos

En este epígrafe se ha optado por dividir los objetivos en tres pilares fundamentales, siendo estos objetivos tecnológicos, del domino sobre el que se trabaja y de contribución a la comunidad científica.

### Objetivos tecnológicos.

Los objetivos tecnológicos son aquellos objetivos más vinculados a la tecnología y al grado en ingeniería informática, estos son:

* Desarrollo de un back-end con nodejs (JavaScript por el lado del servidor), express (framework de nodejs), con operaciones CRUD sobre una base de datos y búsquedas filtradas, así como operaciones sobre esta información.
* Estructuración y diseño de base de datos no relacional y definición de arquitectura de la información.
* Implementación de un sistema basado en reglas sencillo que permita hacer filtrado de datos en la información que se recupera de la base de datos no relacional.
* Aplicación de sistema de autenticación y autorización basado en Json web token que permita al usuario mantener sesiones seguras en la aplicación.
* Investigación y puesta en práctica de buenos patrones arquitectónicos tanto en front como en back, basándose en las necesidades del proyecto.
* Desarrollo de un front-end (lado del cliente) con Angular(framework de javascript) con una interfaz amigable, fluida y que permita al usuario la consecución de sus objetivos.
* Aplicación del patrón FLUX en el front-end.

### Objetivos del dominio sobre el que se trabaja.

* Aportar a personas con pocos recursos una herramienta sencilla y gratuita para poder conseguir sus objetivos nutricionales.
* Asesoramiento para llevar una alimentación saludable.
* Gestionar cuantas calorías diarias has consumido y cuantas te quedan para lograr tu objetivo.
* Poder añadir, eliminar, modificar y consultar de manera personalizada alimentos y organizarlos por secciones para posteriormente añadirlos fácilmente a tu dieta.

### Objetivos de contribución a la comunidad científica.

* Ofrecer un repositorio público con un ejemplo práctico de aplicación full stack que cualquier individuo pueda utilizar para documentarse.
* Proporcionar un proyecto con buenas prácticas implementadas que sirva como guía para futuro desarrolladores sin experiencia en el mundo laboral.
* Desarrollo de un proyecto lo más cercano al mundo laboral real para que cualquier sujeto que estudie este trabajo sea consciente de la realidad laboral en el mundo del desarrollo software.
* Aportación de un análisis de las bases de datos relacionales en contraste con las bases de datos no relacionales.

## Estructura del documento

El capítulo 2 se ceñirá a analizar que es una aplicación web full stack, ventajas y desventajas y estado actual en el mercado laboral del desarrollo de aplicaciones web, así como una breve exposición de hacia dónde se está dirigiendo ahora mismo el desarrollo de aplicaciones web en el mundo laboral.

Los capítulos del proyecto del 3 al 7 representan cada uno los diferentes ciclos de desarrollo de una aplicación (análisis, diseño, implementación, pruebas) y uno adicional que habla de la integración del proyecto en entornos reales.

Además, tanto la fase de diseño como de implementación serán paralelas en estructura, pero diferentes en cuanto a nivel de especificidad

# Estado del arte

## Introducción.

Este apartado podría enfocarse desde dos perspectivas:

* La primera, análisis del estado de las aplicaciones de dietética y nutrición en el mercado, ver sus deficiencias y sus virtudes y como en este trabajo de fin de grado se potencian sus virtudes y se reducen sus deficiencias.
* La segunda, la cual, será la que tomaremos en este epígrafe, consiste en un análisis del estado actual, a nivel tecnológico, del desarrollo de aplicaciones web full stack, tanto a nivel tecnológico como en el mundo profesional real, dado que el objeto de este TFG es demostrar conocimientos en el ámbito tecnológico.

## ¿Qué es una aplicación web full stack?

Una aplicación web full stack, es una aplicación cuyo funcionamiento y desarrollo global esta extendido por un lado denominado front-end (navegadores webs) y por otro lado denominado back-end (lado del servidor).

Estos dos lados están estrechamente bien comunicados para que la aplicación funcione en su conjunto, por lo tanto, una aplicación web no es ni una aplicación nativa de escritorio o servidor, ni una aplicación nativa de dispositivos móviles (Android, IOS..etc.).

Estas aplicaciones normalmente desde el lado del cliente se ejecutan mediante un navegador web, lo cual hace que la aplicación se abstraiga del sistema operativo nativo o de la maquina que ejecuta la aplicación, dado que es el navegador el intermediario entre sistema operativo o maquina y la aplicación web.

## Ventajas de las aplicaciones web full stack

* El hecho de que la aplicación web se abstraiga del sistema operativo y del dispositivo hardware especifico hace que sea multiplataforma sin mucho esfuerzo.
* El lenguaje front y back no cambia entre dispositivos, no hay que diseñar con diferentes lenguajes y tecnologías una misma aplicación para IOS, otra para Android y otra para escritorio, todas funcionan.
* Su potencial para ser fácilmente configurable en cuanto a diseño para cualquier tipo de pantalla.
* Comunidad amplia: sin lugar a duda, una de las características determinantes del éxito de una aplicación es como de participativa sea en ella la comunidad científica de computadores, en este caso, javascript tiene una muy fuerte comunidad científica, prueba de ello es que hay muchísimos frameworks, librerías…etc. desarrollados por la comunidad.

## Desventajas de las aplicaciones web full stack.

* La aplicación web no puede exprimir al máximo las funcionalidades del sistema operativo, dado que depende del navegador, por ejemplo, si los nuevos dispositivos aportan un lector de huella dactilar para una identificación biométrica y, por ende, fase de autorización más robusta, hasta que el navegador no aporta mediante javascript librerías o funciones para utilizar esa funcionalidad, la aplicación web no puede explotar esta funcionalidad.
* Muchas veces, los navegadores guardan automáticamente en memoria caché virtual los archivos que reciben de servidor, con el fin de que cuando se vuelvan a solicitar esos archivos se vayan a buscar a memoria y no a servidor, por lo que, si, por ejemplo, el usuario tiene almacenado el software en un archivo .js y en lugar de pedirlo en servidor, recoge uno que esta almacenado en memoria, es probable, que el servidor.
* La organización de desarrollo es más compleja que la de una aplicación nativa o de escritorio, dado que normalmente esta compuesta por dos equipos que trabajan en diferentes stacks, por un lado, el back-end y por otro lado el front-end, la organización de personas y como se comunican los dos mundos es mucho más critica que en otros tipos de proyecto software.

## Tendencias actuales de las aplicaciones web full-stack en el mundo profesional.

Una de las tendencias mas notorias es el desarrollo de microservicios tanto en el lado front como en el lado back, es la fragmentación de un proyecto en sub-proyectos más pequeños y mantenibles, con esto el acoplamiento del sistema disminuye, dado que cada sub-proyecto debe ser independiente del entorno que le rodea, actuando como una unidad única e indivisble.

Además, cada sub-proyecto puede estar compuesto de diferentes tecnologías, por ejemplo, un micro-proyecto puede estar compuesto de Python con MySql y otro micro-proyecto puede estar conformado por Nodejs y MongoDB, esto permite hacer un proyecto grande que no este limitado a una determinada tecnología, sino que este compuesta por varias, y esto implica que en cada una de las áreas de una aplicación dependiendo de las necesidades que se tengan se usara la tecnología más adecuada para satisfacer esa necesidad.

Por ejemplo, si tenemos una aplicación que consiste en una red social donde los usuarios pueden subir, editar…etc. imágenes y además chatear con otros usuarios en tiempo real, tenemos dos áreas fundamentales identificadas, procesamiento de imágenes y chat en vivo, en este caso para el procesamiento de imágenes es más adecuada Python con su framework Django, y en el caso del chat en vivo podemos hablar de Nodejs con MongoDB es lo más oportuno, permitiendo explotar las ventajas de cada stack tecnológico según su caso de uso.

Además, lo micro-servicios hacen que los proyectos no crezcan de manera desorbitada y permitan al equipo software mantener la aplicación, si se quieren añadir funcionalidades, arreglar un fallo o hacer mantenimiento, probablemente sea mucho más fácil que un proyecto mucho más grande en un único sitio.

