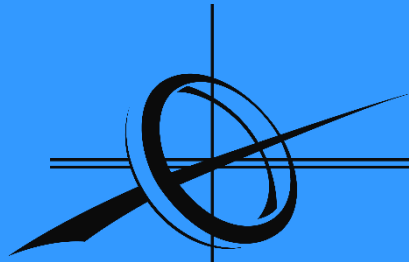




POLITÉCNICA



Universidad
Politécnica
de Madrid

**ETSI SISTEMAS
INFORMÁTICOS**

Alafia: Desarrollo de una Progressive Web Application para la gestión de pedidos y de experiencias inmersivas para un restaurante africano

Proyecto Fin de Grado

Grado en Ingeniería de Computadores

Autor:

Daniel García Alonso

Tutores:

Carlos Camacho Gómez

Agradecimientos

Quiero transmitir mis más sinceros agradecimientos a todas aquellas personas que me han apoyado durante el desarrollo de este proyecto, y en especial a las personas que he conocido durante el transcurso de mi persona por la Universidad.

En especial al Doctor Carlos Camacho, tutor de este proyecto que me ha apoyado en todo lo necesario para el correcto desarrollo del mismo. A todos los profesores y profesores que se han enfocado en que aprendamos a aprender, en lugar de limitarse a su materia. A mi familia, pareja y amigos por tenerlos siempre empujando de mi hacia adelante.

Por supuesto a la Universidad Politécnica de Madrid, por dejarme formar parte de ella y facilitarme sus instalaciones para mi aprendizaje.

Desarrollar una aplicación de esta envergadura ha tenido un gran impacto en mi persona, ya muchas veces no te das cuenta de lo que eres capaz hasta que te pones a hacerlo, y gracias a todos lo que me apoyaron, este proyecto ha podido salir adelante.

Así que, a todos ellos, un millón de gracias.

Resumen

El ser humano tiende a empatizar y a establecer vínculos con aquellos con los que se identifica. Cada día, la sociedad va avanzando hacia un mundo más unificado y globalizado en donde, gracias a la tecnología, las fronteras son cada vez más sutiles. Esto pasa sobre todo en culturas del primer mundo y por ello, las diferencias culturales entre estos países y aquellos del tercer mundo tienden a aumentar. También es notable el hecho de que juntarnos con unas culturas menos desarrolladas, a veces, puede generar cierto rechazo. Sin embargo, hay varios puntos de unión entre culturas, sin lugar a dudas uno de ellos es el gusto por la buena comida.

Por ese motivo nace Alafia, un movimiento que pretende retirar ciertos estigmas que nos retrasa el desarrollo como sociedad. Alafia es un restaurante cuyo foco principal gastronómico es el de las culturas africanas. Comida típica de la zona de Nigeria, realizada con ingredientes de origen africano y empleados provenientes de la zona es lo que ofrece Alafia. Además de la fusión con el mundo contemporáneo con el uso de últimas tecnologías en la mesa, ya que toda la gestión de solicitudes de los clientes es puramente digital a través de una Tablet que dispone cada cliente en la mesa y que muestra una aplicación web.

Debido a las circunstancias actuales producidas por la pandemia, han sido muchos los restaurantes que se han tenido que adaptar y ciertamente modernizar, dado que, para poder seguir dando servicio, han debido tomar medidas como la sustitución de las cartas físicas de los restaurantes por un código QR para poder acceder a la carta.

Este procedimiento funciona muy bien con el público general, sobre todo con los jóvenes, dado que es muy raro el caso de que alguien de la mesa no disponga de un dispositivo con conexión a internet y con la capacidad de escanear un código QR disponibilizado por el restaurante para ver la carta que ofrecen. Sin embargo, con un público de mayor edad, esto puede complicarse debido a una posible falta de conocimientos o destrezas con su dispositivo personal para poder escanear un código QR.

Por eso en Alafia, se pone a disposición de los clientes un dispositivo que será de uso personal durante su estancia, y que estará preparado para facilitarle al máximo la experiencia digital a los clientes. Evitamos así, que los clientes tengan que usar su dispositivo personal, causando un menor derroche de batería y de datos en sus dispositivos personales, pues no les hará falta para nada en el restaurante.

Como posible ampliación, se puede estudiar la opción de exponer la aplicación en una red wifi privada del restaurante, con el objetivo de que los clientes que lo deseen puedan acceder a la aplicación en sus dispositivos personales, aunque tengan a su disposición el dispositivo proporcionado por el restaurante para la experiencia personal óptima.

Es este el cometido principal de este proyecto, el desarrollo full stack de principio a fin de una aplicación web que permita a los clientes y al personal del restaurante, una gestión personalizada sobre su sesión gastronómica, a través de un dispositivo que será personal para cada cliente.

El dispositivo en cuestión muestra una aplicación web, con el que cada cliente podrá gestionar comida, bebida y servicios del restaurante, de forma que todos los movimientos quedan registrados digitalmente, ya que todo movimiento se persiste en una base de datos controlada por un servidor dedicado únicamente al tratado de los datos que le provee la aplicación web.

El proceso que experimenta un cliente en Alafia será muy parecido al descrito a continuación:

- Llegan los clientes al restaurante
- El jefe de sala les indica cuál es su mesa (en el dispositivo está configurado el cliente que hizo la reserva)
- El cliente toma su asiento
- El cliente confirma que está en su mesa o selecciona el cliente que le hizo la reserva si estuviera otro configurado.
- El cliente confirma su asiento, quedando configurado el dispositivo para su experiencia personal o si está sustituyendo a alguien se configura con sus datos personales
- El cliente puede gestionar diferentes operativas que tiene disponibles durante su estancia, como solicitar un gestor de experiencia, pedir bebidas, consultar la cuenta individual o en conjunto de la mesa, o realizar un test de impacto social

Paralelamente, el jefe de sala tendrá su propio flujo de trabajo, por resumirlo al igual que con los clientes será muy parecido a:

- El jefe de sala deberá introducir unas credenciales concretas para poder acceder a esta parte de la aplicación, pues esta securizada.
- Una vez identificado, tendrá disponible un grid con todas las mesas configuradas para el restaurante. En cada casilla se muestra además una casilla de notificación, que indica que algún cliente de esa mesa ha solicitado la participación de un gestor de experiencia.

- Haciendo click en alguna de las mesas configuradas, se abre en una ventana emergente el detalle de esa mesa, mostrando información de cada cliente que la componen.
- Una operación importante a realizar por el jefe de sala es la de lanzar la experiencia a un cliente, que consiste en lanzarle un video en la tablet del cliente que tiene un plato configurado y que le va a ser servido inminentemente.

En el documento se va a ir detallando cada parte de la aplicación en profundidad. Por hacer una introducción, veremos un estudio de las tecnologías más usadas en el mercado actual para este tipo de desarrollos de software; se expondrán todas las funcionalidades con las que cuenta esta version del proyecto; veremos la elección de tecnologías para el desarrollo, tanto del servidor, como de la web y la base de datos; veremos cómo se ha montado la base de datos; se mostrará en detalle cómo se ha montado la web y cada componente que la compone; al igual que con el servidor, que veremos cómo se ha desarrollado y que hace cada parte que lo compone; se expondrán los puntos más importantes de sincronización que requiere la aplicación y como se han abordado; y por último se expondrá una sección con el impacto social que se quiere medir y como, además de unas conclusiones y reflexiones finales para finalizar este documento.

Abstract

Human beings tend to empathize and establish bonds with those with whom they identify. Every day, society is moving towards a more unified and globalized world where, thanks to technology, borders are becoming increasingly subtle. This is especially true in first world cultures and, as a result, the cultural differences between these countries and those of the third world tend to increase. It is also notable that coming together with less developed cultures can sometimes generate a certain rejection. However, there are several points of union between cultures, undoubtedly one of them is the taste for tasty food.

For this reason, Alafia was born, a movement that aims to remove certain stigmas that delay our development as a society. Alafia is a restaurant whose main gastronomic focus is on African cultures. Typical food from the area of Nigeria, made with ingredients of African origin and employees from the area is what Alafia offers. In addition to the fusion with the contemporary world with the use of latest technologies at the table, since all the management of customer requests is purely digital through a Tablet that has each customer at the table and displays a web application.

Due to the current circumstances produced by the pandemic, there have been many restaurants that have had to adapt and certainly modernize, since, in order to continue providing service, they have had to take measures such as replacing the physical letters of the restaurants by a QR code to access the menu.

This procedure works very well with the general public, especially with young people, since it is rare that someone at the table does not have a device with an internet connection and the ability to scan a QR code provided by the restaurant to see the menu offered. However, with an older audience, this can be complicated due to a possible lack of knowledge or skills with their personal device to be able to scan a QR code.

For this reason, Alafia provides customers with a device that will be for their personal use during their stay, and that will be prepared to facilitate the digital experience to the maximum for customers. This way, we avoid those customers have to use their personal device, causing less waste of battery and data on their personal devices, as they will not need them for anything in the restaurant.

As a possible extension, we can study the option of exposing the application in a private Wi-Fi network of the restaurant, with the objective that customers who wish can access the application on their personal devices, although they have at their disposal the device provided by the restaurant for the optimal personal experience.

This is the main task of this project, the full stack development from start to finish of a web application that allows customers and restaurant staff, a personalized management of their dining session, through a device that will be personal to each customer.

The device in question displays a web application, with which each customer can manage food, drink, and restaurant services, so that all movements are digitally recorded, since every movement is persisted in a database controlled by a server dedicated solely to the processing of data provided by the web application.

The process that a customer experiences at Alafia will be remarkably like the one described below:

- Customers arrive at the restaurant
- The waiter tells them which table is their table (the customer who made the reservation is configured in the device).
- The customer takes his seat
- The customer confirms that he/she is at his/her table or selects the customer who made the reservation if another one is configured.
- The customer confirms his seat, and the device is configured for his personal experience or if he is replacing someone else, it is configured with his personal data.
- The customer can manage different operations available to him during his stay, such as requesting an experience manager, ordering drinks, consulting the table's individual or overall bill, or performing a social impact test.

In parallel, the room manager will have his own workflow, to summarize it as with the customers, it will be remarkably like the following:

- The room manager will have to enter specific credentials to access this part of the application, as it is secured.
- Once identified, he will have available a grid with all the tables configured for the restaurant. In each box there is also a notification box, which indicates that a customer of that table has requested the participation of an experience manager.

- By clicking on any of the configured tables, a pop-up window opens with the details of that table, showing information on each client that composes it.
- An important operation to be performed by the room manager is to launch the experience to a customer, which consists of launching a video on the customer's tablet that has a dish configured and that is going to be served imminently.

The document will detail each part of the application in depth. To make an introduction, we will see a study of the most used technologies in the current market for this type of software developments; all the functionalities that this version of the project has will be exposed; we will see the choice of technologies for the development, both the server, the web and the database; we will see how the database has been assembled; it will be shown in detail how the web has been assembled and each component that composes it; As with the server, we will see how it has been developed and what each component does; the most important synchronization points required by the application and how they have been addressed; and finally we will present a section with the social impact that we want to measure and how, as well as some conclusions and final thoughts to conclude this document.

Tabla de contenido

Agradecimientos	2
Resumen	3
Abstract	6
Índice de Figuras.....	11
Tecnologías en el mercado.....	13
Front End.....	13
React.js	13
Angular	14
Flutter.....	14
Back End	14
Node.js	14
Spring Boot.....	15
Ruby on Rails	15
Bases de datos.....	15
MySQL	15
MongoDB.....	16
Funcionalidades de la aplicación web.....	17
Elección de tecnologías	34
Base de datos (MongoDB).....	35
Front End (Angular)	43
Componente AppComponent	49
Componente WelcomeComponent	50
Componente WarmUpComponent.....	50
Componente ReplaceClientComponent	51
Componente AddClientComponent.....	51
Componente WaitDinersComponent.....	51
Componente DrinksComponent	52
Componente AppMenuComponent.....	52
Componente ExtrasComponent.....	53
Componente BillComponent.....	53
Componente TableBillComponent.....	54
Componente MigrationTestComponent.....	54

Componente ExperienceComponent.....	55
Componente LoginComponentComponent.....	55
Componente RoomManagerComponent	55
Componente SelectedTableComponent	56
Back End (Spring Boot).....	57
Gestor de dependencias	57
Generación del proyecto de Spring.....	58
Fichero de configuración del proyecto Gradle.....	60
Estructura de directorios del Back End	61
Configuración del servidor	61
Código de producción	62
Clase principal de la aplicación	62
Packages que componen la aplicación.....	63
Código de Test.....	70
Tests Unitarios.....	70
Tests de Integración	73
Tests Funcionales	75
Puntos clave de sincronización.....	76
Sincronización a la hora de configurar todos los dispositivos para la experiencia personalizada Alafia	76
Sincronización para lanzar contenido multimedia al recibir el plato configurado	77
Sincronización para mostrar notificación cuando se solicita un gestor de experiencia	78
Impacto social	80
Conclusiones	81
Bibliografía.....	82
Diagramas de la aplicación:.....	82

Índice de Figuras

Figura 1. Diagrama de flujo de la aplicación web.....	18
Figura 2. Pantalla de inicio	19
Figura 3. Selección del cliente que hizo la reserva de la mesa	20
Figura 4. Confirmación de comensal en la mesa.....	20
Figura 5. selección del comensal a sustituir	21
Figura 6. Recogida de datos de cliente que sustituye	22
Figura 7. Ventana de espera a los demás comensales.....	23
Figura 8. selección de bebida	23
Figura 9. Confirmación de bebida seleccionada	24
Figura 10. menú principal.....	24
Figura 11. Selección de bebida (ventana modal)	25
Figura 12. Gestor de experiencia solicitado	26
Figura 13. Ventana modal facturas	27
Figura 14. Factura personal.....	28
Figura 15. Factura de la mesa completa	28
Figura 16. Test de concienciación y valoración.....	29
Figura 17. Pantalla de login	30
Figura 18. Pantalla del jefe de sala.....	30
Figura 19. Información de la mesa seleccionada por el jefe de sala.....	31
Figura 20. Estados de un plato	32
Figura 21. Pantalla experiencia visual	33
Figura 22. Diagrama Entidad Relación Alafia	35
Figura 23. Jerarquía de archivos del Front End.....	44
Figura 24. Árbol de ficheros directorio componentes	47
Figura 25. Directorio de modelo del Front End.....	47
Figura 26. Servicios del Front End	48
Figura 27. Spring Initializr	58
Figura 28. Estructura básica proyecto Back End	59
Figura 29. Estructura de ficheros src del servidor.....	61
Figura 30. swagger-ui del servidor	64

Figura 31. Diagrama secuencia bloqueo espera de confirmación de todos los usuarios en la mesa	77
Figura 32. Diagrama secuencia lanzamiento de experiencia por jefe de sala	78
Figura 33. Solicitud de gestor de experiencia	79

Tecnologías en el mercado

Para el desarrollo de la aplicación, hoy más que nunca tenemos una gran variedad de tecnologías a poder elegir. Como se trata de un desarrollo del aplicativo completo, es decir, que no solo se va a construir la parte visual (Front End), sino que va a ser totalmente funcional y va a llevar el respaldo de un servidor (Back End) para el procesamiento y guardado de los datos en la base de datos del sistema.

Por tanto, tenemos que estudiar posibles tecnologías para 3 cometidos diferentes. Abordar en Front End, abordar el Back End y abordar la base de datos. Para quitar cualquier atisbo de dudas, primero definamos que es cada pieza.

El Front End, en pocas palabras, se trata de una serie de tecnologías que son usadas para el desarrollo de la interfaz visual de aplicaciones web, con el fin de diseñar, estructurar, animar o adaptar la interfaz a los usuarios de una forma atractiva para ellos.

El Back End, como adelantamos en párrafos anteriores, es el servidor que va a estar en comunicación con el Front End, ejecutando procesos con datos que le lleguen del Front End, por ejemplo. Hace también de interfaz para la base de datos, y atiende a las peticiones que le lleguen del Front End.