## Problemas detectados en el desarrollo de aplicaciones web y soluciones aportadas en este trabajo de fin de grado.

Este apartado se centrará en mostrar carencias detectadas en el desarrollo de aplicaciones web junto con la solución aportada en el desarrollo de este TFG.

### Alto acoplamiento entre lado del cliente y servidor con patrones clásicos de MVC en el lado del servidor.

#### Explicación del problema.

El patrón MVC (Modelo-vista-controlador), este patrón se basa en recibir una petición del lado del cliente, procesar esta petición mediante un controlador, en caso de que sea necesario, el controlador invoca datos a un modelo (base de datos) y al recibirlos incrusta estos datos en una plantilla html, css y js en el servidor, tras esto, devuelve un archivo html al cliente, con sus vínculos css y js y el cliente los va demandando uno a uno. Esto implica un mínimo de 3 llamadas, uno para el archivo Html, otra para el css y otra para el js, pero realmente el número de llamadas es mucho más grande en proyectos reales.

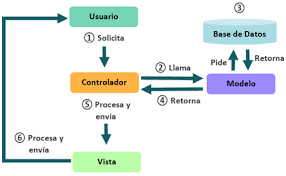


Ilustración 1 Patrón MVC(Modelo-vista-controlador)

Este enfoque tiene varios inconvenientes, el primero es que el lado del cliente depende demasiado del lado del servidor, por cada vista que queremos ver o cada cambio de pantalla hay que hacer peticiones al servidor, esperar a que procese la petición y recibir respuesta, la cual suele derivar en un mínimo de 3 llamadas (en proyectos muy pequeños) para obtener archivos html, css y js. Esto con una persona está bien, pero cuando hay horas de alta demanda de una web, el servidor puede estar procesando miles de llamadas, lo cual, deriva en un rendimiento mucho más bajo y una saturación de peticiones que se puede solventar fácilmente.

Además, si el servidor sirve paginas HTML, CSS y JS esto deriva en que no se podrá utilizar en una aplicación Android, IOS o de escritorio la información aportada por el servidor, dado que estos no se basan en tecnologías web.

#### Solución aportada en este TFG

Una posible solución es hacer SPA (Single page aplication) las cuales consisten en que el front y el back están separados, en este caso el MVC se fragmenta en varias partes, dependiendo del proyecto. En el caso de este trabajo de fin de grado el front-end tiene un modelo controlador-vista. Y el back-end tiene un modelo controlador-modelo.

En este caso, el lado del servidor si puede ser consumido por cualquier aplicación, dado que el servidor se limita a servir información en documento plano de tipo JSON y es el propio lado del cliente el que procesa esa información como le sea necesario, haciendo posible que se puedan utilizar los servicios back-end en cualquier tipo de aplicación sea IOS, Android o de escritorio.

Además, el número de peticiones se reduce considerablemente en la red, esto se debe a que del lado del cliente se descarga una SPA, que tiene su propia lógica, enrutamiento, controladores, servicios…etc. siendo una aplicación con cierta inteligencia y no limitándose a renderizar html, css y javascript. Ahora, cada vez que el usuario cambia de página o de vista, no es necesario hacer peticiones al back-end, esto no tiene solo un efecto local en el proyecto, sino que afecta globalmente también al sistema de internet, cuantas menos peticiones o por lo menos, menos peticiones densas, menos sobrecargadas estarán las redes y más fluida ira la red global de Internet.

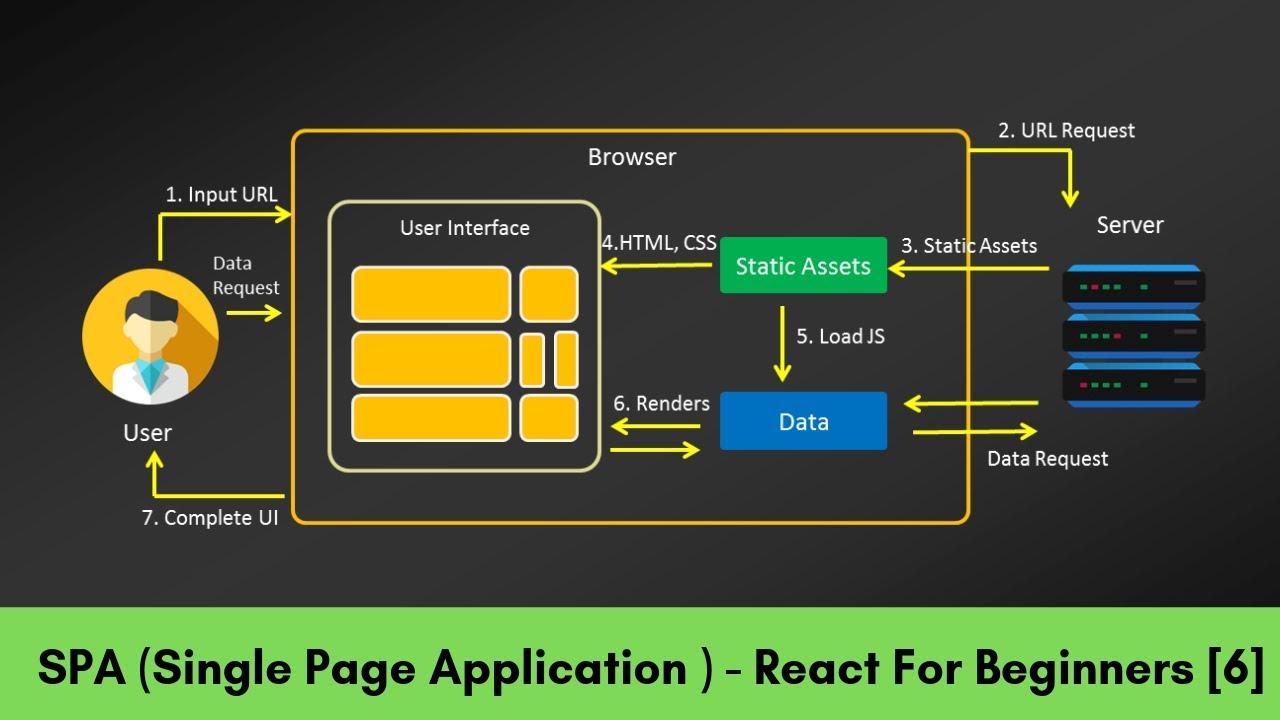


Ilustración 2 Enfoque SPA (Single page application)

### Mala definición de arquitecturas de proyectos software.

#### Definición del problema.

Uno de los mayores inconvenientes en el desarrollo de cualquier proyecto, sea cual sea la tecnología o el proyecto software, es que **la definición inicial de la arquitectura del proyecto no suele ser la adecuada**, este punto es el más determinante de la aplicación, porque su definición determina el potencial del proyecto para ser extensible, mantenible, testeable, migrable a otras tecnologías ...etc. Podríamos afirmar que hay una relación directamente proporcional entre el esfuerzo de desarrollo inicial de una arquitectura y la calidad de la arquitectura en si misma para poder ser extensible, mantenible…etc. Aunque también es cierto que este esfuerzo inicial se verá compensado en el futuro haciendo mucho más fácil el desarrollo de nuevas funcionalidades, el testing de ciertas áreas de la aplicación, el mantenimiento…etc.

#### Solución aportada en este TFG.

En el caso del back-end se ha optado por no hacer sobre ingeniería dado que el número de entidades es pequeño y la complejidad del lado del servidor no es alta, por ello, no se ha optado por usar una “Arquitectura limpia” que lo que busca es que un proyecto sea independiente de la tecnología que usa, es decir, que no dependa de la base de datos concreta que se use o de los frameworks que se utilicen, o una arquitectura orientada a microservicios, que lo que busca es fragmentar la aplicación en diferentes sub-proyectos, independientes, con sus propias pruebas y testing, con sus dependencias a otras librerías o tecnologías e incluso con sus propio stack tecnológico, de esta manera, se puede crear un ecosistema más complejo, en el que hay varios microservicios de diferentes tecnologías que conviven entre si para lograr un objetivo común.