Y, por último, la base de datos, como su propio nombre indica, es el conjunto de datos almacenados en un computador, que nos permite realizar consultas, inserciones, modificaciones y borrado de los datos almacenados.

Veamos en detalle algunas de las tecnologías más importantes disponibles en el mercado:

Front End

React.js

Se trata de una librería open-source licenciada por el MIT y lanzada al mercado en el 2013. Probablemente sea el framework de JavaScript más usado, y con cada actualización mejora cada vez más.

La mantiene el gigante Facebook, y destaca por la facilidad de reusabilidad de sus componentes, ya que hace que las aplicaciones se subdividan fácilmente en componentes.

Algunos ejemplos de uso comercial son Airbnb, Dropbox, BBC, Facebook, New York Times, y Reddit.

Angular

Es la segunda librería más usada en el mercado, y compite cara a cara con la anterior, React.js. Es una librería open-source licenciada por el MIT y lanzada en el año 2009.

La mantiene otro gigante de la tecnología, Google. Destaca por su misión de estandarizar las aplicaciones web, usando módulos y estructuraciones de código estandarizadas.

Ejemplos de uso comercial son PayPal, Gmail, IBM, Netflix y The Guardian.

Flutter

Fundada por Google como Angular, permite a los desarrolladores crear interfaces responsive y vistosas. Licenciada bajo BSD (*Berkeley Software Distribution*).

Su principal atractivo es que como parte de la aplicación web, Flutter permite al desarrollador formar la página con widgets, lo cual la hace muy responsive. Además, permite introducir cambios en el código y hacer un refresco de la página en caliente, lo cual agiliza mucho los desarrollos.

Hookel, Birch Finance, Alibaba, son algunos ejemplos que han optado por esta tecnología para exponer sus Front End a sus clientes.

Back End

Node.js

Es un framework de JavaScript como bien deja ver el propio nombre, y eso mismo lo hace tan viable para los desarrolladores que buscan convertirse en Full-Stack, ya que la parte de JavaScript la tienen muy vista.

Se trata básicamente de escribir el código del servidor con JavaScript, lo cual lo convierte en el framework más amigable para los desarrolladores de Front End. Es decir, que convierte a JavaScript en un lenguaje end-to-end en una aplicación web.

Ejemplos de usos comerciales son LinkedIn, Netflix, Uber, PayPal, NASA y eBay.

Spring Boot

Es un framework de Java, y su intencionalidad se basa en solventar los problemas que tiene Spring para el desarrollo de aplicaciones web, al igual que hace Spring con Java EE.

Spring por otro lado, es básicamente un inyector de dependencias para Java, lo cual nos brinda de muchas funcionalidades extra fácilmente, como exponer un API en el lado del servidor por ejemplo usando la dependencia de spring-web.

Ejemplos de usos comerciales son Accenture, Zalando, MIT, Santander y Fitbit

Ruby on Rails

Es el framework para los desarrolladores que no les atrae Java, JavaScript ni Python.

Se basa en el lenguaje Ruby como su nombre deja intuir, y permite renderizar plantillas HTML, actualizar bases de datos, enviar y recibir emails, provee de seguridad, etc. Es decir, lo hace todo, el Back y el Front End.

Ejemplos de usos comerciales son GitHub, Twitch, SoundCloud y Coinbase.

Bases de datos

MySQL

MySQL es un gestor de Bases de relacionales de código abierto con un modelo cliente-servidor. Para que sea posible esta comunicación, se usa el lenguaje SQL (Structured Query Language). Y nos permite realizar operaciones sobre la base de datos tales como:

- Control de acceso a los datos: decidir quién y puede y quien no puede ver o usar los datos almacenados en la base de datos.
- Identidad de datos: definir la tipología de los datos y la definición del esquema y relaciones de la base de datos.
- Consulta de datos: solicitar la información que se desea obtener que esté almacenada en la base de datos.
- Manipulación de datos: realizar operaciones a los datos tales como agregar, modificar datos existentes, eliminar registros, ordenación de datos, etc.

Ejemplos de uso comercial son empresas tales a Facebook, Twitter, YouTube, Yahoo!...

MongoDB

MongoDB es una base de datos no relacional que opera con documentos a diferencia de MySQL por ejemplo que trabaja con tablas. Los documentos que usa para el almacenamiento de la información tienen un formato JSON.

Dentro de un documento, nos permite no anclarnos a un único esquema, lo cual nos da más flexibilidad que una tabla de un sistema de base de datos relacional.

Las consultas que podemos realizar se basan también en JSON, ya que son llamadas a funciones sobre una consola JavaScript y lo que le pasaremos como parámetro será la consulta en un objeto JSON.

MongoDB por tanto es perfecto para cuando necesitamos almacenar datos semi estructurados, y se suele usar para aplicación CRUD (Create Read Update Delete) en muchos de los desarrollos web actuales.

Ejemplos de uso comerciales actuales son empresas como Ebay, Google, SquareSpace, Sega y Electronic Arts.

Funcionalidades de la aplicación web

La aplicación web se va a encargar de mostrar una interfaz gráfica de usuario (UI) para comunicar los datos que los clientes, a través de peticiones, deseen comunicar al servidor mediante diferentes funcionalidades disponibles en la aplicación.

Así que vamos a ir exponiendo pantalla a pantalla de la UI, y detallando cada una de las funcionalidades disponibles en dicha pantalla. Para un mejor entendimiento del flujo, la **Figura 1** muestra un diagrama con el flujo de la aplicación web y, seguidamente, veremos cada uno de los componentes de la aplicación en más detalle.

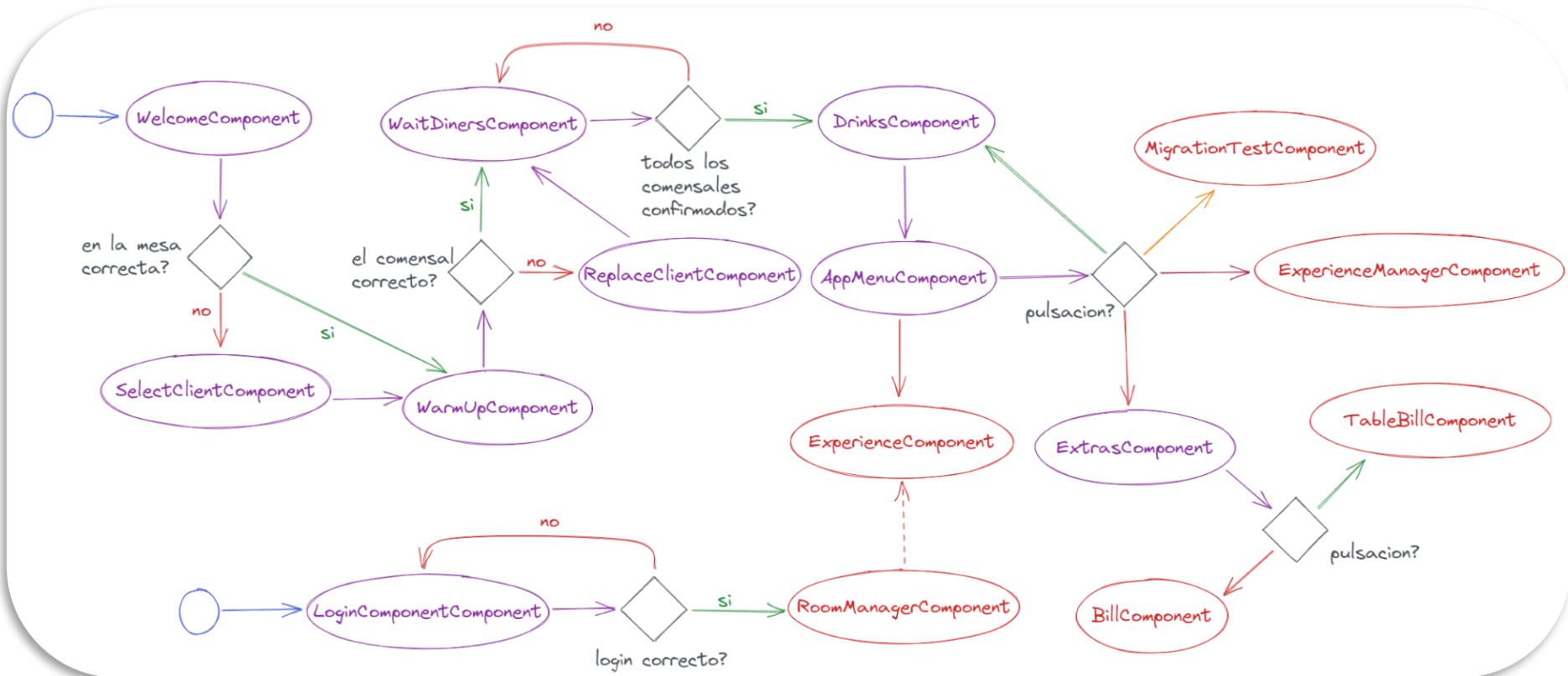


Figura 1. Diagrama de flujo de la aplicación web

La primera pantalla que se encuentran los clientes, nada más sentarse en la mesa será la que se muestra en la **Figura 2**.



Figura 2. Pantalla de inicio

En la pantalla de inicio, se disponen un texto de bienvenida, en el cual se indica el nombre del cliente que hizo la reserva de la mesa. En el ejemplo de la **Figura 2**, la reserva la habría realizado el cliente registrado con nombre “*name client_1*”. Seguidamente, se muestran dos botones. El primero, en color verde para que dé a entender que es un botón positivo, para continuar. Tiene la funcionalidad de confirmar que la tablet está configurada para el cliente “*name client_1*” y no para otro cliente que tenga otra reserva. El segundo, en color más rojizo, permite al comensal seleccionar el nombre del cliente que hizo la reserva de la mesa en la que está incluido. Al pulsar el botón se abre una ventana modal emergente, en la cual se incluye un listado con todos los nombres de los clientes que tienen reserva para ese día en el restaurante, tal y como vemos en la **Figura 3**.



Figura 3. Selección del cliente que hizo la reserva de la mesa

Una vez se ha confirmado que el comensal tiene en su Tablet, una mesa confirmada para el cliente que hizo la reserva, se procede a confirmar el sitio del comensal en la propia mesa. Para ello, tras pulsar el botón verde de la pantalla de la **Figura 2**, se muestra la siguiente pantalla en el flujo de la aplicación, la **Figura 4**.



Figura 4. Confirmación de comensal en la mesa

En esta pantalla, se muestra una cuadrícula con todos los comensales que tenía configurada la reserva de la mesa. En cada casilla, se muestra el nombre del comensal, por lo que cada uno, deberá pulsar sobre su nombre en la Tablet que tenga asignada en la mesa, quedando así asociado el comensal a la instancia de la aplicación web. En el caso de un cambio repentino, que no se pudo modificar la reserva original por algún motivo, la aplicación dispone a los usuarios un botón, que permite modificar los datos de cualquiera de los comensales. Tal y como se muestra en la **Figura 5**.

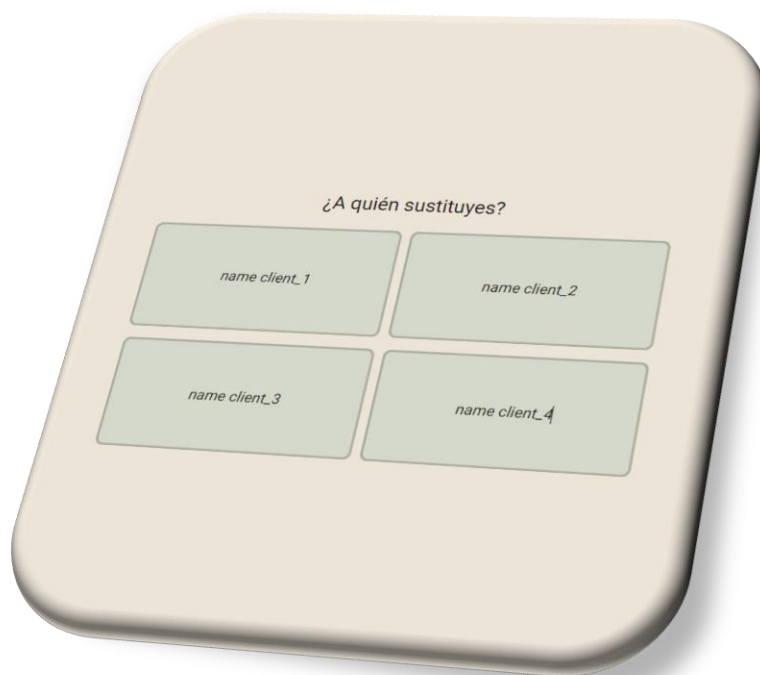


Figura 5. selección del comensal a sustituir

Una vez seleccionado a quien se quiere sustituir, se le piden al usuario a través de una pantalla modal emergente, los datos del sustituto, con el fin de poder enviarle información al correo electrónico, y para tener un control demográfico de los clientes que pasan por el restaurante. Esta pantalla es la mostrada en la **Figura 6**.

Una funcionalidad de navegación básica que incluye la aplicación es que cualquier ventana modal que se abra, el usuario puede pulsar sobre el fondo en cualquier momento para cerrarla. En algunas ventanas modales como esta, se incluyen botones de confirmación o para cerrar la

ventana. En el caso de esta pantalla modal, estos botones no aparecen hasta que se introducen datos en las entradas de texto que se muestran. Se muestra todo en la **Figura 15**.



Figura 6. Recogida de datos de cliente que sustituye

Pulsando en el botón de confirmación verde, pasaríamos a la siguiente pantalla. De igual manera, pasaríamos de pantalla si en la mostrada en la **Figura 4** el cliente se selecciona de los que se le muestra en la cuadrícula.

El usuario avanza hasta la siguiente pantalla, donde se le indica que tiene que esperar a que los demás comensales de la mesa en la que se encuentra, confirmen su sitio. Automáticamente, y mediante un sistema de sincronización que se detallará en el siguiente capítulo del documento, a todos los usuarios de la mesa se les avanza a la siguiente pantalla. Esta ventana de espera es la mostrada en la **Figura 7**.

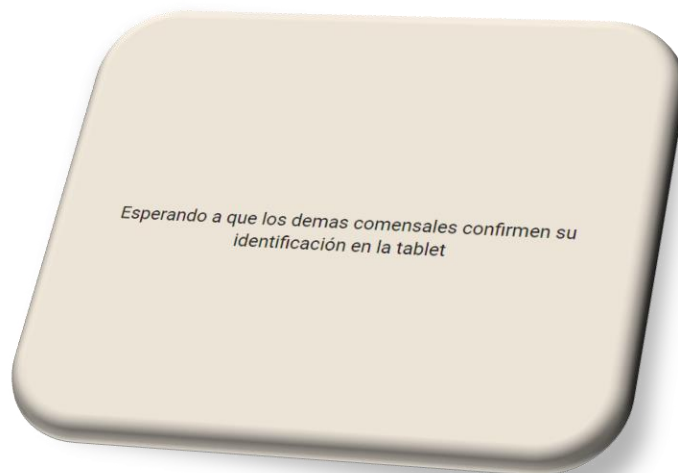


Figura 7. Ventana de espera a los demás comensales

La siguiente pantalla es la dedicada a la selección de bebida. Lo podemos ver en la **Figura 8**.



Figura 8. selección de bebida

De forma parecida a como se muestran los clientes de una reserva, se muestran las bebidas disponibles en el restaurante. Se les da a los clientes por supuesto la opción de no añadir ninguna bebida, pulsando el botón rojizo de la parte inferior de la UI.

Si el cliente opta por añadir una bebida, deberá pulsar sobre la que quiera seleccionar, y cuando lo haga, se mostrará un botón nuevo de confirmación de bebida seleccionada, en el cual se mostrará el nombre de la bebida seleccionada, tal y como muestra la **Figura 9**.



Figura 9. Confirmación de bebida seleccionada

Para pasar a la siguiente pantalla, el usuario debe pulsar uno de los dos botones, el verde si finalmente quiere añadir una bebida a su cuenta, o el rojizo si no quiere añadir nada. Por cualquiera de los dos, en este caso, se avanza a la siguiente pantalla, que mostrara el menú principal de la aplicación, mostrada en la **Figura 10**.



Figura 10. menú principal

En esta pantalla, es en la cual los usuarios van a tener más control sobre la aplicación, ya que es donde se les permite hacer más acciones. Se muestra una cuadrícula con cuatro botones:

- **Bebidas:** Muy parecida a la ventana por la que el usuario pasó previamente para seleccionar si quisiera una bebida que se añade a su cuenta personal. Pero esta vez en forma de ventana modal. Se muestra en la **Figura 11**.



Figura 11. Selección de bebida (ventana modal)

- **Gestor de experiencia:** Al pulsar sobre este botón, el usuario solicita que uno de los gestores de experiencia, es decir, que uno de los empleados del restaurante, nativo africano, se acerque al cliente para mantener una pequeña conversación acerca del arte del restaurante, por ejemplo, o sobre cualquier tema que tenga curiosidad de hablar con el gestor de experiencia. Al usuario entonces, le saldrá un aviso de que ha solicitado a un gestor de experiencia en el menú principal de la aplicación, tal y como se muestra en la **Figura 12**. Esto provoca que, al jefe de sala, le salga una notificación en la mesa que ha solicitado atención por un cliente. El jefe de sala es el encargado de asignar a un gestor de experiencia para atender la petición del cliente.

En todo momento, el cliente puede volver a pulsar sobre el botón de su menú principal para cancelar la solicitud. Esta acción, hace que se sincronice de nuevo con el dispositivo del jefe de sala para que la notificación sobre la mesa se elimine. De igual modo, el jefe de sala puede eliminar esta notificación pulsando sobre la misma en su pantalla, una vez la haya gestionado.

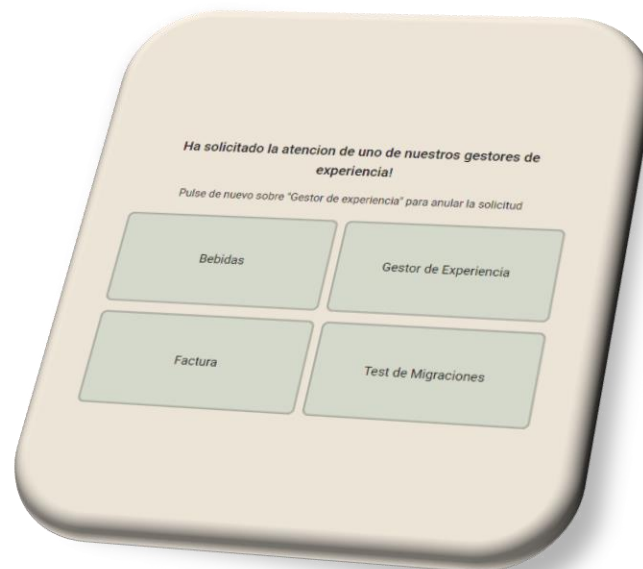


Figura 12. Gestor de experiencia solicitado

- **Factura:** Es el tercero de los botones del menú, y como bien indica su nombre, le permite al usuario acceder a la generación de la factura. Cuando el usuario pulsa sobre este botón, aparece una ventana modal con tres opciones, dos para generar una factura y la tercera para salir de la ventana modal, tal y como ilustra la imagen de la **Figura 13**.



Figura 13. Ventana modal facturas

El primero, genera la factura personal del comensal al que esta asignado el dispositivo en la mesa, sin tener en cuenta el resto. De forma que cada comensal pueda saber con total exactitud el importe de su factura.

El segundo de ellos “Pedir cuenta en conjunto”, genera la factura completa de la mesa, es decir, genera una factura añadiendo todos los platos y todas las bebidas de todos los comensales que están configurados en la mesa.

Pongamos un ejemplo ilustrado para la mesa reservada por “*name client_1*” y para uno de los comensales que está asignado a la reserva:

La cuenta individual del comensal “*name client_3*” es la que se muestra en la **Figura 14**, mientras que la cuenta total de la mesa es la que se muestra en la **Figura 15**.

Además, se incluye un botón que desde ambas opciones tiene el mismo comportamiento, volver al menú principal para seguir gestionando peticiones para el restaurante.



Figura 14. Factura personal

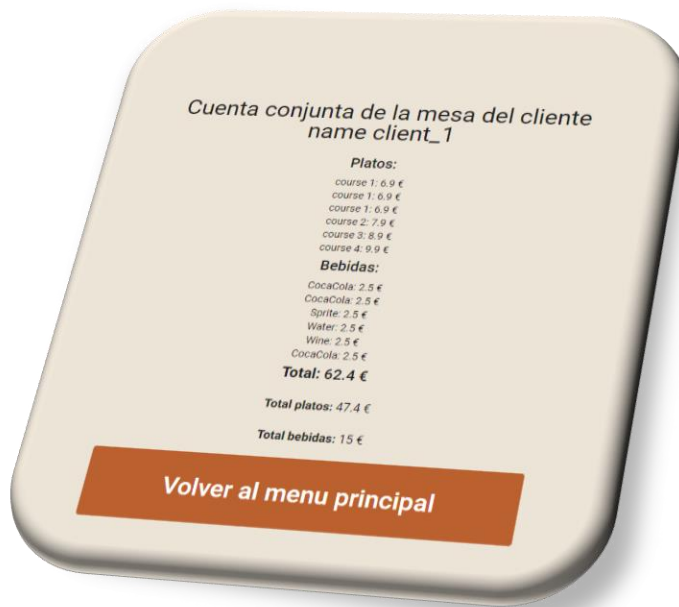


Figura 15. Factura de la mesa completa

- **Test de Concienciación y valoración:** En este último botón, se espera y se aconsejará a todos los clientes que pasen por él, ya que es uno de los motivos principales de Alafia. Con las respuestas de los clientes, se pretende extraer estadísticas para conocer el impacto social movimientos como este en la sociedad actual. Al pulsar el botón, se muestra una ventana modal emergente, en el que le aparecen al usuario una serie de preguntas, a las que el usuario responderá, normalmente tras la experiencia completa, tras conocer un poco más estas culturas africanas. La ventana mostrada es parecida a la que se muestra en la **Figura 16**.

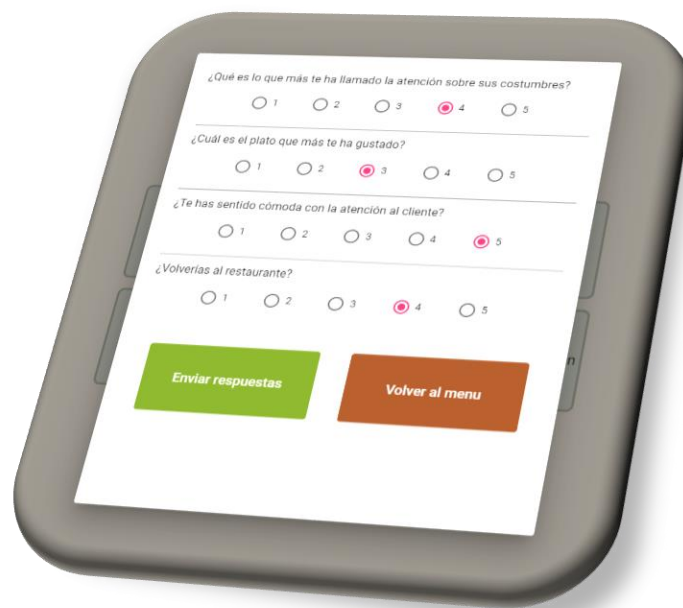


Figura 16. Test de concienciación y valoración

Tras responder las preguntas, el usuario tiene de nuevo dos opciones, enviar las respuestas, por lo que se guardarían en la base de datos de Alafia para su posterior estudio para la medición del impacto social, o pulsar el botón rojizo, que, de nuevo, devolvería al usuario a la pantalla del menú principal.

Esas son las funcionalidades que un usuario puede desencadenar mediante la pulsación de botones en la aplicación web, pero hay más procesos que suceden, aunque el usuario no haga nada.

La otra parte fundamental de la aplicación es la del jefe de sala. Este dispondrá de ciertas ventanas a las que solo puede acceder mediante un par usuario contraseña definida para este rol. Para ello, existe una ruta en la aplicación “/login”, que muestra una ventana donde el jefe de sala deberá introducir un email y una contraseña y pulsar sobre el botón “Login” para poder autenticarse. La ventana la vemos ilustrada en la **Figura 17**.



Figura 17. Pantalla de login

Una vez autenticado el jefe de sala, se le muestra la pantalla de la **Figura 18**.



Figura 18. Pantalla del jefe de sala

En esta ventana, al jefe de sala se le muestra una cuadrícula formada por casillas, las cuales contienen información de las mesas con reserva que tiene ese día el restaurante.

De cada mesa se muestra el id único de la mesa y el nombre del cliente que hizo la reserva. Cada casilla es seleccionable pulsando sobre ella. Cuando una mesa es seleccionada, se abre una venta modal emergente, donde se le muestran al jefe de sala información de cada comensal de la mesa, tal y como se muestra en la **Figura 19**.



Figura 19. Información de la mesa seleccionada por el jefe de sala

Al igual que con las mesas, se muestra información de cada comensal en forma de cuadrícula, y en cada casilla se muestran los siguientes datos del comensal: el id único del cliente, su nombre, un listado con los platos y las bebidas que ha pedido. Además, por cada plato se muestra su estado, si ya ha sido servido o no. Este estado es pulsable por el jefe de sala, y al pulsar sobre él, provoca un cambio en el estado del plato. Si estaba en estado “En espera”, el plato se pasa al estado “Servido” y viceversa, para poder gestionar posibles errores humanos del jefe de sala. Estos estados los podemos ver en la **Figura 20**.



Figura 20. Estados de un plato

Volviendo a la parte de la aplicación destinada a los clientes, estando en el menú principal un usuario, significa que ese usuario está identificado en una Tablet, y que además todos los comensales de la mesa a la que pertenece están en la misma situación, a la espera de que les sirvan las bebidas, los platos de comida, una petición de un usuario para un gestor de experiencia o para la recogida de la factura.

En el caso de que alguno de los comensales esté esperando la recepción de un plato, como acabamos de decir, este estará en el menú principal. Cuando ese plato esté listo para ser servido, al jefe de sala del restaurante se le notificará de que hay un plato que puede ser servido. En ese momento, el jefe de sala, le cambiará el estado de “En espera” a “Servido”.

Este cambio de estado provoca que en la Tablet del usuario al que pertenece el plato, se lance un video contando información de interés acerca de ese plato, su historia, ingredientes...

Como digo, esto de nuevo conlleva un problema de sincronización, del que se habla en el siguiente capítulo. La pantalla que se le muestra al comensal es la ilustrada en la **Figura 21**.



Figura 21. Pantalla experiencia visual

Como vemos, se muestra un video en la pantalla, pausable y rebobinable en alta resolución, para que el usuario pueda revisualizar las veces que quiera el contenido. Cuando el usuario quiera volver al menú principal para seguir gestionando su experiencia en el restaurante, tiene disponible que le devuelve al menú principal con las 4 opciones anteriormente mencionadas.

Elección de tecnologías

Teniendo en cuenta la preselección de tecnologías hecha en el estudio de las disponibles actualmente en el mercado para el desarrollo de una aplicación web, toca escoger las más viables para el desarrollo de este proyecto.

Empecemos por el Front End.

Aunque para el propósito de Alafia cualquiera de las tres tecnologías expuestas anteriormente, React, Angular y Flutter, vamos a escoger Angular, ya que su principal cometido es estandarizar mediante el uso de módulos los desarrollos del Front End. Lo cual me parece un punto muy a favor de esta tecnología. Además, cuenta con una comunidad muy grande en las redes ya que su popularidad está en auge, y la facilidad de trabajo al usar metodologías más estándares es vital para los desarrolladores como yo que no tenemos experiencia creando aplicaciones web.

Pasemos a la elección de la tecnología de Back End.

En este apartado, haremos una poda inicial sobre las tres tecnologías elegidas, Node.js, Spring Boot y Ruby on Rails. Vamos a eliminar esta última, ya que su cometido es hacerlo todo (Front y Back End) ya que con el desarrollo de este proyecto se busca el uso de varias tecnologías para su aprendizaje entre otras cosas. Por tanto, la elección se hace sobre Node.js y Spring Boot. Ambas dos nos sirven para desarrollar un servidor que haga de interfaz para la base de datos, pero ya que en el grado se hace más hincapié sobre el lenguaje Java que sobre JavaScript, vamos a elegir el framework que trabaja sobre Java, es decir, Spring Boot.

Por último, la elección de la base de datos.

Se han expuesto dos muy diferentes en el apartado de análisis de bases de datos disponibles en el mercado, una base de datos relacional, el modelo tradicional, y otra base de datos no relacional, más novedosa. El modelo de datos de Alafia no es un modelo complejo como tal, lo veremos más a detalle en el capítulo [Base de datos \(MongoDB\)](#). La comunicación con el cliente, el Front End se hace mediante el uso de objetos JSON. Por tanto, y ya que no requerimos de transacciones complejas, y el modelo de datos no se ciñe a un esquema cerrado, sino que está abierto a posibles objetos ligeramente diferentes en estructura, pero conteniendo información parecida, escogemos MongoDB para la capa de persistencia de la aplicación.

Base de datos (MongoDB)

Aunque MongoDB sea una base de datos no relacional, no significa que no se puedan relacionar datos en su almacenamiento. El modelo de Alafia es un modelo parecido a un árbol, ya que cada nodo tiene hijos relacionados con el padre.

Para representarlo visualmente, veamos el diagrama Entidad Relación simplificado del modelo en la **Figura 22**.