Por todo esto, en el lado back-end se ha optado por un modelo clásico de modelo-controlador, en la que la arquitectura se basa en la recepción de peticiones del lado del cliente en el servidor, procesamiento de estas por un controlador y en caso de ser necesario, peticiones a un modelo que sería la base de datos.

En el caso del front-end, si que se ha hecho un esfuerzo por una arquitectura mucho más definida que permite hacer la aplicación más extensible y mantenible en el futuro, lo primero que se ha hecho es la fragmentación de la aplicación en módulos, según el área de negocio en el que se mueve, con esto mientras el usuario va navegando por el área de un módulo, en el caso de moverse hacia otra área que implique otro modulo, ese nuevo módulo puedo descargarse de servidor, de esta manera, en el momento inicial de carga de la aplicación, no se cargan todos los módulos de la aplicación, si no, solo aquellos imprescindible para el funcionamiento de la aplicación en el área inicial, si el usuario se mueve a nuevas áreas, entonces, se solicitarán módulos asociados a esas áreas, esto se conoce como “lazy-load” o “carga perezosa”. Otras medidas son, la creación de un modulo denominado “Shared” que tiene todos los componentes transversales a toda la aplicación, además, con el lenguaje TypeScript, el cual, es un super-conjunto de javascript, se permite hacer un tipado más fuerte, con la posibilidad de creación de interfaces, enumerados, modelos, clases más tipadas…etc. Lo cual, deriva en una arquitectura más organizada y limpia.

También se ha implementado el patrón FLUX, el cual se expone en el siguiente apartado.

### Componentes front-end altamente acoplados con peticiones al lado del clientes o fragmentación de los estados de una aplicación.

#### Definición del problema.

Tal y como se expuso en el anterior párrafo uno de los inconvenientes es el fuerte acoplamiento entre peticiones al lado del servidor y los componentes de la aplicación en el lado del cliente. Otro problema es que los componentes alberguen su propio estado, los cuales, son sub-estados de la aplicación global. Por ejemplo, si un componente se encarga de mostrar un listado de alimentos, esto implica que el propio componente tenga que almacenar el listado de alimentos para mostrarlo y además para obtenerlo es necesario que el componente llame a servidor, dotándoles de cierta “inteligencia”.

Esto se debe evitar, dado que luego las depuraciones se hacen mucho más difíciles, por ejemplo, si hay algún error, se debe buscar a nivel de componente ¿Cuál fue de los muchos componentes existentes, el que disparo la llamada a obtener los alimentos, si tengo más de 20 componentes que llaman a ese “obtener alimentos”?

Además, hay un riesgo potencial de duplicación de información o de riesgo de integridad de datos, dado que, si dos componentes tienen sus propias copias de información, si alguno la altera, el otro componente no se cerciorará de ello y tendrá información incorrecta.

#### Solución al problema.

Una manera de solucionar el problema es con el patrón FLUX, que consiste en la centralización del estado de la aplicación en un modulo determinado de la aplicación, haciendo más mantenible y depurable la información, dado que este patrón normalmente lleva un registro de quien ha disparado ese servicio…etc. Una visión más específica del patrón flux, nos la puede mostrar una de sus implementaciones, en este caso tenemos NgRx que es una librería que implementa este patrón, su arquitectura queda como sigue:

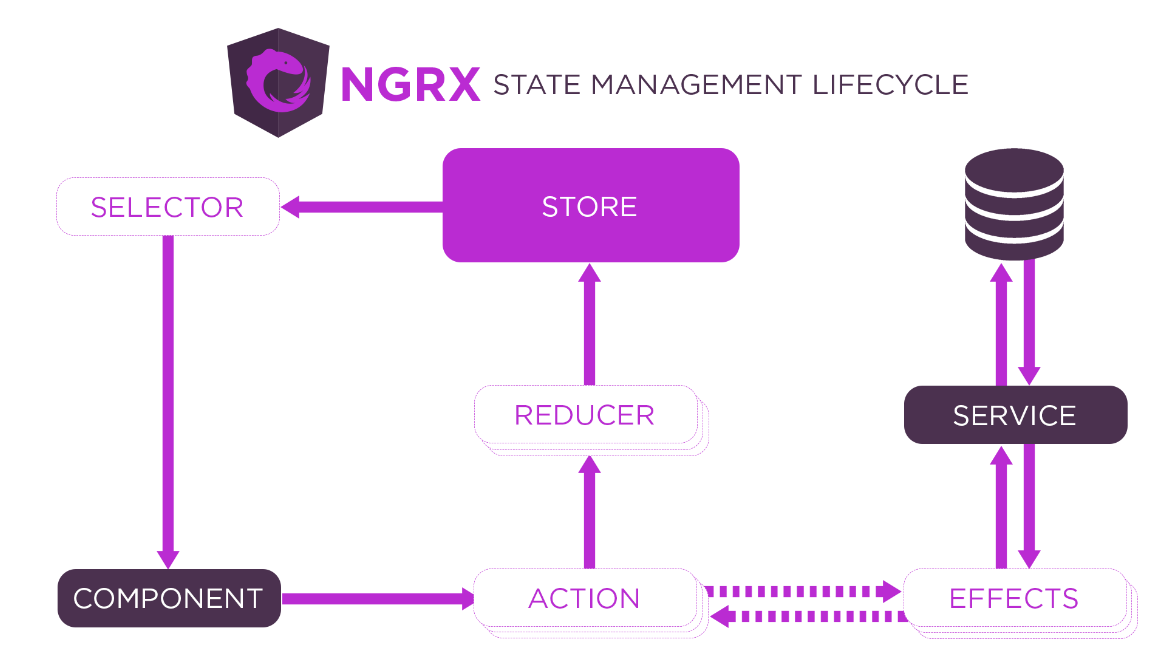


Ilustración 3 ARQUITECTURA FLUX CON NGRX

En este caso tenemos varios elementos:

* **Componente:** Es una unidad fundamental que tiene su propio controlador, estilos y código html asociado, podemos decir, de manera burda, que es un fragmento de la vista de una aplicación.
* **Store:** Lugar centralizado que almacena todos los sub-estados de la aplicación. Por ejemplo, en este lugar, se pueden almacenar el listado de alimentos, variables indicativas de si hay algo “esperando” a ser cargado de un servidor, datos del usuario logueado…etc.
* **Action:** Como dice su nombre, se tratan de acciones a aplicar sobre el “Store” por ejemplo, añade un alimento a la lista de alimentos, borra datos de usuarios” …etc.
* **Reducer:** Es la asociación entre una acción y la actualización del estado, por cada acción, el reducer hace transitar el “store” de un estado hacia otra, con esto se consigue tener una especie de registro de como se va transitando de un estado a otro según qué acciones se disparen.
* **Efectos:** Esto es una herramienta un poco más avanzada y se ha agregado recientemente, en este caso, son intermediarios entre las acciones y las peticiones a servidor mediante servicios. Con esto, lo que se consigue es que cuando se dispare una acción “agregar alimento” tenga como efecto, llamar al servidor para insertar alimentos y una vez se inserte, volver a llamar a servidor para obtener la lista de alimentos actualizadas, tras esto, se dispara una acción que dice, “actualizar alimentos” y así el almacén central se actualizar.
* **Selectores:** Los componentes “seleccionan” el sub-conjunto del estado centralizado que les interesa y lo hacen mediante “suscripciones” que permiten escuchar a cualquier cambio o incidencia que acontezca.

Lo más importante para que esta arquitectura se vea de manera pragmática a nivel “vista” es el uso de suscripciones sobre el estado central, es decir, cada componente se suscribe a las partes que les convenga del estado central, escuchando a cualquier cambio que se produzca y si un componente altera la información del estado centralizado, que deriva en un efecto colateral para otros componentes, al estar estos componentes “suscritos” a cambios en la información, se soluciona de manera inmediata el problema de duplicación del a información.

### Grandes anidaciones de sentencias condicionales

#### Definición del problema.

Podemos afirmar que cuando tenemos una serie de hechos que juntos terminan llegando a una hipótesis o una conclusión final, podemos abordar este problema de varias maneras, una es introduciendo en clausulas condicionales cada uno de los atributos que definen los hechos e ir redirigiendo el flujo según los valores, sin embargo, en situaciones donde hay muchos hechos y una conclusión o hipótesis depende del estado de cada uno de ellos, se puede llegar a construir una jerarquía de condiciones complejas y esto hace que el código sea, a largo plazo, más difícil de mantener o de añadir nuevas funcionalidades.

#### Solución al problema: Sistema basado en reglas.

Cuando tenemos un problema de tipo “un conjunto de hechos implica una cierta hipótesis” podemos afirmar que un sistema basado en reglas es idóneo para la solución de este problema. Según la Wikipedia podemos decir que un sistema basado en reglas tiene la siguiente definición:

*Los****sistemas basados en reglas****trabajan mediante la aplicación de reglas, comparación de resultados y aplicación de las nuevas reglas basadas en situación modificada para aprobar o desaprobar una determinada hipótesis.*

Es decir, de manera más coloquial, un sistema basado en reglas, es un conjunto de reglas de tipo “x implica y” que al juntarse todas e instanciarse con unos hechos derivan en una hipótesis o conclusión.

La mejorar manera de interiorizar este concepto es con la exposición de una parte más pragmática. Por ejemplo, en el dominio de este trabajo de fin de grado tenemos el mundo de la nutrición y dietética y podemos afirmar lo siguiente:

* Regla 1: Si el usuario es vegetariano se elimina la carne, el pescado y el marisco de su dieta y se buscan fuentes de proteínas en la Soja.
* Regla 2: Si, además, el usuario es vegetariano y alérgico a la soja, podemos afirmar que al usuario no se le recomendará la soja.

Una vez tenemos las reglas en nuestro sistema basado en reglas podemos inyectar en este sistema unos hechos:

* Hecho 1: El usuario es vegetariano.
* Hecho 2: El usuario no es alérgico a la soja.

Una vez insertamos estos hechos en nuestro sistema basado en reglas, podemos afirmar lo siguiente:

* Hipótesis: El usuario no debe comer carne, pescado o marisco, y se debe recomendar soja como fuente de proteína.

Mientras que, con una estrategia de programación imperativa, para resolver este problema tan sencillo, tendríamos una jerarquía de sentencias condicionales (if-else) anidadas que con el paso del tiempo y la aplicación de nuevas reglas se haría compleja, difícilmente extensible y mantenible, en una ingeniería de reglas, la reglas estaría separadas y el propio sistema basado en reglas juntaría todas las reglas y sacaría conclusiones con ellas. Permitiendo tener un código más organizado, extensible y mantenible.

Además, no solo se trata de organización de código, simplicidad, extensibilidad y mantenimiento, sino que, además, los sistemas basados en reglas, al haber sido diseñados para funcionar bajo una estrategia de programación más orientada hacia la lógica, serán más eficientes y tendrán un mayor rendimiento en la resolución de las reglas para llegar a unos hechos.

### Tabla resumen de inconvenientes y soluciones.

|  |  |  |
| --- | --- | --- |
| HECHO | INCONVENIENTES | SOLUCIÓN EN ESTE TFG |
| Hacer una petición por cada vista de la aplicación web. | * Sobrecarga de esfuerzo de procesamiento de vista * Muchas peticiones * Contribución a un internet más saturado. | Implementación de SPA (Single page application) |
| No universalidad del  servidor | * El lado del servidor no puede ser reutilizable en tecnologías diferentes de las asociadas al mundo web | Servidor REST que se limite a proveer/recibir indatos en formato JSON delegando más responsabilidad en el lado front-end |
| Mala implementación de arquitecturas de proyectos software | * Implementación de arquitecturas sin tener una perspectiva de futuro que permita hacer la aplicación más mantenible, extensible, depurable ..etc.. | No hacer sobre ingeniería en el lado del servidor implementando un patrón Modelo-controlador.  Modularizar lado front-end y aplicación del patrón flux |
| Componentes front-end altamente acoplados con peticiones a servidor | * Los componentes llaman directamente a los servicios, se dos o más componentes llaman al mismo servicio y hay algún problema con la llamada a ese servicio, es más, difícil detectar que componente ha llamado a ese servicio | Implementación del patrón FLUX. |
| Fragmentación del estado global de una aplicación en los propios componentes | * Duplicación de la misma información de diferentes componentes. * Riesgo de integridad de la información al existir varios lugares donde existe la misma información y, además, se puede alterar. | Implementación del patrón FLUX |
| Anidaciones complejas de sentencias condicionales | * La modelación de un sistema de conocimiento experto con sentencias condicionales puede ser difícil de mantener, extender o de que tenga un buen rendimiento. | Sistemas basados en reglas. |

Ilustración 4 Contraposición entre problemas actuales en el desarrollo de aplicaciones y soluciones aportadas en este trabajo de fin de grado

# Análisis

## Modelo de dominio

El modelo de dominio sobre el que está enfocado la aplicación es el de nutrición y dietética, este dominio se caracteriza por ser bastante parametrizable. En este caso lo que se busca es que el usuario pueda hacer un seguimiento de su dieta y recibir algún tipo de asesoramiento por parte de la aplicación para poder conseguir sus objetivos nutricionales (subir de peso, bajar de peso, mantener peso, ganar masa muscular o definir).

Además, dado que el foco del TFG es la ingeniería de software y no el mundo de la nutrición el modelo de dominio es un subconjunto muy pequeño del amplio mundo de la nutrición y dietetica, quedando, a grandes rasgos, como sigue:

* Un alimento está compuesto de kilo calorías que definen **su aporte energético** y macronutrientes.
* Una persona tiene unas características fisiológicas (altura, peso, edad…etc.) que definen la cantidad de **aporte energético diario** que necesita durante un día.
* Además, un usuario puede tener cierto objetivo que incrementa o decrementa en un determinado porcentaje la cantidad de calorías diarias a consumir, es decir, el aporte energético diario necesario.
* Con todo esto, el objetivo de la aplicación es ofrecer los alimentos que debe consumir el usuario diariamente para satisfacer sus necesidades energéticas según su objetivo y proporcionar datos sobre su progreso.

A partir de esta base se pueden sacar muchas funcionalidades, que van extendiendo el modelo de dominio que se va abarcando, sin embargo, no es objeto de este trabajo de fin de grado desviarse del foco de este, que es el mundo tecnológico

## Análisis de requisitos

El primer requisito que se capta es la necesidad de crear, leer, modificar o eliminar alimentos, dado que la unidad básica fundamental de la aplicación son los alimentos. Además, que el usuario pueda agrupar alimentos por secciones en una especie de almacén personal y privado hace que este pueda organizarse mejor.

Ahora bien, muchos usuarios no querrán meter manualmente un alimento, si no que quieren añadir un alimento ya creado poniendo el nombre, para ello, es necesario hacer consultas filtradas, por lo que de todo esto se deduce que es necesario tanto un CRUD de secciones de alimentos como búsquedas filtradas sobre los alimentos para añadirlos a estas secciones.

Además, es necesario tener información del usuario como altura, peso, edad, actividad física diaria, genero, que tipo de alimentación lleva (vegana, vegetariana, omnívora) y cual es su objetivo. Con esta información se pueden calcular otros datos como la cantidad de kilo calorías diarias que necesita, cantidad de agua necesaria al día, índice de masa corporal, distribución de macronutrientes diarios…etc.