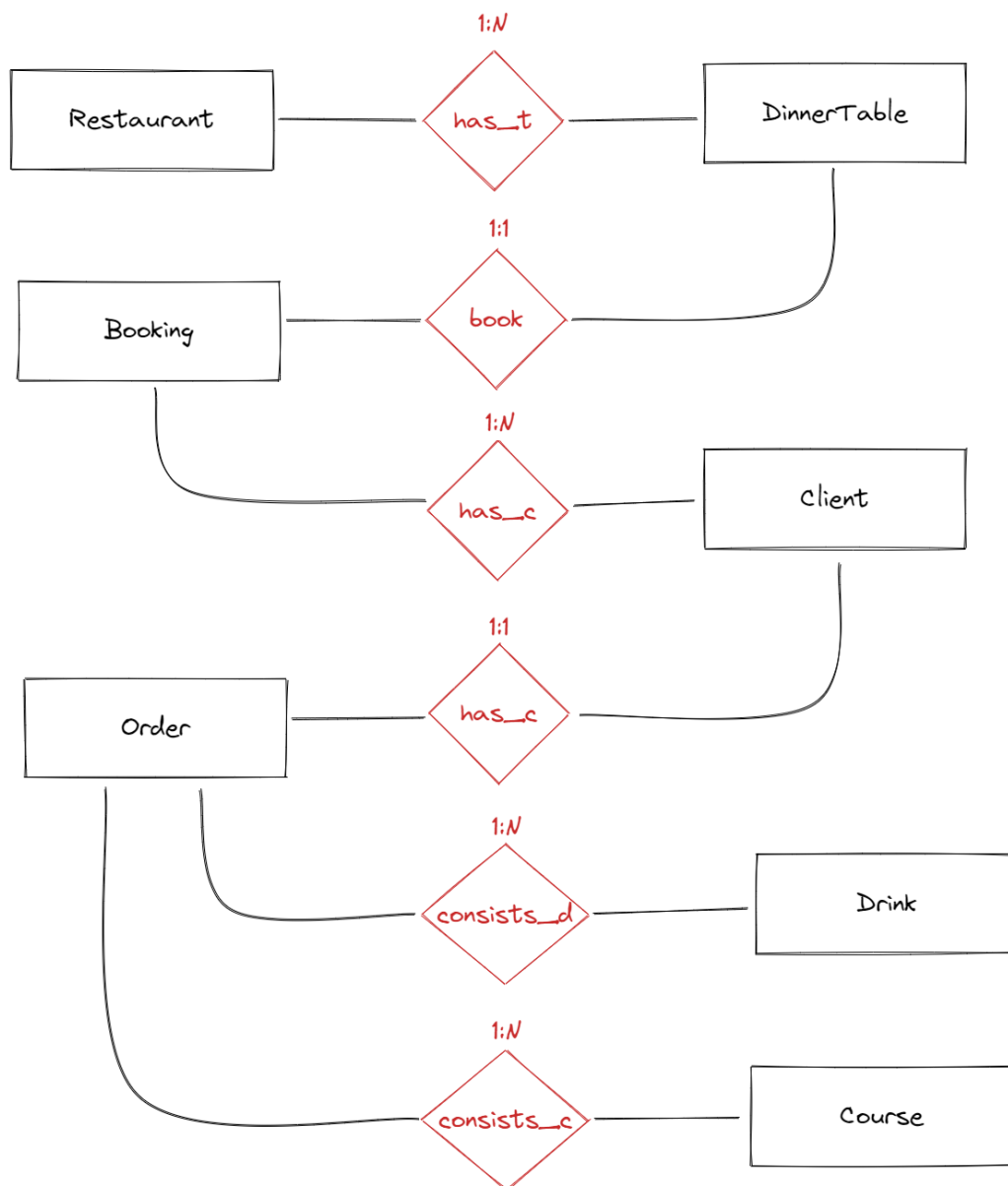


Figura 22. Diagrama Entidad Relación Alafia

Como podemos observar, todo parte de un documento padre, que es Restaurant. Este documento es necesario modelarlo ya que, de cara al futuro, contemplo una posible expansión de Alafia a más de un restaurante, y por tanto es necesario el modelado de dicha información.

La estructura por tanto es, explicada de palabra, la siguiente. Un restaurante (**Restaurant**), contiene mesas (**Table**). Las mesas solo están en un restaurante, y un restaurante puede tener muchas mesas, por tanto, relación 1: N. De cada restaurante se guarda su id único y las mesas que contiene, al igual que de cada mesa, únicamente su id único y el cliente que la reserva.

La mesa se reserva por un cliente que realiza la reserva (**Booking**), y no interesa mantener el histórico de las mesas reservadas por los clientes, por lo que con una relación 1:1 nos vale.

Cada cliente que realiza una reserva, la hace para él y para sus acompañantes (**Client**), que pueden ser uno, contándose a sí mismo o varios, lo cual nos deja una relación 1: N. De los clientes que hacen la reserva se guarda un id único, además de la información que se guarda de todos los clientes, como la de sí mismo como cliente. De los clientes se guarda un id único por cliente, su nombre y su correo electrónico en primera instancia. Se guardan además su comanda personal, sus respuestas al test de medición del impacto social y si ha confirmado o no su asiento en la mesa.

A cada cliente se le asigna una comanda (**Order**) que es única y efímera durante la estancia del cliente en el restaurante. Por tanto, relación 1:1. Y de cada comanda se guarda su identificador único, los platos contenidos durante la estancia, los platos que se le han servido ya, y las bebidas que ha pedido el cliente.

Cada cliente puede pedir varios platos y bebidas, o no pedir nada, por tanto, relaciones 0: N. de cada plato (**Course**) se guarda su identificador único, una descripción del plato, su precio, los valores nutricionales, información de cómo se obtienen los ingredientes y la historia de ese plato. Y de cada bebida (**Drink**) se guarda un identificador único, la descripción y el precio.

Para las siguientes secciones del documento, se va a trabajar con unos datos mockeados, es decir, datos introducidos con anterioridad para simular una situación real de una cena en Alafia. Para mostrar dichos datos, extraigo de la base de datos el nodo padre, la entidad Restaurante, lo que me dará la base de datos al completo. Estos son, por tanto, los datos con los que vamos a ir haciendo pruebas a lo largo del documento.

```
[
  {
    "id": "61ee50e61d881d753dfacbd8",
    "dinnerTables": [
      {
        "id": "61ee50e61d881d753dfacbd8",
        "booking": {
          "id": "61ee50e61d881d753dfacbd6",
          "client": {
            "id": "61ee50e61d881d753dfacbd0",
            "name": "name client_1",
            "mail": "mail client_1",
            "order": {
              "id": "61ee50e51d881d753dfacbcc",
              "courses": [
                {
                  "id": "61ee50e51d881d753dfacbc8",
                  "description": "course 1",
                  "price": 6.9,
                  "nutritionalValues": "nutritional values course 1",
                  "obtainProcess": "obtain process course 1",
                  "history": "history course 1",
                  "urlVideo": "-Hi8uEbGk_4"
                }
              ],
              "coursesIdServed": [
                {
                  "id": "61ee50e51d881d753dfacbc4",
                  "description": "CocaCola",
                  "price": 2.5
                }
              ]
            },
            "confirmed": false
          },
          "diners": [
            {
              "id": "61ee50e61d881d753dfacbd0",
              "name": "name client_1",
              "mail": "mail client_1",
              "order": {
                "id": "61ee50e51d881d753dfacbcc",
                "courses": [
                  {
                    "id": "61ee50e51d881d753dfacbc8",
                    "description": "course 1",
                    "price": 6.9,
                    "nutritionalValues": "nutritional values course 1",
                    "obtainProcess": "obtain process course 1",
                    "history": "history course 1",
                    "urlVideo": "-Hi8uEbGk_4"
                  }
                ]
              }
            }
          ]
        }
      ]
    }
  ]
]
```

```

        "coursesIdServed": [

        ],
        "drinks": [
            {
                "id": "61ee50e51d881d753dfacbc4",
                "description": "CocaCola",
                "price": 2.5
            }
        ]
    },
    "confirmed": false
},
{
    "id": "61ee50e61d881d753dfacbd1",
    "name": "name client_2",
    "mail": "mail client_2",
    "order": {
        "id": "61ee50e61d881d753dfacbcd",
        "courses": [
            {
                "id": "61ee50e51d881d753dfacbc8",
                "description": "course 1",
                "price": 6.9,
                "nutritionalValues": "nutritional values course 1",
                "obtainProcess": "obtain process course 1",
                "history": "history course 1",
                "urlVideo": "-Hi8uEbGk_4"
            }
        ],
        "coursesIdServed": [

        ],
        "drinks": [
            {
                "id": "61ee50e51d881d753dfacbc4",
                "description": "CocaCola",
                "price": 2.5
            },
            {
                "id": "61ee50e51d881d753dfacbc5",
                "description": "Sprite",
                "price": 2.5
            },
            {
                "id": "61ee50e51d881d753dfacbc6",
                "description": "Water",
                "price": 2.5
            },
            {
                "id": "61ee50e51d881d753dfacbc7",
                "description": "Wine",
                "price": 2.5
            }
        ]
    }
},

```

```

    "confirmed": false
  },
  {
    "id": "61ee50e61d881d753dfacbd2",
    "name": "name client_3",
    "mail": "mail client_3",
    "order": {
      "id": "61ee50e61d881d753dfacbc9",
      "courses": [
        {
          "id": "61ee50e51d881d753dfacbc8",
          "description": "course 1",
          "price": 6.9,
          "nutritionalValues": "nutritional values course 1",
          "obtainProcess": "obtain process course 1",
          "history": "history course 1",
          "urlVideo": "-Hi8uEbGk_4"
        },
        {
          "id": "61ee50e51d881d753dfacbc9",
          "description": "course 2",
          "price": 7.9,
          "nutritionalValues": "nutritional values course 2",
          "obtainProcess": "obtain process course 2",
          "history": "history course 2",
          "urlVideo": "AEvclAvvMcI"
        },
        {
          "id": "61ee50e51d881d753dfacbca",
          "description": "course 3",
          "price": 8.9,
          "nutritionalValues": "nutritional values course 3",
          "obtainProcess": "obtain process course 3",
          "history": "history course 3",
          "urlVideo": "-Hi8uEbGk_4"
        },
        {
          "id": "61ee50e51d881d753dfacbcb",
          "description": "course 4",
          "price": 9.9,
          "nutritionalValues": "nutritional values course 4",
          "obtainProcess": "obtain process course 4",
          "history": "history course 4",
          "urlVideo": "AEvclAvvMcI"
        }
      ]
    },
    "coursesIdServed": [

  ],
  "drinks": [
    {
      "id": "61ee50e51d881d753dfacbc4",
      "description": "CocaCola",
      "price": 2.5
    }
  ]
]

```

```

    },
    "confirmed": false
  },
  {
    "id": "61ee50e61d881d753dfacbd3",
    "name": "name client_4",
    "mail": "mail client_4",
    "order": {
      "id": "61ee50e61d881d753dfacbcf",
      "courses": [

      ],
      "coursesIdServed": [

      ],
      "drinks": [

      ]
    },
    "confirmed": false
  }
]
},
{
  "id": "61ee50e61d881d753dfacbd9",
  "booking": {
    "id": "61ee50e61d881d753dfacbd7",
    "client": {
      "id": "61ee50e61d881d753dfacbd3",
      "name": "name client_4",
      "mail": "mail client_4",
      "order": {
        "id": "61ee50e61d881d753dfacbcf",
        "courses": [

        ],
        "coursesIdServed": [

        ],
        "drinks": [

        ]
      },
      "confirmed": false
    },
    "diners": [
      {
        "id": "61ee50e61d881d753dfacbd4",
        "name": "name client_5",
        "mail": "mail client_5",
        "order": {
          "id": "61ee50e61d881d753dfacbce",
          "courses": [
            {
              "id": "61ee50e51d881d753dfacbc8",

```



```

        "description": "course 1",
        "price": 6.9,
        "nutritionalValues": "nutritional values course 1",
        "obtainProcess": "obtain process course 1",
        "history": "history course 1",
        "urlVideo": "-Hi8uEbGk_4"
    },
    {
        "id": "61ee50e51d881d753dfacbc9",
        "description": "course 2",
        "price": 7.9,
        "nutritionalValues": "nutritional values course 2",
        "obtainProcess": "obtain process course 2",
        "history": "history course 2",
        "urlVideo": "AEvc1AvvMcI"
    },
    {
        "id": "61ee50e51d881d753dfacbca",
        "description": "course 3",
        "price": 8.9,
        "nutritionalValues": "nutritional values course 3",
        "obtainProcess": "obtain process course 3",
        "history": "history course 3",
        "urlVideo": "-Hi8uEbGk_4"
    },
    {
        "id": "61ee50e51d881d753dfacbc4",
        "description": "course 4",
        "price": 9.9,
        "nutritionalValues": "nutritional values course 4",
        "obtainProcess": "obtain process course 4",
        "history": "history course 4",
        "urlVideo": "AEvc1AvvMcI"
    }
],
"coursesIdServed": [

],
"drinks": [
    {
        "id": "61ee50e51d881d753dfacbc4",
        "description": "CocaCola",
        "price": 2.5
    }
]
},
"confirmed": false
},
{
    "id": "61ee50e61d881d753dfacbd5",
    "name": "name client_6",
    "mail": "mail client_6",
    "order": {
        "id": "61ee50e61d881d753dfacbcf",
        "courses": [

```

```

    ],
    "coursesIdServed": [
        ],
    "drinks": [
        ]
    },
    "confirmed": false
  }
]
}
]
]

```

Como podemos ver en el Json autogenerado de la extracción de los datos precargados en la aplicación, se carga un único restaurante, con ID **61ee50e61d881d753dfacbd8**.

En el restaurante hay cargadas dos mesas, suficiente para comprobar todas las funcionalidades de la aplicación. En la mesa con ID **61ee50e61d881d753dfacbd8** vemos que la reserva tiene el id **61ee50e61d881d753dfacbd6** y que la ha realizado el cliente con id **61ee50e61d881d753dfacbd0** y con nombre **name client_1**.

Vemos también que la mesa está reservada para cuatro personas, pues el array de **dinners** tiene cuatro entradas. Por ejemplo, para el cliente que hizo la reserva **name client_1**, vemos que su pedido personal tiene el id **61ee50e51d881d753dfacbcc**.

Dentro del pedido, observamos que se compone, un array para los platos que tiene asociados, un array para indicar los IDs de los platos que ya se le han servido, y un último array para asociar las bebidas que ha solicitado el cliente.

Esta jerarquía se repite de pedido, se repite por cada cliente asociado a la mesa, y de igual manera, la estructura de datos de la mesa se repite para cada mesa del restaurante.

Front End (Angular)

El Front End de la aplicación es la capa encargada de mostrar al usuario la información que se quiere mostrar, tratando los datos para mostrarlos del modo que se quiera y maquetándolos para darles forma.

De por sí, esta capa cuenta con muy pocos datos, ya que se nutre de los datos que solicita al servidor, al Back End, que es la capa que tiene más control sobre la base de datos.

La obtención de estos datos se puede abordar de múltiples formas, pero hay dos grandes bloques bien diferenciados, mediante peticiones REST o mediante el uso de eventos. Obviamente, en ambos casos hay que establecer una comunicación con el Back End, pero esta comunicación puede ser síncrona o asíncrona. Dependiendo del flujo que se esté ejecutando y de la arquitectura de la funcionalidad, puede convenir más una que otra. La diferencia más notoria entre estos tipos de comunicaciones es básicamente que el llamante se quede bloqueado a la espera de la respuesta del llamado para manejarla, o no quedarse bloqueado y manejar la respuesta cuando quiera llegar.

En el caso de Alafia, la comunicación que se ha implementado es síncrona, ya que no se manejan procesos muy costosos computacionalmente y las respuestas son muy rápidas.

Hablemos ahora de las capas de las que se compone el Back End y de cómo está organizado.

Como dije en la presentación de Angular, se caracteriza por llevar un estándar de organización del código a los desarrollos de aplicaciones web, y para conseguirlo, Angular propone a sus usuarios organizar su código mediante el uso de componentes.

Cada componente debe encargarse de una funcionalidad, en este caso, de una pantalla, y se compone de tres archivos normalmente, un html, un ts y un css. En el fichero html, se encuentra el código html que maqueta la pantalla de ese módulo. En el fichero ts (TypeScript) se encuentra el código en TypeScript que se encarga de orquestar el flujo que requieran las acciones del usuario en la pantalla, como recogida de datos, envío de datos o la navegación a otra pantalla entre otras muchas cosas que se podrían hacer. Por último, el fichero css, es el fichero que da formato a la maquetación del html, dándole colores, posición dentro de la pantalla, formas, etc.

Antes de entrar en cada componente y su explicación, hay que exponer como se ha configurado la aplicación. Para ello, Angular nos proporciona un fichero, *package.json* en el cual podemos

indicarle diversos comandos para ejecutar, y donde le podemos indicar que dependencias externas tiene que importar. Nos aportan también un fichero *angular.json* en el cual podemos configurar cosas más específicas de la aplicación, tales como definir cuál es el directorio raíz de nuestro proyecto, cuál es el componente principal de la aplicación, y definir varios perfiles de ejecución.

Dicho esto, podemos empezar a desgranar la aplicación Angular en todas sus capas que la componen. Para apoyar visualmente las siguientes explicaciones, es conveniente mostrar una imagen de la estructura del proyecto. Se muestra en la **Figura 23**.