También hará falta que la aplicación recomiende alimentos, para ello, con los datos obtenidos del usuario debe ser capaz de discernir que alimentos son adecuados para el usuario. Otros requisitos son que la información se persista si el usuario sale de la aplicación, por lo que los datos se deben asociar a ese usuario y además hace falta algo de autenticación y autorización para que un usuario tenga protegidos sus datos sensibles. Podemos concluir que la captación de requisitos en la etapa inicial queda como sigue:

* CRUD de alimentos.
* CRUD de secciones de alimentos.
* CRUD de datos de usuario.
* Gestor de usuarios con sistema de autenticación y autorización.
* Generar recomendaciones de alimentos basado en ingeniería de reglas.

En el mundo real, es muy difícil captar todos los requisitos funcionales de una aplicación en el momento inicial, es por ello, que se exponen a continuación los requisitos funcionales detectados en fases como la de diseño, implementación y testing. Cabiendo señalar, que en el caso hipotético de que un cliente nos hubiera mandado a hacer esta aplicación, normalmente, durante el propio desarrollo se le ocurren nuevos requisitos, corrige anteriores requisitos especificados e incluso elimina algunos, por lo que conviene de primera mano tener una muy buena comunicación con el cliente.

Requisitos funcionales captados durante fases posteriores del desarrollo software:

* **Requisito:** Necesidad de que el usuario vea como ha sido su progreso en el día de hoy ¿Cuántas calorías ha consumido en contraposición de las que tiene que consumir? ¿Qué alimentos les queda por consumir?

**Implicaciones:** Es necesario crea una nueva entidad de tipo “registro diario de consumo” que tenga una fecha y que alimentos se han consumido para cada tipo de comida. Además, se debe añadir a los datos del usuario campos calculados que devuelvan cuantos macronutrientes se deben consumir en reposo o haciendo ejercicio.

# Diseño

## Introducción

En este capítulo se bajará un escalón más en la capa de abstracción, enfocándose más en los detalles del desarrollo de la aplicación, para ello, se abordará una visión desde el lado del servidor hacia el lado del cliente, siguiendo el siguiente esquema:

* Diseño de base de datos: Modelado de la información.
* Diseño de sistema basado en reglas.
* Diseño del servidor: Arquitectura REST
* Contratos entre servidor-cliente: compromiso de comunicación entre servidor-cliente
* Diseño del fron-end: Arquitectura front-end

## Diseño de base de datos.

Sin lugar a duda, uno de los aspectos más importantes en el desarrollo de cualquier aplicación sobre un determinado modelo de dominio es como se diseña la arquitectura de datos, de tal forma que el resultado de este diseño puede ser determinante en el rendimiento de la base de datos y en como de eficiente es el acoplamiento entre el controlador software que dirige el comportamiento lógico de la aplicación con la manera que está estructurada la información.

Subrayar que se resaltan en verde aquellas filas que tienen que ver con el desarrollo de la ingeniería basada en reglas.

### Entidades de la aplicación.

#### Usuario.

La entidad usuario se encarga de almacenar información de un usuario para su autenticación y autorización de acceso a determinados recursos.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Descripción** | **Tipo de dato** |
| Nombre | Nombre del usuario | Cadena de texto |
| Email | Email del usuario. Es un atributo que no admite duplicados | Cadena de texto |
| Password | Contraseña | Cadena de texto |
| PasswordConfirm | Contraseña repetida (Para doble validación) | Cadena de texto |
| rol | Rol del usuario (para control de permisos en la aplicación) | Cadena de texto |
| Último cambio del password | Última vez que cambio el password, para añadir una capa de seguridad en el control mediante Json Web Token. | Fecha |

Tabla 1 Entidad usuario

#### Datos de usuario.

En este caso, por cada usuario tenemos unos datos fisiológicos que permiten al sistema inteligente ser más preciso con las recomendaciones que hace al usuario a nivel dietético y nutricional.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Descripción** | **Tipo de dato** |
| Objetivo | “Ganar peso”, “Perder peso”, “Aumentar músculo” …etc. | Cadena de texto |
| Edad | Edad del usuario | Número |
| Peso | Peso del usuario | Número |
| Altura | Altura del usuario | Número |
| Tipo de alimentación | Dieta omnívora, vegetariano, vegano…etc. | Cadena de texto |
| Alimentos prohibidos | Alimentos que el usuario ha especificado de manera explícita que no desea consumir | Array de ids de alimentos |
| Actividad física | Sedentario, Moderada, intensa…etc. | Número |
| Alergenos | Tipos de alergenos a los que es alérgico el usuario: Huevo, gluten, soja, lactosa, pescado, marisco..etc. | Array de cadenas de texto |
| **Resultado de ingeniería de reglas** | Información que optimiza y precisa mejor el asesoramiento nutricional al usuario. | Objeto complejo de booleanos |
| Identificador usuario | Identificador de la entidad usuario asociado. |  |
| Foto | Url de la foto en el servidor |  |
| Campos calculados | | |

Tabla 2 Entidad Datos usuario

Con respecto al último campo, cuando los datos de un usuario se introducen en el Sistema basado en regla, esta nos devuelve información que filtra el tipo de recomendaciones que se hacen al usuario a nivel nutricional. Dado que la altura, la edad, peso…etc. no cambian frecuentemente en el tiempo es recomendable guarda resultado de la ingeniería de reglas en lugar de ejecutarla constantemente cada vez que se haga una consulta

#### Alimento.

Un alimento es una de las unidades fundamentales del dominio de negocio, dado que estamos en el ámbito de la nutrición.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Descripción** | **Tipo de dato** |
| Nombre | Nombre del alimento | Cadena de texto |
| Descripción | Descripción del alimento | Cadena de texto |
| Peso | Peso de una unidad | Número |
| Calorías | Aportación energética alimento | Número |
| Hidratos | Hidratos del alimento por 100gr | Número |
| Proteínas | Proteínas del alimento por 100gr | Número |
| Grasas | Grasas del alimento por 100gr | Número |
| Etiquetas | Etiquetas que definen el alimento: [‘Carne’, ‘Vegano’, ‘Sin gluten...etc.]. Estas etiquetas son fundamentales para la ingeniería basada en reglas | Array de Cadena de texto |
| Para que comida es recomendable | Desayuno, Almuerzo, cena, genérico. | Array de Cadena de texto |
| Imagen fuente | url en la que se almacena la imagen | Cadena de texto |
| Fecha de creación | Fecha de creación del alimento | Fecha |

Tabla 3 Entidad alimento

#### Secciones de alimentos.

El usuario podrá crear sus propias agrupaciones de alimentos de manera personalizada, esto le permitirá buscar alimentos propios de manera más eficiente en incluso cargar en ciertas comidas estas secciones directamente, siendo más prioritarias que las recomendaciones de alimentos del sistema inteligente.

|  |  |  |
| --- | --- | --- |
| **Campo** | **Descripción** | **Tipo de dato** |
| Nombre | Nombre de la sección | Cadena de texto |
| Descripción | Descripción de la sección | Cadena de texto |
| Imagen | Imagen representativa de la sección |  |
| Alimentos | Alimentos que constituyen la sección | Array de identificadores de entidades de tipo alimento |

Tabla 4 Entidad sección de alimentos

#### Registro diario de comidas.

Algo muy intrínseco de una aplicación de nutrición es el factor tiempo, dado que no todos los días comemos lo mismo, es de vital importancia que exista un registro de las comidas que se consumen a lo largo del tiempo

|  |  |  |
| --- | --- | --- |
| **Campo** | **Descripción** | **Tipo de dato** |
| Fecha del registro | Día en el que se efectuó la comida | Fecha |
| Comidas y alimentos | Array de objetos complejos, los cuales, están compuestos de un nombre de comida (Desayuno, Almuerzo…etc) y un array de entidades de tipo alimento | Array de objetos complejo |
| Identificador usuario | Usuario asociado | Identificador |

Tabla 5 Entidad registro diario comidas

### Relaciones entre entidades.

Una vez definidas las entidades del dominio se procede a establecer relaciones entre ellas, en este caso, se pasará de una visión pequeña a una visión más grande del conjunto de relaciones de toda la aplicación.

Además, cabe señalar que el diseño no se contempla la especificación de la base de datos elegida, que en este caso es una base de datos No relacional, se verá cómo influye este hecho en la fase de implementación.