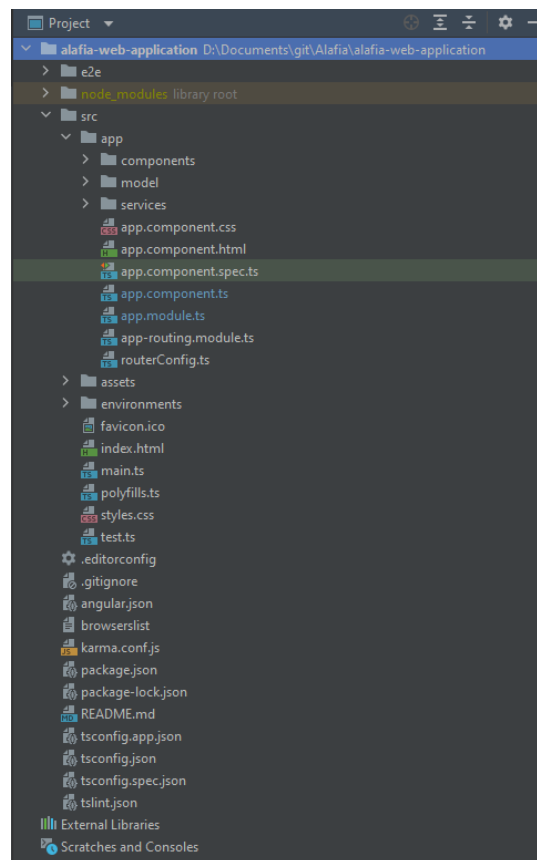


Figura 23. Jerarquía de archivos del Front End

La aplicación consta de dos archivos raíz, los cuales se encuentran dentro del directorio *src/* y son los archivos *main.ts* y *index.html*. En el primero de ellos se hace una configuración básica para que la aplicación arranque correctamente, y en el segundo, se hace una maquetación muy básica de nuevo, en la cual se establece el lenguaje del documento, se establece el icono de la aplicación y se importan ciertos iconos. Además de insertar el componente más importante de la aplicación llamado *app-root*.

Para realizar esta inserción, hay que realizar varios pasos. El primero es tener el componente a insertar al menos declarado. Para ello podemos hacerlo de manera manual, aunque angular nos proporciona herramientas para este cometido. En este caso, como se trata de la generación de un componente, podemos ejecutar `ng generate component <nombre del componente>` para generar un nuevo componente de la aplicación de forma automática. El resultado de este comando es la generación de un nuevo directorio en la ruta donde nos encontráramos cuando lanzamos el comando, y la inserción del nombre del componente generado en el fichero `app.module.ts` que es el fichero que recoge todos los módulos de los que está compuesta la aplicación.

Una vez existe el componente, en el fichero ts del mismo, vemos como tiene una anotación `@Component` la cual tiene las siguientes propiedades: `selector`, `templateUrl` y `styleUrls`. Son bastante autodescriptivas, y se rellenan con el nombre del componente que se le quiera dar, el cual va a ser el nombre a usar cuando se quiera insertar en cualquier parte de otro componente, el fichero con el código html que dará forma a lo que muestre el componente, y el fichero css que da estilo al html anterior.

Como acabo de decir, el atributo `selector` es el que da nombre al componente y es el atributo que se usa para importar el componente en otros. Para ello, basta con abrir una etiqueta con el valor del atributo `selector` del componente que se quiere insertar.

En el caso del fichero `index.html` se inserta el componente `app-root`, y se hace como hemos dicho insertando la siguiente línea en el fichero: `<app-root></app-root>`.

Entrando en el siguiente nivel de directorios, dentro del directorio `app`, nos encontramos una serie de directorios y unos archivos. Los podemos ver en la **Figura 23**.

El archivo `app.module.ts` es el encargado de recoger todos los componentes que se crean en la aplicación, de forma que podamos importarlos en cualquiera de los componentes que creamos. Además de los componentes, se importan también los módulos externos y el archivo de rutas de la aplicación.

El archivo de rutas es `routerConfig.ts` y como deja ver su propio nombre, es el encargado de definir que ruta muestra que componente. La definición de rutas de la aplicación angular queda definida por la **Tabla 1**.

Path	Component
/	WelcomeComponent
/index	AppComponent
/login	LoginComponentComponent
/book-table	BookTableComponent
/welcome	WelcomeComponent
/warm-up	WarmUpComponent
/replace-client	ReplaceClientComponent
/drinks	DrinksComponent
/wait-diners	WaitDinersComponent
/app-menu	AppMenuComponent
/experience-manager	ExperienceManagerComponent
/extras	ExtrasComponent
/migration-test	MigrationTestComponent
/experience	ExperienceComponent
/bill	BillComponent
/table-bill	TableBillComponent
/room-manager	RoomManagerComponent

Tabla 1. Rutas de la aplicación Angular

Por último, tenemos los tres ficheros del componente principal de la aplicación, el que hemos importado con anterioridad en el archivo *index.html*. Se trata del componente *app-root* que se define en los tres archivos que ya hemos mencionado que casi todo componente en angular tiene.

Antes de entrar en cada uno de los componentes, dentro del directorio *app*, nos encontramos con tres directorios, *components*, *model* y *services*. En *components*, encontraremos definidos todos los componentes de la aplicación, cada uno de ellos en su propio directorio y en cada subdirectorio de cada componente, nos encontraremos normalmente con los tres ficheros que ya hemos mencionado, componen un componente, un fichero *css*, un *html* y un *ts*, tal y como vemos en la **Figura 24**.

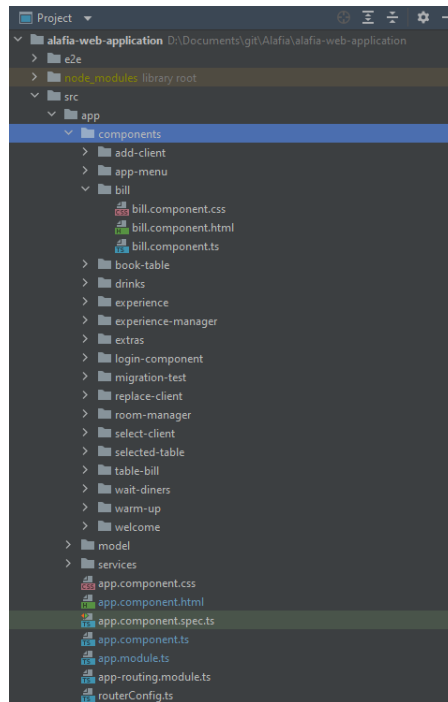


Figura 24. Árbol de ficheros directorio componentes

El siguiente directorio en orden que nos encontramos es *model*, y como su nombre nos indica, encontramos las definiciones del modelo de datos que usará la pantalla para mostrar sus datos. Además, encontramos un subdirectorio con nombre *dto*, siglas de *Data Transfer Object*, y dentro de este encontramos una serie de clases de definición de objeto que serán usadas para la comunicación entre el Front End y el Back End. Estos ficheros guardarán cierta similitud con los correspondientes a los del modelo de la aplicación, pero pueden añadir o ignorar ciertos campos en post de la comunicación. La jerarquía de este directorio queda reflejada en la **Figura 25**.

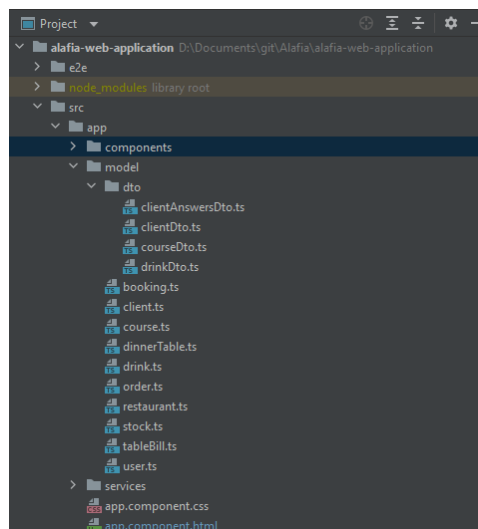


Figura 25. Directorio de modelo del Front End

Por último, el directorio `services` contiene los servicios del Front End. Un servicio podemos definirlo como una clase dedicada exclusivamente a una tarea específica. No tienen ninguna pantalla asociada, simplemente realizan las tareas que tienen definidas. En el caso del Front End de Alafia, dispone de dos servicios, *LoginService* y *DataService*, tal y como vemos en la **Figura 26**.

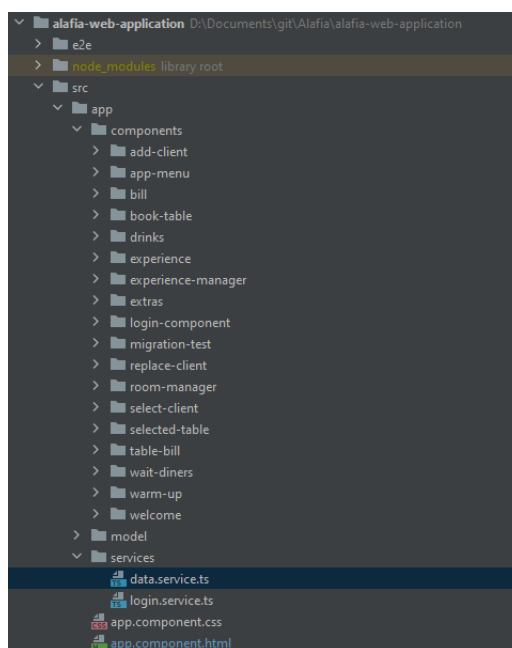


Figura 26. Servicios del Front End

El primero, *LoginService*, se encarga de hacer la validación de login en la aplicación. Es llamado en el componente *login-component*.

El segundo, *DataService*, se encarga de la gestión de los datos que se requieren guardar en memoria para el correcto funcionamiento de la aplicación y se encarga además de las comunicaciones con el Back End, ya que ambos cometidos tienen una muy estrecha relación.

Por último, el directorio `components`, en el cuál como hemos dicho anteriormente, se encuentran todos los componentes definidos en la aplicación. Veamos cada uno de ellos con detenimiento.

Componente AppComponent

Es el componente principal de la aplicación, y por eso es el único que no se encuentra dentro del directorio componentes. Se compone de los tres archivos de los que se compone normalmente un componente, un css, un html y un ts.

En su archivo css *app.component.css* se definen casi todos los estilos de la aplicación, ya que es el componente principal y todos los demás componentes van a ser “hijos” de este, recogen los estilos de *app-root*.

El archivo html *app.component.html*, está estructurado por una única sección, en la que, dependiendo del contexto de ejecución en la que estemos, se incluyen más o menos líneas de código para ocultar ciertas funcionalidades que sirven para debuggear cuando se está desarrollando. En el contexto de producción, se muestra simplemente el contenido del componente *router-outlet*, el cual nos permite mostrar el contenido de los diferentes componentes definidos por los diferentes paths en esta sección del componente principal. Si nos encontráramos en un contexto de ejecución de desarrollo, se muestra además del *router-outlet* un botón *Menú* que al pulsarlo nos muestra en el lateral de la pantalla un listado con todos los componentes de la aplicación, cada uno de ellos pulsable, para navegar directamente a ese componente, como digo, siempre por motivos de debug. Este listado es completamente configurable y para configurarla simplemente hay que irse a la sección de la clase *sidenav* del mismo fichero html *app.component.html* y añadir o quitar los componentes que están declarados de la siguiente forma:

```
<p (click)="onClick('welcome'); drawer.toggle()">Welcome</p>
```

Por último, el archivo *app.component.ts*, se encarga de varias cosas, entre otras, definir el nombre del componente con la propiedad *selector*, que como ya hemos dicho tiene el valor *app-root* para este componente. Se definen también el fichero html que le da forma y el fichero css que le otorga estilo al html, en las propiedades *templateUrl* y *styleUrl* respectivamente. Todo componente, en este archivo ts, tiene un constructor, mediante el cual se pueden inyectar dependencias de servicios que tengamos en la aplicación. En este caso se importan tres servicios, *LoginService*, *Router* y *DataService*. Los cuales se explicarán en mayor detalle en próximas secciones del documento. Además del constructor, se pueden declarar métodos, que pueden ser llamados para ser ejecutados desde el propio fichero ts si este tuviera una complejidad alta, para extraer funcionalidad por método, por ejemplo, o pueden ser llamados por alguna funcionalidad definida en el html. En este caso, para el modo de ejecución en debug,

se establece el comportamiento que tienen los elementos del listado de componentes que hemos mencionado antes. Simplemente se establece que cuando el usuario haga click en cualquiera de los componentes del listado, se navegue al componente que ha pulsado. Esto se consigue recibiendo como parámetro del método un String, que está definido en cada ítem del listado de componentes. Como vemos en fragmento de código anterior, se define que cuando se pulse sobre el párrafo, se llame al método *onClick* con el argumento *'welcome'*, ruta del componente *WelcomeComponent*. con lo cual en el método basta con ejecutar el siguiente código para navegar a ese componente:

```
this.router.navigateByUrl(page);
```

Componente WelcomeComponent

Es el componente que visualizarán los comensales cuando vean por primera vez el dispositivo que tendrán disponible en la mesa. Se encarga de dar la bienvenida al comensal en primera instancia, y permite al comensal confirmar o cambiar el cliente que reservó la mesa.

Además, al ser la primera pantalla, este componente se encarga en primera instancia de hacer la carga de datos que le lleguen desde el servidor. En el caso que no hubiera datos cargados, mostraría un mensaje indicando que la aplicación está a la espera de que los datos lleguen desde el servidor.

Tiene dependencias con el servicio *DataService*, para la carga de datos inicial, y con el ruteador, para la navegación a la siguiente pantalla, *warm-up*.

Componente WarmUpComponent

Es la segunda pantalla de la aplicación, y se mostraría al comensal cuando este confirme que este en la mesa en la que está el cliente que hizo la reserva inicial.

En esta pantalla, el comensal tendrá que seleccionar su nombre en un listado que se le mostrará, para configurarle la experiencia tecnológica personalizada. Si no se viera en el listado, se le proporciona un sistema de sustitución de comensal, para personalizarle su experiencia. Esta sustitución se lleva a cabo navegando al componente *ReplaceClientComponent*.

En ambos casos, tanto si el comensal se confirma en la Tablet como si selecciona sustituir a alguien, el comensal avanzará de pantalla y se le mostrará el componente *WaitDinersComponent*.

Componente *ReplaceClientComponent*

Se le muestra al comensal cuando este en la pantalla anterior decide que sustituye a un comensal que ya estaba configurado en la mesa. Por ello, se le pregunta e indica al comensal que a qué persona está sustituyendo, ya que heredará los platos que tenía el anterior configurados. Una vez seleccionado a quien sustituye, se le mostrará una ventana modal dirigida por el componente *AddClientComponent*.

Componente *AddClientComponent*

En esta pantalla, se le recogerán los datos personales, nombre y email al cliente que está sustituyendo a otro, con el fin de personalizarle su experiencia tecnológica.

Se le permite volver al componente anterior mediante la pulsación de un botón, o confirmar los datos para proseguir con el flujo de la aplicación. En cualquier caso, cuando el cliente termine de confirmar sus datos, se devolverá el control al componente anterior *ReplaceClientComponent*, que proseguirá en el flujo mostrando el siguiente componente, *WaitDinersComponent*.

Componente *WaitDinersComponent*

Este componente es de los más interesantes tecnológicamente, ya que nos encontramos con el primero de los problemas de sincronización de la aplicación que se querían resolver. En la sección **El Problema de la Sincronización** entraré más al detalle. Resumiéndolo mucho, se quiere una pantalla para sincronizar el estado de todos los comensales de la mesa, y se decide hacer en el momento en el que todos los comensales han confirmado su sitio en la Tablet, y por tanto se considera que cada cual está correctamente sentado en la mesa y configurado en el sistema.

Básicamente esta pantalla tiene dos posibles estados. El primero, cuando uno o varios comensales han confirmado su sitio, pero queda alguien sin confirmar. Se les muestra a los

comensales el siguiente mensaje: *Esperando a que los demas comensales confirmen su identificación en la Tablet.*

El segundo posible estado se produce cuando el ultimo comensal de la mesa que quedaba sin confirmar, confirma su sitio. Entonces automáticamente, todos los comensales navegan a la siguiente pantalla, mostrando el componente *DrinksComponent*.