#### Relación Usuario – Datos de Usuario.

En este caso se trata de una relación 1:1, es decir, por cada entidad de tipo “usuario” existirá una entidad de tipo “Datos de usuario” y lo mismo a la inversa, por cada entidad “Datos de usuario” existirá un único “Usuario” asociado. Aunque no es nada común que una relación 1:1 se desacople de una misma entidad transformándolo en dos entidades, en este caso es así, porque en el mundo del software es buena praxis separar la parte de autenticación de usuarios con la de los datos propios del usuario en el dominio de negocio. Es más, normalmente se suele separar en otra entidad los roles del usuario y las contraseñas, nunca yendo en la misma entidad usuario-contraseñas-rol, sin embargo, no se ha hecho en este caso así, dado que la aplicación está más enfocada en un sistema basado en reglas que en ciberseguridad.

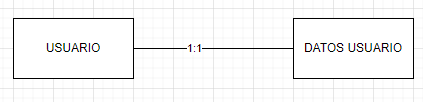


Ilustración 5 Relación Usuario-Datos Usuario

#### Relación Usuario – Sección de alimentos.

En este caso un usuario puede tener varias secciones de alimentos y una sección solo puede tener un usuario, es decir, se trata de una relación 1 a muchos. Acoplándolo a nuestro sistema anterior queda como sigue:

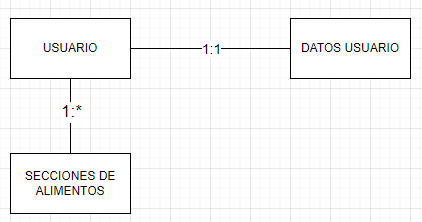


Ilustración 6 Relación usuario - Sección alimentos

#### Relación Sección de alimentos – Alimentos.

Está relación es de muchos a muchos, dado que una sección de alimentos puede tener varios alimentos y un alimento puede estar en varias secciones de alimentos. Esto implica un doble entrelazamiento, tanto la sección de alimentos apunta a varias entidades de Alimentos como un Alimento apunta a varias secciones.

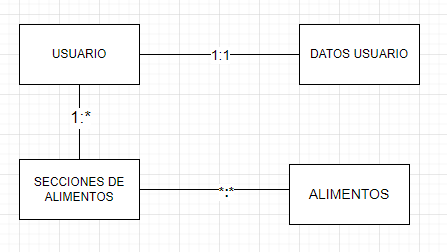


Ilustración 7 Relación sección de alimentos – Alimentos

#### Relación Usuario – Registro histórico diario de comidas.

En este caso se trata de una relación de 1 a muchos, un usuario puede tener varios registros históricos de comida diaria pero un registro solo puede tener un usuario asociado.

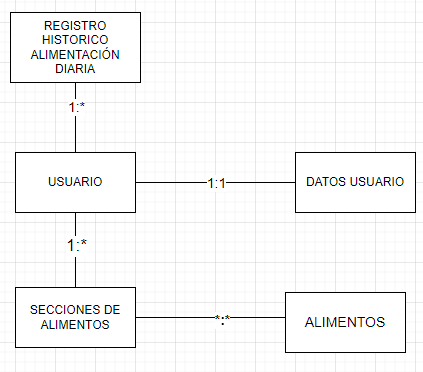


Ilustración 8 Relación usuario - Registro diario alimentación

## Diseño de sistema basado en reglas.

En esta aplicación se ha optado por utilizar una ingeniería de reglas sencilla, su función principal es proporcionar un subconjunto de alimentos más adecuado para el usuario según sus necesidades nutricionales, alergias y tipo de alimentación. En este epígrafe se definirán las reglas de manera superficial dado que representa la fase de diseño, y en el apartado [5.5 Implementación de sistema basado en reglas](#_Implementación_de_sistema) podremos ver a un nivel mucho más específico como se aborda este caso.

Tal y como se argumentaba en el anterior párrafo la ingeniería de reglas se limita a ayudar en el filtrado de alimentos y lo primero que hay que hacer antes de su implementación es diseñarla, para ello debemos definir una serie de reglas, las cuales se exponen a continuación.

* Si un usuario es vegano entonces no debe mostrarse ningún alimento de origen animal.
* Si un usuario es vegetariano, no debe mostrarse ningún alimento que sea carne, pescado, marisco o crustáceos.
* Si un usuario tiene algún tipo de alergia deben filtrarse los alimentos que estén asociados a esa alergia, por ejemplo, para celiacos no deben cargarse alimentos de tipo cereal o con gluten.
* El usuario podrá prohibir que se le muestren algunos alimentos, es por ello, que deben filtrase estos alimentos para el usuario.

Como vemos, estas reglas están fundamentadas en lenguaje simbólico y no en lenguaje matemático, por lo que hay que buscar una forma de parametrizar la información y hacerla más determinista y discreta que el lenguaje natural, para lograr esto, se ha optado por la siguiente estrategia:

* El núcleo de la ingeniería de reglas estará fundamentado sobre etiquetas que definan los alimentos, estas etiquetas son características de tipo “CARNE”, “ORIGEN\_ANIMAL”, “GLUTEN”, “HUEVO” …etc. que definen al alimento.
* Cada alimento este compuesto por una o más etiquetas.
* Además, sobre cada usuario se almacena información, por ejemplo, a que tipos de alimentos es alérgico, el tipo de alimentación, alimentos que no desea consumir…etc.

Con todo esto podemos establecer una asociación entre los datos del usuario que solicita alimentos y los datos almacenados de alimentos en la base de datos, por ejemplo, si un usuario tiene la etiqueta “GLUTEN” en alergenos, entonces todos los alimentos que contengan una etiqueta gluten no se muestran.

Además, esta asociación directa entre un dato de un usuario y la filtración de alimentos con determinados valores se acopla muy bien a consultas filtradas a base de datos, por ello, el sistema basado en reglas esta muy acoplado con consultas a la base de datos no relacional, en este caso, mongoDB, para más detalles acerca de la implementación, ir al apartado [5.5 Implementación de sistema basado en reglas](#_Implementación_de_sistema).

## Diseño arquitectónico del servidor.

El diseño arquitectónico del servidor es el propio de un servicio REST, el cual consiste a limitarse a hacer operaciones CRUD sobres los datos, hacer búsquedas filtradas, servidor datos calculados a partir de otros que recibir y enviar/recibir información en formato JSON en la mayoría de las operaciones, además, se debe enfatizar que se ha seguido un patrón MVC sin la vista, es decir, un patrón modelo-controlador en el que el controlador se encarga de recibir peticiones del lado del cliente, procesarlas, recuperar datos de la base de datos y hacer toda la lógica de negocio necesaria para servir respuestas adecuadas al propósito de las peticiones.

Además, el servidor tendrá un área para tener centralizado el código del sistema basado en reglas, así como distintas áreas para las diferentes entidades de la aplicación, las cuales, son:

* **Controladores:** Se encargan de ejecutar la lógica funcional asociada a una petición, por ejemplo, “añadir un alimento”, hay que comprobar que el usuario es quien dice ser y que el alimento es integro y cumple validaciones básicas para ser insertado en la base de datos.
* **Modelos:** Es la estructuración de la información y como se relaciona entre ella, estableciendo campos de cada una de las entidades (alimentos, registros diarios de consumo, secciones del usuario…etc.), requisitos, validaciones, limitaciones, relaciones, campos calculados…etc.
* **Routes:** Establecen una relación directa entre la ruta del servidor que se invoca y el controlador relacionado, por ejemplo, si se llama con el verbo GET al servidor y la url miservidor.com/alimentos se invoca al controlador que devuelve todos los alimentos.

Con estos tres elementos, ya se puede apreciar el modelo controlador-vista aplicado en la arquitectura, en el apartado [5.3 – Implementación de arquitectura back-end](#_Arquitectura_back-end.) se podrá apreciar más al dellate otros componentes

## Diseño de sistema de autenticación y autorización (Ciberseguridad).

El sistema de autenticación y autorización estará basado en la tecnología json-web-token, esta tecnología se fundamenta en la generación de token para validar que el usuario es quien dice ser.

Cuando el usuario inicia sesión o crea una nueva cuenta, al entrar en la aplicación, se firmará un token, en el lado del servidor, el cual, solo es capaz de comprobar su validez el propio servidor, dado que tiene una clave privada a la que no puede acceder nadie más que el servidor.

El token tiene además incrustada información como el tiempo de expiración del token (cuando deja de ser valido), o datos imprescindibles del usuario, como, por ejemplo, el identificador de usuario o el email.

Entonces, una vez se ha generado el token, cada petición postuma a la fase de autorización tendrá injectada el token en el header de la petición, concretamente en el campo Authorization que permitira al usuario autenticarse.



La seguridad no se implementa a nivel de capacidad para descifrar el token, dado que cualquier persona que tenga un json-web-token podría descifrar la información que este contienen sin ningún tipo de problema de forma muy fácil (por esto, nunca se debe poner información sensible como la contraseña, roles…etc. en el propio token). Sin embargo, la seguridad radica en la capacidad de comprobar si el token recibido es autentico y sino ha sido modificado.

Una característica intrínseca de los tokens generados con json-web-token es que, al ser modificados, la firma cambia, por ende, el servidor se cerciorara de que ha sido modificado y lo verá como no autentico, denegando acceso a recursos o rutas.

Además, una vez se recibe el token en el lado del servidor con la petición, tras comprobar que el token es válido, se pasa a la fase de autorización, en la que se comprueba, gracias a la información contenida en el token, como el identificador de usuario, que este tiene permisos para acceder al recurso solicitado.

Con todo esto tenemos un sistema de autenticación y autorización bien definido que nos permite abordarlo más al detalle en la fase de implementación, la cual, esta especificada en el apartado [5.6 Implementación de sistema de autenticación y autorización.](#_Implementación_de_sistema_1)

## Contratos de comunicación servidor-cliente.

En este apartado se diseña de manera superficial la frontera entre el lado del servidor y el lado del cliente, es decir, a que se compromete el servidor servir antes determinadas llamadas. Cabe reseñar que esta parte es fundamental para que el equipo de desarrollo del lado del cliente sea totalmente independiente del equipo de desarrollo del lado del servidor. Si el equipo del lado del cliente, sabe de antemano que tipo de información se recibe del lado del cliente para cada tipo de llamada antes de la implementación especifica de los puntos de llamada del servidor (endpoints), este puede ir construyendo el lado del cliente sin ni siquiera recibir datos del back-end, haciendo datos falsos en front-end para hacer pruebas y desarrollar.

Con el fin de que estos datos falsos se parezcan lo máximo posible a lo que se encontrara el sistema cuando se conecte con el lado del servidor, se deben establecer unos contratos o compromisos que digan como van a hacer las peticiones desde el lado del cliente (parametors, valores enviados en el body, forma de url…etc.) y que es lo que se espera recibir del back (respuesta en formato json, una imagen, un valor numérico…etc.

Tal y como se ha hecho en anteriores apartados en esta fase se verá un diseño menos especifico que en la fase de implementación, limitándonos a hacer una descripción con lenguaje natural de lo que esperamos que cumpla el lado del servidor cuando se le hacen ciertas peticiones, estas peticiones las podremos agrupar en las siguientes familias:

* Llamadas asociadas al sistema de autenticación.
* Llamadas asociadas a operaciones CRUD sobre alimentos.
* Llamadas asociadas a operaciones CRUD sobre secciones de alimentos.
* Llamadas asociadas a operaciones CRUD sobre registros diarios de consumo de alimentos.
* Llamadas asociadas a operaciones CRUD sobre datos del usuario, como, por ejemplo, datos fisiológicos, foto de perfil, alergenos, tipo de alimentación, objetivo nutricional…etc.
* Llamadas que invoquen al servicio de recomendación de alimentos.

Además, como aspecto relevante, todas las peticiones deberán tener en su header el token que les permite al servidor identificar al usuario y comprobar que realmente es quien dice ser, así como controlar el acceso a recursos de este.

A continuación, en los próximos epígrafes se detalla el diseño de cada una de estas familias o grupos de peticiones.

### Llamadas asociadas al sistema de autenticación.

|  |  |  |
| --- | --- | --- |
| **AUTENICACIÓN Y AUTORIZACIÓN** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
| Crear cuenta | * Se esperará un objeto de tipo usuario con email, contraseña y nombre de usuario. | * Un objeto de tipo usuario con identificador asociado y con un token en el cuerpo de la respuesta. |
| Iniciar sesión | * Se espera un objeto en formato json, con email y password. | * Misma respuesta que en caso anterior, el objeto de tipo usuario en formato json, con token de la nueva sesión. |
| Obtener todos los usuarios | * Que el usuario que haga la petición sea de tipo admin o super-admin | * Un listado de todos los usuarios del sistema. |

### Llamadas asociadas a operaciones CRUD sobre alimentos.

|  |  |  |
| --- | --- | --- |
| **OPERACIONES CRUD SOBRE ALIMENTOS** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
| Crear nuevo alimento | * Se esperará un objeto de tipo alimento en formato json con los datos validados e íntegros sin identificador | * Un objeto de tipo alimento con un identificador de base de datos asociado |
| Consultar un alimento según su identificador | * Se espera en la propia url un identificador del alimento. A consultar | * En el cuerpo de la petición se responder con una instancia de un alimento en formato json o un objeto vacío. |
| Consultar todos los alimentos | * Se espera, en caso de que el usuario lo desee, una url cuyos parámetros filtren la búsqueda, con paginación, ocultado de ciertos campos, ordenación, condiciones, expresiones regulares…etc. | * Un array de alimentos que cumplas las especificaciones delimitadas en la URL, junto con el número de alimentos encontrados. |
| Editar un alimento | * Se espera como parte de la url el identificador del alimento a modificar. * Se espera como cuerpo de la petición aquellos atributos que se desean modificar (no son necesarios todos). | * El alimento modificado en formato json en el cuerpo de la respuesta |
| Eliminar un alimento | * Se espera como parte de la url el identificador del alimento a eliminar. | * Nada, un estado de respuesta 204 indicativo de no contenido. |

### Llamadas asociadas a operaciones CRUD sobre secciones de alimentos

|  |  |  |
| --- | --- | --- |
| **OPERACIONES CRUD SOBRE SECCIONES DE ALIMENTOS** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Llamadas asociadas a operaciones CRUD sobre registros diarios de consumo de alimentos.

|  |  |  |
| --- | --- | --- |
| **OPERACIONES CRUD SOBRE REGISTROS DIARIOS DE CONSUMO** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Llamadas asociadas a operaciones CRUD sobre datos del usuario

|  |  |  |
| --- | --- | --- |
| **OPERACIONES CRUD SOBRE DATOS DEL USUARIO** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

### Llamadas que invoquen al servicio de recomendación de alimentos

|  |  |  |
| --- | --- | --- |
| **AUTENTICACIÓN** | | |
| **Caso** | **¿Qué se espera como petición?** | **¿Qué se espera como respuesta?** |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

## Diseño arquitectónico del front-end (lado cliente).

## Conclusiones.

# Implementación

## Introducción

En este apartado se baja un escalón más hacia un nivel menos de abstracción, siendo más específicos con la tecnología y la manera en la que se va a implementar la aplicación, empezando por la definición arquitectónica global y continuando con una estructura similar a la de la fase de diseño, es decir, con un enfoque desde back-end hacia front-end.

Además, subrayar que, con objeto de no extender demasiado la memoria, solo se adjuntan aquellas capturas de código fuente necesarias, dado que todo el código fuente se adjunta junto con la memoria.