Componente *DrinksComponent*

Este componente, como deja ver su nombre, es el encargado de gestionar las solicitudes de bebida de los comensales. Para ello, muestra en la Tablet un listado de las bebidas disponibles en el restaurante y un par de botones para confirmar la bebida seleccionada o para no añadir una bebida si no quisiera el comensal.

Al iniciarse el componente, se hace una carga de las bebidas disponibles en el restaurante desde la base de datos, lo cual implica una llamada a través del *DataService*. Y ocurre lo mismo cuando el comensal confirma que quiere añadir una bebida, se le modifica su pedido añadiendo una bebida y se persiste en la base de datos.

Al pulsar cualquiera de los dos botones, confirmando una bebida nueva o no añadiendo ninguna, el usuario avanza a la siguiente pantalla, controlada por el componente *AppMenuComponent*.

Componente *AppMenuComponent*

Este es el componente que proporciona más funcionalidades al usuario, ya que le permite hacer múltiples operaciones y tiene el control sobre la pantalla que más tiempo debería estar visualizando un usuario normalmente.

Se trata del menu principal de la aplicación web. Se muestran cuatro botones, cada uno con su funcionalidad: *Bebidas*, *Gestor de Experiencia*, *Factura* y *Test de concienciación y valoración*.

Bebidas tiene mucha similitud con la pantalla del componente *DrinksComponent*, se podría decir que es incluso la misma pantalla, salvo por una pequeña diferencia, y es que, en este caso, se le muestra al usuario desde una ventana modal, de modo que, pulsando fuera de esta, el usuario puede volver al menu principal sin confirmar en ninguno de los botones que tiene disponible, *Añadir bebida* o *No Añadir bebida*.

Gestor de Experiencia lanzará una petición para que uno de los gestores de experiencia se acerque a la mesa para atender a la petición del usuario. Cuando esta solicitud esté en proceso, al usuario se le muestra un mensaje en la pantalla indicando que la solicitud está en proceso de ser atendida. Pulsando de nuevo sobre el botón *Gestor de Experiencia* la solicitud se cancela y el mensaje se elimina de la vista del usuario.

Factura mostrará una ventana modal que está controlada por el componente *ExtrasComponent* y al igual que las demás pantallas modales, se cerrará pulsando fuera de esta.

Por último, *Test de concienciación y valoración* muestra una ventana modal controlada por el componente *MigrationTestComponent*.

Componente *ExtrasComponent*

Este componente tiene el control sobre una de las últimas pantallas que visualizará el usuario. Se muestra en una ventana modal emergente desde el componente *AppMenuComponent* y actualmente muestra tres botones, dos relacionados con la factura de la mesa o del usuario individualmente y el último para volver al menú principal.

El primer botón con nombre *Pedir la cuenta*, hace navegar al usuario al componente *BillComponent*. El segundo botón con nombre *Pedir cuenta en conjunto* hace navegar al usuario al componente *TableBillComponent*.

Componente *BillComponent*

Desde este componente, el usuario visualizará en la pantalla su cuenta personal. Para ello, como el cliente para llegar hasta esta pantalla ya ha debido ser configurado en el sistema, el sistema tendrá identificado el usuario que está solicitando la cuenta. Por tanto, usando el servicio *DataService*, realizará una consulta al Back End para que le devuelva el estado actual de su cuenta personal.

Recordemos que el modelo de datos desde el nodo *Client* consta de información del cliente y además de la comanda del cliente, compuesta por platos y bebidas. Esta información se recoge del Back End y se hace un pequeño proceso para calcular la suma total y mostrarla por pantalla.

Por último, se pone a disposición del usuario un botón para volver al menú principal si el usuario quisiera seguir con la experiencia o si simplemente quería ver a modo informativo su cuenta personal.

Componente *TableBillComponent*

Muy parecida al componente *BillComponent*, muestra la cuenta, pero esta vez a nivel de la mesa en conjunto, es decir, de todos los comensales. El comportamiento es muy similar, se hace una llamada al Back End a través del servicio *DataService* para obtener todos los pedidos de la mesa configurada.

Si hacemos el mismo ejercicio que en el anterior componente y miramos el modelo, vemos como esta operativa es posible, ya que en cada instancia de *Table* se encuentra el *Booking*, usuario que hace reserva y en el cual se encuentra los *Client* que contienen la información de su pedido personal.

El Back End responde al Front End con un objeto DTO, con nombre *TableBill*, que contiene dos conjuntos, uno para el total de platos de la mesa, y otro para el total de bebidas de la mesa.

Al igual que en el componente *BillComponent*, se pone a disposición de los usuarios un botón para volver al menú principal si el usuario lo quisiera.

Componente *MigrationTestComponent*

Este componente tiene el control sobre la que debería ser la última pantalla que visualizan los usuarios, aunque se disponibiliza a los usuarios desde el menú principal en el componente *AppMenuComponent*.

La pantalla muestra unas cuestiones al usuario con el fin de obtener datos para extraer de ellos estadísticas del impacto social que ha tenido la experiencia de Alafia en el usuario. El usuario deberá responder a cada pregunta seleccionando un *radio button* de los que tiene disponibles, y cuando termine, podrá enviar las respuestas al Back End, usando de nuevo el servicio *DataService*, donde serán persistidas en la base de datos, o podrá volver al menú principal si el usuario considera que no ha terminado su experiencia en Alafia.

Componente ExperienceComponent

Este componente, al igual que *WaitDinersComponent*, es uno de los más interesantes tecnológicamente, ya que, de nuevo, implica sincronización.

Se lanzará por una petición manual del jefe de sala (room manager) cuando un plato se le sirva al usuario en la mesa. Con esa petición, la instancia de la aplicación web del usuario detectará la petición del jefe de sala para ese plato y automáticamente, la aplicación navegará hasta este componente.

Se le muestra entonces al usuario un video, con relación a su plato. Historia del plato, elaboración... información visual de interés acerca del plato. El usuario podrá volver a reproducir su contenido las veces que quiera, ya que, si no pulsa el botón que tiene disponible para volver al menú principal, no se le cambia de pantalla al usuario.

Componente LoginComponentComponent

Este componente no es conocido por los clientes de Alafia, ya que es un componente que solo está pensado para los empleados. Se trata de un componente que tiene la funcionalidad de hacer login en el sistema, para identificar al empleado, que normalmente será un jefe de sala.

Para ello, se le muestra al usuario un formulario con dos campos a rellenar, el nombre de usuario y su contraseña. Y un botón para confirmar los datos y hacer login.

Si las credenciales introducidas son correctas, se navega hacia el siguiente componente *RoomManagerComponent*, si no lo son, se muestra un mensaje indicando que el Login ha fallado.

Componente RoomManagerComponent

Este componente es el que hace posible la gestión en tiempo real de cada una de las mesas del restaurante, y al igual que el anterior componente, estará disponible únicamente para el personal, en concreto para el jefe de sala.

Se le mostrará en la pantalla una matriz con todas las mesas que han sido configuradas para el restaurante. En cada mesa, se indicará el id único de la mesa y el nombre del cliente que hizo la reserva. Cada casilla de la matriz es pulsable, lo que provocará que se abra una ventana modal

controlada por el componente *SelectedTableComponent*. Desde el cuál, el jefe de sala podrá realizar diferentes acciones que se detallarán en la explicación del componente.

Componente *SelectedTableComponent*

Como hemos dicho en el componente anterior, este componente muestra al jefe de sala información de la mesa que haya seleccionado. Indicará el id de la mesa e información de los comensales que están configurados en la mesa.

Por cada comensal se muestra el id del cliente, su nombre, sus platos y sus bebidas. Y he aquí una de las funciones más importantes del jefe de sala. Recordemos que en la explicación del componente *ExperienceComponent*, se lanzaba automáticamente cuando el plato se le sirve al comensal por una acción del jefe de sala. Pues bien, es justo esta. Del listado de platos que se le muestran al jefe sala, se muestra también el estado en el que se encuentra, *En espera* o *Servido*. El jefe de sala puede cambiar el estado pulsando sobre este. Cuando se produce el cambio de estado de *En espera* a *Servido*, al usuario que le pertenece el plato al que se le cambia el estado, se le muestra el componente *ExperienceComponent*.

Back End (Spring Boot)

El Back End de una aplicación web es la parte que no se ve de la aplicación, es el servidor propiamente dicho. Y Como tal, se encarga de orquestar la comunicación entre las diferentes capas de la aplicación. En el caso de Alafia, disponemos de una instancia de Base de Datos, una instancia del servidor (Back End), y N instancias de la aplicación web (Front End), una por cada usuario en el restaurante.

Para desarrollar el servidor, se ha optado por la tecnología Spring Boot, ya que es una de la más potentes y más usadas en el mercado.

Al igual que hicimos con el capítulo de **Front End**, vamos a ir desgranando la aplicación en sus diferentes capas que la componen.

Lo primero, definir que es Spring Boot. Ya sabemos que Spring es un framework de Java que nos facilita el trabajo bastante, nos brinda su capacidad de inyectar dependencias fácilmente. ¿Y Spring boot entonces que es? Es lo mismo, pero con el añadido de que nos facilita la vida todavía más ya que nos permite crear aplicaciones que están preparadas para ejecutarse de por sí solas, sin dependencias de terceros. Nos ofrece diferentes servlets sobre los que ejecutar la aplicación como Tomcat o Jetty, configura las dependencias que puede de Spring y de terceros automáticamente siempre que pueda, brinda a la aplicación de funcionalidades muy útiles como métricas, health checks, configuración externalizada... Como vemos, facilita mucho el trabajo y se le permite al usuario empezar a programar más rápido ya que se abstrae bastante de configuraciones y demás.

Habiendo introducido brevemente la tecnología que nos brinda Spring Boot, vamos a hablar de como configurarlo. Como hemos dicho, una de las funcionalidades que nos da es la de gestionar dependencias de terceros. Para ello, Spring puede trabajar con diferentes gestores de dependencias, como por ejemplo Maven o Gradle.

Gestor de dependencias

Maven es una herramienta de gestión de proyectos de desarrollo, que se usa principalmente para desarrollos que usan el lenguaje Java y usa los conceptos originarios de Apache Ant (automatización del proceso de compilación y construcción del proyecto). Su configuración se

basa en un fichero XML en el cual se indican tanto las dependencias que se requieren para el proyecto, como los requerimientos que son necesarios para la construcción del proyecto.

Por otro lado, Gradle es una herramienta de automatización de compilación de código abierto que está basada en los conceptos que introduce Maven, mejorándolo en aspectos como:

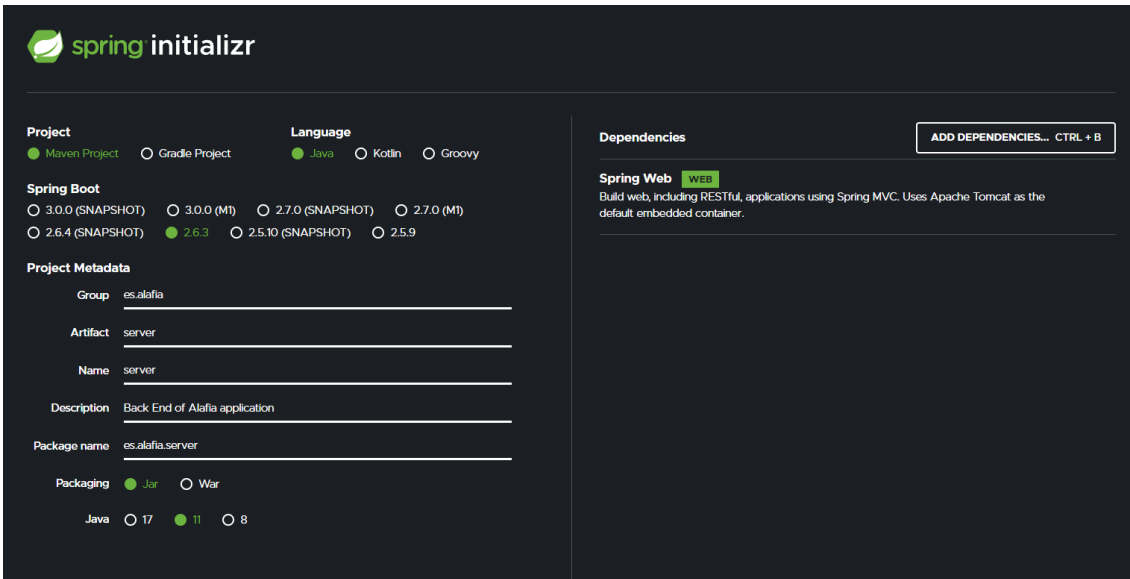
- El lenguaje de programación, pues no usa XML, sino que se basa en DSL, es decir, un lenguaje de programación que únicamente está dedicado a la resolución de un problema en particular, representarlo y proveer las técnicas para la correcta resolución en particular.
- La gestión del ciclo de vida del software, pues nos permite dar soporte a las diferentes fases del ciclo de vida, compilación, pruebas, implementación...

Por simplicidad en las configuraciones, tiempos de construcción y lenguaje de programación, se opta por la opción de Gradle para la configuración del Back End de Alafia.

Generación del proyecto de Spring

Una vez decididos por una tecnología de gestión, se puede proceder a la generación del proyecto. Para ello, Spring ofrece una herramienta para autogenerar un proyecto con el esqueleto de la aplicación ya montado (estructura de ficheros y directorios básica).

La herramienta es spring inicializr y tiene la forma que se muestra en la **Figura 27**.



The screenshot shows the Spring Initializr web application interface. It features a dark theme with a green Spring logo and the text 'spring inicializr'. The interface is divided into several sections: 'Project' with radio buttons for 'Maven Project' (selected) and 'Gradle Project'; 'Language' with radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'; 'Spring Boot' with radio buttons for various versions, including '2.6.3' (selected); 'Project Metadata' with input fields for 'Group' (es.alafia), 'Artifact' (server), 'Name' (server), 'Description' (Back End of Alafia application), and 'Package name' (es.alafia.server); 'Packaging' with radio buttons for 'Jar' (selected) and 'War'; and 'Java' version with radio buttons for '17', '11' (selected), and '8'. On the right, there is a 'Dependencies' section with a button 'ADD DEPENDENCIES... CTRL + B' and a list of dependencies, including 'Spring Web' (selected) with a description: 'Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.'

Figura 27. Spring Initializr

Como vemos, podemos generar proyectos tanto con Maven como con Gradle, le es indiferente a la herramienta. Debemos seleccionar también para que lenguaje va a ser generada, Java, Kotlin o Groovy, y la versión de Spring Boot que vayamos a utilizar. A continuación, debemos configurar la herramienta con los datos de nuestra aplicación, nombre del grupo, nombre del artefacto, descripción de la aplicación, nombre del paquete, el empaquetado Jar o War y la versión de Java. Por último, queda añadir las dependencias que requiera el proyecto. Toda esta configuración se usa para la generación del esqueleto del proyecto, pero por supuesto, una vez se haya generado todos estos valores son modificables manualmente, es decir, podemos refactorizar nombres de paquetes, de grupo o artefacto, al igual que podemos gestionar manualmente desde el fichero build.gradle (fichero de configuración del Gradle) las dependencias externas que se vayan a añadir.

Una vez generado el proyecto nos quedaría con la estructura mostrada en la **Figura 28**.

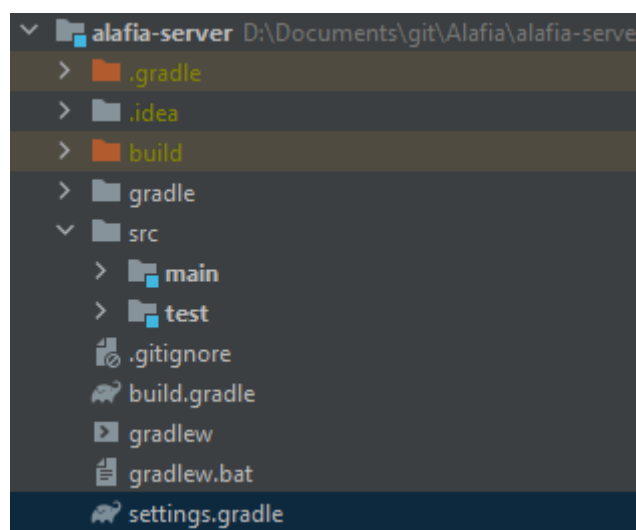


Figura 28. Estructura básica proyecto Back End

Fichero de configuración del proyecto Gradle

Para terminar con el fichero de configuración de Gradle, veamos su contenido:

```
plugins {
    id 'org.springframework.boot' version '2.2.7.RELEASE'
    id 'io.spring.dependency-management' version '1.0.9.RELEASE'
    id 'java'
}

group = 'es.alafia'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-data-mongodb'
    implementation 'de.flapdoodle.embed:de.flapdoodle.embed.mongo'
    implementation 'cz.jirutka.spring:embedmongo-spring:1.1'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    compile("io.springfox:springfox-swagger2:2.9.2")
    compile("io.springfox:springfox-swagger-ui:2.9.2")

    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'

    testImplementation 'net.javacrumbs.json-unit:json-unit-assertj:2.17.0'
    testImplementation('org.springframework.boot:spring-boot-starter-test') {
        exclude group: 'org.junit.vintage', module: 'junit-vintage-engine'
    }
}

test {
    useJUnitPlatform()
}
```

Como podemos ver, casi toda la configuración que habíamos seleccionado en la herramienta *Spring Initializr* se encuentra en este fichero. Centrándonos en la sección *dependencias*, en la cual encontramos todas las dependencias de Spring y externas de la aplicación, vemos como solo se añaden unas pocas, de las cuales destacar *spring-boot-starter-web*, para la exposición de un rest api para la comunicación con el Front End; las dependencias de mongo db, para la gestión

de la base de datos desde la aplicación de spring boot; las dependencias de swagger, para la generación automática de documentación enfocada en la parte del api expuesta; dependencias de lombok, para logs y generación de código; y junit, para tests.

Estructura de directorios del Back End

La aplicación Back End, está estructurada mediante directorios, tal y como vemos en la **Figura 28**. Todo parte de un nodo con nombre *alafia-server*, y en el cual encontramos los ficheros de configuración de Gradle, un directorio dedicado tambien para esto, el archivo *.gitignore* para evitar subir ficheros al repositorio indeseados y el directorio *src*, que contendrá el código del servidor.

Dentro del directorio *src*, nos encontramos con dos subdirectorios, *main* y *test*.

main contiene el código de producción del servidor, es decir, el código que se ejecutará cuando el servidor esté en ejecución. Mientras que *test* contiene el código que se usa para testear la aplicación de forma automática. Podemos ver su estructura en la **Figura 29**.

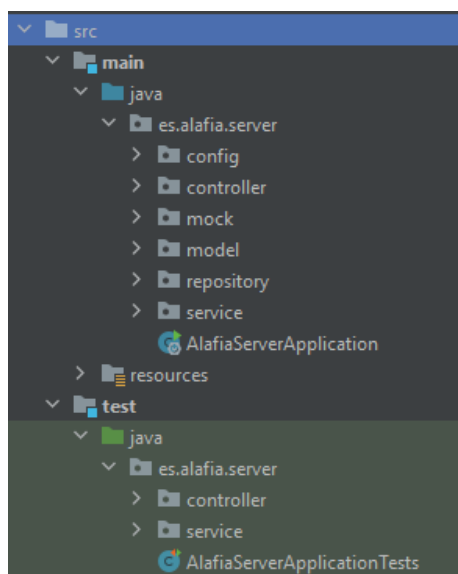


Figura 29. Estructura de ficheros src del servidor

Configuración del servidor

Antes de ver el código de producción, vamos a exponer como se ha hecho la configuración del servidor. Para ello, tenemos que entrar en el directorio *resources* del *main*, en el que encontramos un fichero *Application.yml*, que contiene toda la configuración y

parametrizaciones del servidor, en formato yml. En el caso del servidor de Alafia, no requiere mucha configuración, simplemente para configurar el acceso a la base de datos mongodb. Esa configuración quedaría de la siguiente forma:

```
mongodb:
  url: localhost
  db-name: embedded_db
```

Código de producción

Clase principal de la aplicación

Veamos ahora el código de producción. Como vemos, está estructurado por paquetes, dedicándose cada paquete a una determinada funcionalidad. El paquete raíz del servidor es *es.alafia.server* y contiene a los demás paquetes y a la clase principal *AlafiaServerApplication*.

AlafiaServerApplication es muy sencilla, y tiene una funcionalidad muy concreta al igual que las otras clases. Se encarga de arrancar la aplicación ejecutando también el contexto de ejecución de Spring. Para ello, basta con usar la anotación a nivel de clase *@SpringBootApplication* y con ejecutar la siguiente sentencia en el método *main* de la clase (el único que hay en todo el proyecto):

```
SpringApplication.run(AlafiaServerApplication.class, args);
```

Con esa sentencia, al arranque de la aplicación Spring arrancará su contexto de Spring, dejando expuesto un servidor Tomcat levantado en el puerto que hayamos configurado para el servidor. Arrancar el contexto de Spring implica varias cosas, tales como inyectar todas las dependencias para que puedan ser usadas en cualquier parte de la aplicación.

Es uno de los cometidos de Spring, brindar la opción de inyectar dependencias. Para ello, Spring se basa en el uso de *decorators* de java, es decir, anotaciones. Ya hemos visto el primer ejemplo de anotación en la clase *AlafiaServerApplication* que acabamos de mencionar.

Para que una clase se convierta en un artefacto inyectable, Spring hace uso del patrón de diseño software *singleton*, es decir, que solo se puede instanciar dicha clase una vez. Y para conseguirlo

de forma sencilla, *Spring* nos facilita anotaciones con las que podemos indicar que una clase ha de ser un *singleton* como es la anotación `@Component`. Anotando una clase con esta anotación, Spring cuando levanta su contexto de ejecución hace un barrido escaneando todas las clases y métodos que estén anotados, y levanta las instancias *singleton* de cada clase que esté anotada. Además de `@Component`, existen más anotaciones que hacen a Spring realizar las mismas acciones, pero además nos brindan semántica en la propia anotación. Por ejemplo `@Service`, nos indica que la clase que se anota es un servicio, como lo era en el Front End la clase *DataService*. `@Controller` nos indica que la clase va a ser una interfaz de comunicaciones vía API, pues expondrá normalmente unos endpoints para poder acceder al servidor para realizar las operativas que se expongan. `@Repository` nos indica que la clase va a ser el interfaz de comunicaciones con la base de datos. También se permite realizar la misma operativa a nivel de métodos, para ello tenemos que crear una clase que esté anotada con `@Configuration` y crear métodos anotados con `@Bean`. Normalmente solo se usa en estas situaciones, para crear configuraciones. Combinando todas estas anotaciones, se nos permite construir el servidor completo.

Packages que componen la aplicación

Las siguientes capas de la aplicación, como hemos dicho antes, se encuentran en su paquete correspondiente. Vamos a ir desgranando cada uno según el orden que van siendo ejecutados en un flujo normal. Para ello, lo primero sería arrancar el servidor, lo hacemos con la clase *AlafiaServerApplication* que acabamos de exponer.

Lo siguiente que se ejecuta como acabamos de explicar, es el escaneo de componentes para crear una instancia de cada uno en el contexto de Spring y que puedan ser inyectables en otros componentes.

Package config

En el servidor se crean dos clases de configuración, *MongoConfiguration* y *ServerConfig*. En la primera, como su nombre nos indica se configura la conexión de la base de datos MongoDB y para ello usa una nueva anotación que no habíamos contemplado hasta ahora. `@Value` nos permite extraer valores del fichero de configuración que esté leyendo la aplicación, en este caso recordemos que esa información se encuentra en el fichero *application.yml* que ya expusimos antes. Esto nos permite crear parametrizaciones, que pueden ser diferentes para cada entorno

de despliegue del servidor. Por ejemplo, la url de la base de datos del entorno de desarrollo seguramente no será la misma que la url de la base de datos del entorno de producción. Pues gracias a este tipo de anotaciones se nos permite realizar estas parametrizaciones de una manera muy sencilla, tal y como vemos en el siguiente fragmento de código:

```
@Value("${mongodb.url}")
private String mongoDbUrl;

@Value("${mongodb.db-name}")
private String mongoDbName;
```

En la otra clase de configuración, *ServerConfig*, se configura el servidor para habilitar CORS (Cross-Origin Resource Sharing), de forma que se permita el uso de los métodos http *HEAD*, *GET*, *PUT*, *POST*, *DELETE* y *PATCH* en el servidor. Y se configura también un método para la generación automática de documentación del api expuesta. Para ello, el servidor usa una funcionalidad de *swagger*, que nos permite generar automáticamente la definición del api expuesta. Para acceder a dicha documentación debemos irnos a un navegador y hacer una petición a la siguiente url: <http://localhost:8080/swagger-ui.html>.

Se nos mostrará una pantalla como la siguiente:

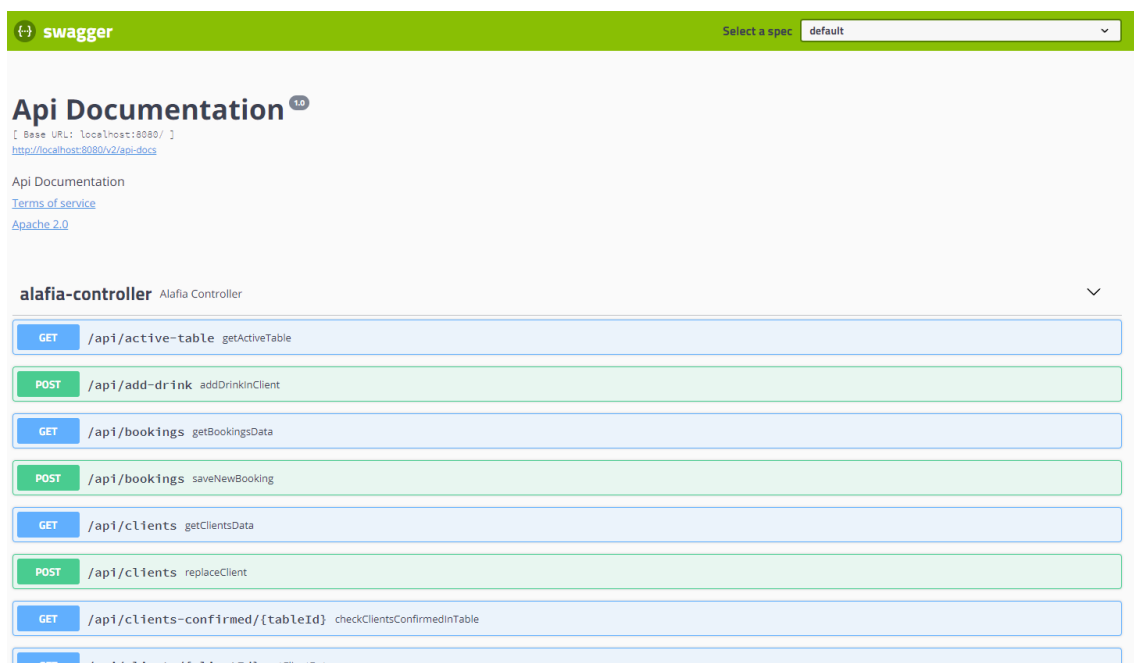


Figura 30. swagger-ui del servidor

En la que podemos ver todos los endpoints expuestos en el servidor, y los modelos con los que trabaja cada uno. Además, se permite hacer ejecuciones sobre cada endpoint pulsando sobre cualquiera de ellos, aparecerá un botón *try it out*, gracias al cual se nos permite lanzar la petición del endpoint seleccionado hacia el servidor, obteniendo así la respuesta. Es decir, nos sirve tanto como documentación como de herramienta de testing.

Package controller

Visto el paquete *configuration*, veamos *controller*, que como bien indica su nombre, contendrá las clases que actúen de controllers (exposición de api) del servidor. En nuestro caso solo disponemos de uno *AlafiaController*, ya que expone todos los endpoints relacionados con las operativas del restaurante. Otro enfoque que se podría haber llevado a cabo es la diferenciación de controladores por modelo, es decir, crear un controller específico para tratar con el modelo de restaurante, otro controlador para tratar con clientes, otro para pedidos, etc.

Spring nos ofrece gran ayuda a la hora de exponer un servicio al exterior, en forma de API en este caso. Para ello, nos basta con indicar a la clase que es un controller, anotándolo con *@RestController*, con eso spring lo detectará en el escaneo que hace al levantar el contexto y lo dejará preparado. Además, como es en nuestro caso, se configura también un base-path, es decir, un path de url que será común a todos los endpoint que se expongan desde este controller. Para ello, debemos añadir otra anotación *@RequestMapping* en la que además le indicamos en el argumento *value* el base-path del controller, y adicionalmente se le puede indicar también el tipo de datos que van a devolver los endpoint con el argumento *produces*, en nuestro caso quedaría así configurado:

```
@RequestMapping(value = "/api", produces = "application/json;charset=UTF-8")
```

Una vez tenemos el controlador anotado a nivel de clase correctamente, pasemos a las propiedades que declara el controller, en este caso dos, y son dos componentes a inyectar, *DataService* y *LoadInitData*. El primero tiene una funcionalidad pareja al *DataService* que vimos en la parte del Front, lo explicaremos en detalle más adelante, y *LoadInitData* se encarga de preparar y persistir los datos de prueba que vimos en la sección de Base de Datos de este documento. Para inyectar estos componentes, spring nos ofrece varias formas, las más habituales son inyectar las dependencias usando la anotación *@Autowired* en el componente a inyectar, o bien inyectarlo por constructor. En mi caso, por cuestiones de diseño se opta por este

último enfoque, quedando las primeras líneas del controller y de la mayoría de los componentes del servidor de esta forma:

```
@Slf4j
@RestController
@AllArgsConstructor
@RequestMapping(value = "/api", produces = "application/json;charset=UTF-8")
public class AlafiaController {

    private final DataService dataService;
    private final LoadInitData loadInitData;
```

Vemos dos anotaciones que no hemos comentado todavía, *@Slf4j* y *@AllArgsConstructor*. Ambas provienen de la librería *Lombok*, que nos ofrece ciertas funcionalidades a través del uso de anotaciones, tales como operaciones básicas para los pojo, generación de getters, setters, constructores, etc. En este caso, la primera de ellas nos ofrece un *log*, que de manera implícita nos añade timestamp y diversa información muy útil de la instancia desde la que se lanzó. La segunda anotación, hace referencia a un tipo de constructor con los que puede trabajar *lombok*, en este caso, como bien indica su nombre, generaría un constructor con todas las propiedades de la clase. Por tanto, como tenemos los dos componentes declarados, y estamos usando un constructor con todos los argumentos de la clase, estos se inyectan, quedando el controller con la disponibilidad de los métodos públicos de estos.

Lo siguiente que vemos en el controller, son los métodos públicos, que se corresponden con cada recurso que se expone en forma de endpoint hacia el exterior.

Vemos que cada método está anotado de forma que cada método del controller es un endpoint que expone el servidor hacia el exterior. Spring ofrece varios tipos de anotaciones de nivel de método para este cometido, y se corresponden con cada verbo HTTP que existen: GET, POST, PATCH, PUT, DELETE... En el caso del servidor, solamente se exponen métodos GET y POST, pero el resto de los métodos HTTP se trabajarían de forma muy similar.