## Stack tecnológico del proyecto.

NODEJ  
*JavaScript en el lado del servidor*

EXPRESS  
*Frameworkde nodejs que facilita el trabajo del programador*

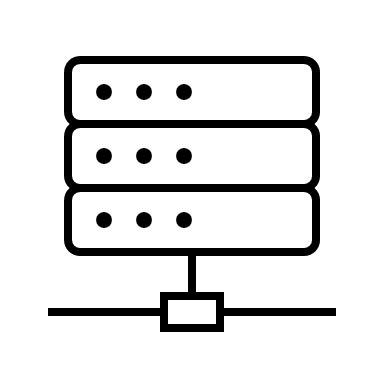
MONGOSE  
*Interfaz nodejs con la base de datos no relacional*



MONGODB  
Base de daos no relacional orientada a documentos

HTML, CSS, JS  
*Tecnologís de desarrollo web en el lado cliente*

ANGULAR  
*Framework de javascript que facilita el trabajo al programador*



Tal y como se ilustra en la imagen de arriba, en el lado front-end se ha optado por usar Angular como entorno de trabajo, la tecnología Angular lo que permite es facilitar el trabajo del programador, implicando que cosas para las que antes se necesitaban varias líneas de codificiación, ahora se pueden hacer en solo una línea. Lo mismo pasa en el caso de Express y mongoose, son entornos de trabajo que agilizan los proyectos, permitiendo al programador centrarse en la lógica de negocio y no tanto en el desarrollo e implementación de operaciones que suelen repetirse mucho en la mayoría de los proyectos software.

En el mundo profesional, es impensable trabajar sin tener estos tipos de frameworks, dado que el cliente que encarga el desarrollo del software, en la mayoría de los casos, tiene mucha prisa por utilizar el software demandado. Es por ello, que, si el cliente tiene prisa, el programador no pierda mucho tiempo, por ejemplo, desarrollando una función que dada una url y un verbo emita una petición HTTP o HTTPS a servidor. Con un framework esa función ya esta implementada y el programador no debe preocuparse por ello, agilizando procesos de desarrollo.

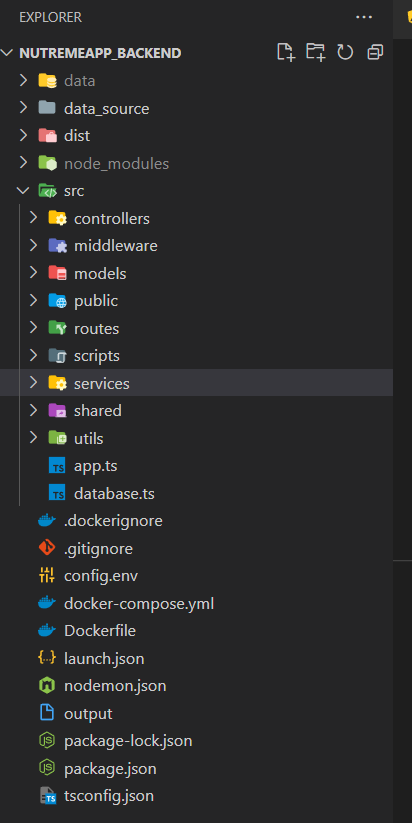
El motivo por el que se ha elegido una base de datos no relacional es por su flexibilidad en cuanto a comportarse como una base de datos relacional o como una base de datos orientada a documentos, es decir, ciertas partes del modelado de datos pueden hacer referencia con identificadores a otros documentos y también se puede optar por hacer anidamiento de documentos dentro de otros documentos, dependiendo del caso de uso se utilizará una estrategia u otra.

Además, las bases de datos no relacionales al devolver la información lo hacen en un formato JSON lo cual se acopla muy bien a nodejs e incluso a la transmisión de información propia de una API.

Finalmente, se usa nodejs con express dada su simplicidad en el proceso de desarrollo, se consiguen hacer ciertas operaciones como establecer conexión con base de datos, levantar servidor, activar puertos, desarrollo de end-points…etc con una cantidad considerablemente más baja de líneas de código y de esfuerzo de desarrollo, por lo que el programador puede centrarse más en el desarrollo propio de la lógica de negocio y no tanto en tareas rutinarias propias de configuración.

## Implementación de arquitectura back-end.

A nivel arquitectónico el back-end tiene la estructura expuesta en la imagen de abajo, se sigue una arquitectura Model-controller dado que la aplicación es pequeña y se presupone que no crecerá a lo largo del tiempo, por lo que no es necesario hacer sobre-ingenieria desarrollando microservicios o arquitecturas limpias que separen la dependencia del software de la base de datos que se utiliza o del framework que se utiliza, pudiendo hacer que sea más testeable y que se pueda migrar con facilidad a otros frameworks o base de datos.

En la imagen de la izquierda tenemos los siguientes elementos:

* **Src/app.ts:** Archivo encargado de levantar el servidor y cargar configuraciones básicas del servidor.
* **Database.**ts: Conexión con la base de datos.
* **Models:** Esquemas de la base de datos, es decir, definen como se estructura y relaciona la información.
* **Routes:** Por cada ruta que se llame se asocia un determinado controlador.
* **Controllers:** Establecen la conexión entre las peticiones recibidas y la lógica de programación asociada y también actúan como puente entre peticiones y base de datos.
* **Middleware:** Es un software que se ejecuta entre la petición desde el cliente y la devolución de esta, por ejemplo, para cada petición de cliente se debe ejecutar un software que compruebe si el usuario es quien dice ser antes de permitirle acceder al controlador.
* **Servicios:** Servicios que permiten cálculos nutriticonales, la ingeniería de reglas…etc.
* **Shared:** Clases, enumerados, interfaces…etc. que se comparten a lo largo de todo el proyecto en diferentes lugares.
* **Scripts:** Software que automatiza ciertas operaciones, por ejemplo, insertar de manera masiva información.
* **Utils:** Utilidades como conversión de tipos, manejador de errores…etc.
* **DockerFile:** Monta una imagen virtual del proyecto, haciéndolo exportable a cualquier sistema operativo que tenga Docker instalado, sin preocuparse por dependencias o versiones instaladas en el pc.

## Implementación de base de datos.

### Entidades de la relación.

Las entidades de relación y sus relaciones están explicadas en la fase de diseño, concretamente en el apartado [4.2 Diseño de base de datos.](#_Diseño_de_base)

Los aspectos que reseñar con respecto a la fase de diseño son lo siguientes:

* Cada entidad puede estar compuesta por cualquier tipo de atributo, tanto vectores compuestos por objetos complejos, como objetos complejos, así como tipos simples, esto hace que el modelado de la información sea mucho más flexible que en las bases de datos más tradicionales como las relacionales.

## Implementación de sistema basado en reglas.

## Implementación de sistema de autenticación y autorización.

# Pruebas

## Introducción.

Este apartado será para definir y comprobar las pruebas realizadas, las primeras para ver que cada bloque iba funcionando correctamente y la última es donde haremos una interpretación de los resultados. Como tenemos datos de los últimos 3 meses, haré una prueba completa para ver que errores nos hubiera dado el sistema en la previsión.

Con el sistema de dimensionamiento, compararé los dimensionamientos, indicados, los que realmente se han tenido y obtendremos la fiabilidad de los datos.

## Pruebas unitarias

## Pruebas de integración

## Pruebas de validación

## Conclusiones

# Cronograma y costes del proyecto

Incluiré el cronograma real, horas de investigación, análisis y elaboración de la aplicación y del trabajo fin de grado.

De esta forma puedo aplicar un coste real a la aplicación y a su implantación separándola de lo que supone hacer un TFG, en el que incluye memoria, defensa, preparación, etc...

# Conclusiones y trabajos futuros

Usaré este apartado para indicar que el sistema está en producción, que está dando una tasa de error en la desviación baja y que ya se han solicitado varías mejoras al sistema, como incluir la previsión de ventas en base a tasas de conversión históricas y un módulo para la revisión de costes asociados a distribuir el tráfico entre plataformas de otra forma.

### Un aspecto fundamental en la ingeniería: La abstracción.

# Bibliografía

# Glosario

# Anexos

## Manual de la aplicación

## Datos origen para simulación y pruebas