Para los GET, Spring nos ofrece la anotación *@GetMapping*, en la que le indicaremos por argumento de la anotación el path con el que será invocado el método del controller. Si en este path, se indica alguna variable de path, esta debe ponerse entre llaves, indicando así que se trata de una variable de path. Esta variable ha de ser declarada en la firma del método java y el nombre debe coincidir con el indicado en el valor declarado en el path. Poniendo un ejemplo:

```

@GetMapping(value = "/clients-confirmed/{tableId}")
public boolean checkClientsConfirmedInTable(@PathVariable String tableId) {
    log.info("Checking status of clients for table {}", tableId);
    return dataService.checkClientsConfirmedInTable(tableId);
}

```

El path con el que accederíamos a este método sería con */clients-confirmed* seguido de */tableId*, es decir, con un ejemplo real con los datos preparados:

/clients-confirmed/61ee50e61d881d753dfacbd8

Comprobaría si todos los comensales de la mesa con id *61ee50e61d881d753dfacbd8* han confirmado su asiento, es decir, se han configurado su dispositivo para la experiencia tecnológica de Alafia.

De forma muy parecida, trabajamos con los métodos POST. La anotación cambiaría a *@PostMapping*, y al igual que con la anotación del GET, indicamos como argumento de la anotación el path por el que será invocado el método del controller. Poniéndolo en el contexto de un ejemplo, para añadir una bebida a un cliente, el endpoint que se declara es el siguiente:

```

@PostMapping(value = "/add-drink")
@ResponseStatus(HttpStatus.CREATED)
public Client addDrinkInClient(@RequestBody AddDrinkDTO addDrinkDTO) {
    log.info("Trying to add drink with id {} in client with id {}",
        addDrinkDTO.getDrinkId(), addDrinkDTO.getClientId());
    return dataService.addDrinkInClient(addDrinkDTO);
}

```

Vemos dos anotaciones adicionales, la primera *@ResponseStatus*. Es una anotación que se usa para indicar a Spring que el resultado satisfactorio de la ejecución del método invocado debe retornar el código HTTP, en este caso *CREATED 201*. La segunda anotación, *@RequestBody* sirve, para indicar el objeto de entrada hacia el método, en este caso, recibiría un *AddDrinkDTO*, que contiene información de la bebida que se añadirá al cliente, cuya información viaja también en este DTO (Data Transfer Object).

Package service

Visto el paquete *controller*, ya que no se declaran más controladores en el servidor, pasamos a exponer el paquete *service*, donde nos encontramos los servicios del servidor de Alafia. En este

caso, de nuevo, solo cuenta con una clase de servicio, *DataService*, que será bien pareja a la clase *DataService* del Front End. Para la declaración de servicios en Spring, basta con anotar la clase con *@Service*, con esto Spring es capaz de detectar el servicio, e inyectarlo en el contexto en el arranque del servidor.

Si nos fijamos en el contenido de los métodos del controller, vemos que su lógica es muy simple, loggean cierta información y realizan una llamada a este servicio para que realice la operación correspondiente. De esta forma, toda la lógica, por muy simple que sea, queda en manos del servicio. Y es precisamente lo que hace el servicio, exponer los métodos que son consumidos desde la capa del controlador del servidor, para el manejo de los datos.

Package repository

Para ello, el servicio debe tener accesibles los datos, y para ello, el servidor hace uso de interfaces *MongoRepository*, que nos facilitan los métodos CRUD (Create, Read, Update y Delete) de bases de datos, aunque siempre podemos añadir nuestras consultas a bases de datos customizadas haciendo uso de los *method names* de JPA (Java Persistence API). Estas interfaces están ubicadas en el paquete *repository* del servidor, y se declaran de la siguiente manera:

```
@Repository
Public interface RestaurantRepository extends MongoRepository<Restaurant, String> {
}
```

Como vemos, se trata de una simple interfaz que extiende a la interfaz *MongoRepository* y que está anotada con *@Repository*, para que se inyecte en el contexto de Spring en el arranque del servidor. *MongoRepository* necesita que le indiquemos que clase del modelo es la que va a tratar, en este caso con la clase *Restaurant*, y el tipo de dato del *Id* de la clase de modelo. En este caso *String* ya que el *Id* de la clase *Restaurant* es un *String*. Simplemente con esto, ya tenemos las operativas CRUD para el modelo de *Restaurant*.

Volviendo al servicio *DataService*, vemos como este declara que es un servicio, como bien dijimos antes con la anotación *@Service*. Pero vemos también que usa dos anotaciones provenientes de *Lombok*. *@Slf4j* que ya sabemos que se usa para logs, y *@RequiredArgsConstructor*, que es un tipo de constructor en el que solo añadirá las propiedades de clase que sean identificadas como *final*.

De nuevo, se usa la inyección de dependencias por medio de constructor con esta anotación, y se inyectan los repositorios de todos los modelos del servidor:

- *RestaurantRepository*
- *DinnerTableRepository*
- *BookingRepository*
- *ClientRepository*
- *OrderRepository*
- *CourseRepository*
- *DrinkRepository*

Y se irán invocando según vayan haciendo falta para el tratado correcto de los datos.

Package model

Por último, nos queda por exponer el paquete *model*, en el cual encontramos todos los modelos de datos con los que trabaja el servidor. Dentro de este paquete, vemos que están los modelos que se corresponden con los modelos de la base de datos, pero, además, dos paquetes más aparecen aquí, uno para almacenar las excepciones del servidor, y otro para almacenar las clases de DTO con las que se transferirán los datos por la red a través de las peticiones que reciba el controller.

Las clases de modelo de base de datos simplemente tienen las propiedades para modelar la información en la base de datos. Por ejemplo, viendo la clase *Client*:

```
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class Client {
    @Id
    private String id;
    private String name;
    private String mail;
    private Order order;
    private Boolean confirmed = false;
    private List<String> testAnswers;
}
```

Vemos que a nivel de clase solo tiene anotaciones de *Lombok*, algunas que no hemos explicado todavía.

@Data generará automáticamente el código equivalente a los getters, setters, constructor con propiedades finales de la clase, equivalente al uso de la anotación *@RequiredArgsConstructor*, un método *toString* y el método *equalsAndHashCode*.

@Builder nos generará el patrón *builder* para la construcción de instancias de la clase. Es muy cómodo y bajo mi punto de vista, le da un toque de estilo al código ya que no es necesario el uso de constructores ni setters para la construcción de los pojos como este. Un ejemplo de construcción de pojos usando esta anotación puede ser la siguiente:

```
Client.builder()  
    .name("name")  
    .mail("mail")  
    .build();
```

Por último, las anotaciones *@NoArgsConstructor* y *@AllArgsConstructor*, crearán constructores vacíos y con todas las propiedades de la clase en la que estén declarados, respectivamente.

Código de Test

Todo el código de producción del Back End ha sido testado no solo por las pruebas realizadas probando la aplicación en funcionamiento, es decir, integrándose con las llamadas recibidas por el Front End. Además, se han desarrollado Test unitarios y de Integración para el servidor.

Por tanto, podemos diferenciar los siguientes niveles de Test:

- Tests Unitarios
- Test de integración
- Test funcionales

Tests Unitarios

Su misión principal es la detección temprana de errores en las funcionalidades desarrolladas, en este caso, sin salir del servidor. Debido a esto, nos ahorrará mucho tiempo en los siguientes niveles de pruebas, ya que habremos corregido muchos errores gracias a estos tests. Nos garantizan la funcionalidad del código siempre que desarrollemos los test de manera correcta, lo que nos permite refactorizar el código sin el temor de variar su comportamiento, ya que esta asegurado con los tests.

El código sobre el que se ha decidido desarrollar test unitarios en el Back End es sobre la clase del controller, es decir, la clase que expone el api del servidor, y sobre la clase de servicio, que se encarga de operar con los datos y orquestar las llamadas a los repositorios mongo correspondientes que hacen de interfaz contra la base de datos MongoDB.

Para el desarrollo de estos test, uso la librería de testing Junit5, junto a Mockito para poder darle comportamiento simulado a ciertas piezas del servidor.

Casi todos los test a este nivel han sido desarrollados usando el mismo modus operandi. El siguiente fragmento de código muestra las primeras líneas de una clase de test, siendo las demas muy parecidas.

```
@ExtendWith(MockitoExtension.class)
public class AlafiaControllerTest {

    @Mock
    private DataService dataService;

    @InjectMocks
    private AlafiaController alafiaController;

    @Test
    void shouldCallDataServiceToRetrieveAllRestaurantsSavedInDB() {
        // Given
        when(dataService.retrieveRestaurantsData())
            .thenReturn(List.of(Restaurant.builder().build()));

        // When
        var restaurantsRetrieved = alafiaController.getRestaurantsData();

        // Then
        verify(dataService, times(1)).retrieveRestaurantsData();
        assertFalse(restaurantsRetrieved.isEmpty());
    }
}
```

Para poder usar los mocks de Mockito, hay que indicarle a la clase que se va a usar la extensión de MockitoExtension, tal y como vemos en el fragmento de código, con la anotación:

```
@ExtendWith(MockitoExtension.class)
```

A continuación, se definen las clases a las que se les quiere dar comportamiento, declarándolas como variables a nivel de clase. Como solo se van a usar en la propia clase de test, el modificador de acceso optimo es private. Y para indicarles que la clase no debe heredar su comportamiento, sino que le será dado por el propio test, se le añade la anotación:

```
@Mock
```

Después de la declaración de las clases a las que les queremos dar comportamiento en el propio Test, declaramos la clase que va a ser testeada, al igual que con las demás clases importadas, se declara con el modificador de acceso en private dado que solo es necesario su uso en la propia clase de test. A esta clase, hay que inyectarle los mocks declarados, para ello podemos hacerlo de varias formas.

Como la clase `AlafiaController` inyecta sus dependencias mediante el uso de constructor, podríamos hacer un método `BeforeAll`, que se ejecute nada más instanciar la clase de test, en el cual construyamos la clase a testear, en este caso `AlafiaController`, con todos los Mocks necesarios.

Otro método más elegante bajo mi observación es el uso de la anotación:

```
@InjectMocks
```

Que nos hará el mismo trabajo, pues nos inyectará todos los Mocks declarados en la clase de Test.

Una vez que tenemos todas las clases de Mock y la propia clase a testear declaradas, es la hora de escribir los Test como tal, que son métodos de la clase de Test, que van a comprobar una funcionalidad muy concreta de la clase a testear. Para ello, hay que anotar el método de test con la anotación:

```
@Test
```

El nombre del método puede ser cualquiera, pero las buenas prácticas dicen que ha de ser un nombre completamente descriptivo, de forma que con solo leer el nombre del método se sepa con exactitud lo que va a probar.

En el contenido de los métodos del Test, vamos a poder diferenciar normalmente tres bloques, que los podemos llamar `given`, `when` y `then`, tal y como muestro en el código anterior.

En la sección `given`, se suelen declarar todas las variables que se necesitaran para la ejecución del test, además de darle comportamiento a los mocks que sea necesario mediante el método `when` de Mockito.

En la sección `when`, se suele escribir la llamada a testear como tal. Es la ejecución a testear. Si el resultado de la ejecución produce alguna respuesta, esta se guarda en una variable declarada en esta misma sección.

Por último, en la sección `then`, es donde se comprueba que todo ha funcionado como se espera. Se verifican las llamadas a los `mocks`, con las firmas que se esperan, y se asegura que lo que ha devuelto la ejecución como resultado es lo esperado.

Tests de Integración

Su principal cometido es asegurar que todos los módulos que componen la aplicación se integren correctamente y realicen las funcionalidades esperadas correctamente. En el caso de la aplicación de Alafia, estos tests se han desarrollado mockeando (simulando) las llamadas del Front End, con el objetivo de asegurar que cuando se integre con el Back End, todo funcione según lo esperado.

La forma de los test de integración cambia ligeramente con respecto a los test Unitarios. En estos test, se evita el uso de `Mocks`, ya que se quiere comprobar el funcionamiento real de las piezas del servidor integradas. El planteamiento a seguir es, simular las llamadas REST que haría el Front End, para comprobar el correcto funcionamiento del Back End.

En el siguiente fragmento de código, veremos las principales características de un Test de Integración:

```

@EnableWebMvc
@AutoConfigureMockMvc
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes = AlafiaServerApplication.class)
class AlafiaServerApplicationTests {

    @Autowired
    private MockMvc mockMvc;

    @Autowired
    private ObjectMapper mapper;

    @Autowired
    private BookingRepository bookingRepository;

    @Autowired
    private DinnerTableRepository dinnerTableRepository;

    @Autowired
    private RestaurantRepository restaurantRepository;

    @Autowired
    private ClientRepository clientRepository;

    @Test
    void shouldCreateNewRestaurant() throws Exception {
        var restaurant = Restaurant.builder().build();

        var mvcResult = mockMvc.perform(MockMvcRequestBuilders
            .post("/api/restaurants")
            .contentType(MediaType.APPLICATION_JSON)
            .content(mapper.writeValueAsString(restaurant)))
            .andExpect(status().isCreated())
            .andReturn();

        assertNotNull(mapper.readValue(
            mvcResult
                .getResponse()
                .getContentAsString(),
                Restaurant.class)
            .getId());
    }
}

```

Como se puede observar, cambiamos la extensión de MockitoExtension, a SpringExtension, para indicar que la clase debe levantar el contexto de Spring porque vamos a trabajar con la aplicación en ejecución como si lo hiciéramos en una ejecución normal.

Con las anotaciones

```
@EnableWebMvc  
@AutoConfigureMockMvc
```

declaramos y configuramos un servidor Mock, con el cual simularemos al Front End, simulando las peticiones REST.

Y para terminar la cabecera de la clase del test de integración, con la anotación

```
@SpringBootTest(classes = AlafiaServerApplication.class)
```

Indicamos cual es la clase principal de la aplicación, para que se pueda ejecutar correctamente.

A continuación, como variables de clase, se declaran todos los Beans que se van a usar en el test, y se inyectan en este caso usando la anotación *@Autowired*.

Y, por último, al igual que con los test unitarios, quedan los métodos de test como tal, que se declaran exactamente igual. Su contenido sigue la misma estructura anteriormente explicada, con la diferencia de que en estos test nos se invoca la llamada al método como tal, sino que usando el servidor MockMVC simularemos la llamada al controller, punto de entrada para el Back End, y comprobaremos que hace las operaciones que debe realizar.

Tests Funcionales

Por último, los tests funcionales se aseguran de que la aplicación en conjunto trabaje correctamente y este libre de defectos, minimizando los riesgos que puedan surgir cuando la aplicación esté puesta en marcha y expuesta al público.

Puntos clave de sincronización

Como ya se ha mencionado previamente en este documento, hay varios puntos en los que la sincronización es muy importante en el flujo de la aplicación.

Sincronización a la hora de configurar todos los dispositivos para la experiencia personalizada Alafia

El primero de ellos se produce casi al principio del flujo. Por resumirlo de nuevo, el flujo que experimenta un cliente al llegar al restaurante es el siguiente:

- Llegan los comensales al restaurante
- El jefe de sala les indica cuál es su mesa (en el dispositivo está configurado el cliente que hizo la reserva)
- El cliente toma su asiento
- El cliente confirma que está en su mesa o selecciona el cliente que le hizo la reserva si estuviera otro configurado.
- El cliente confirma su asiento, quedando configurado el dispositivo para su experiencia personal o si está sustituyendo a alguien se configura con sus datos personales

En ese momento, al cliente se le muestra una pantalla de espera, indicándole de que debe esperar hasta que los demás acompañantes de la mesa confirmen su sitio, configurando un dispositivo para cada comensal. Por tanto, la instancia de la aplicación que está ejecutando el dispositivo, se mantiene en espera hasta que el último de los comensales que no se haya configurado en su dispositivo lo haga. Cuando este último comensal confirma su asiento, todos los dispositivos que estan asociados a la mesa en cuestión, se desbloquean y muestran la pantalla de bebidas, que es la siguiente en el flujo.

Para un mejor entendimiento, el diagrama de la **Figura 31** muestra este mismo flujo para una mesa de dos comensales:

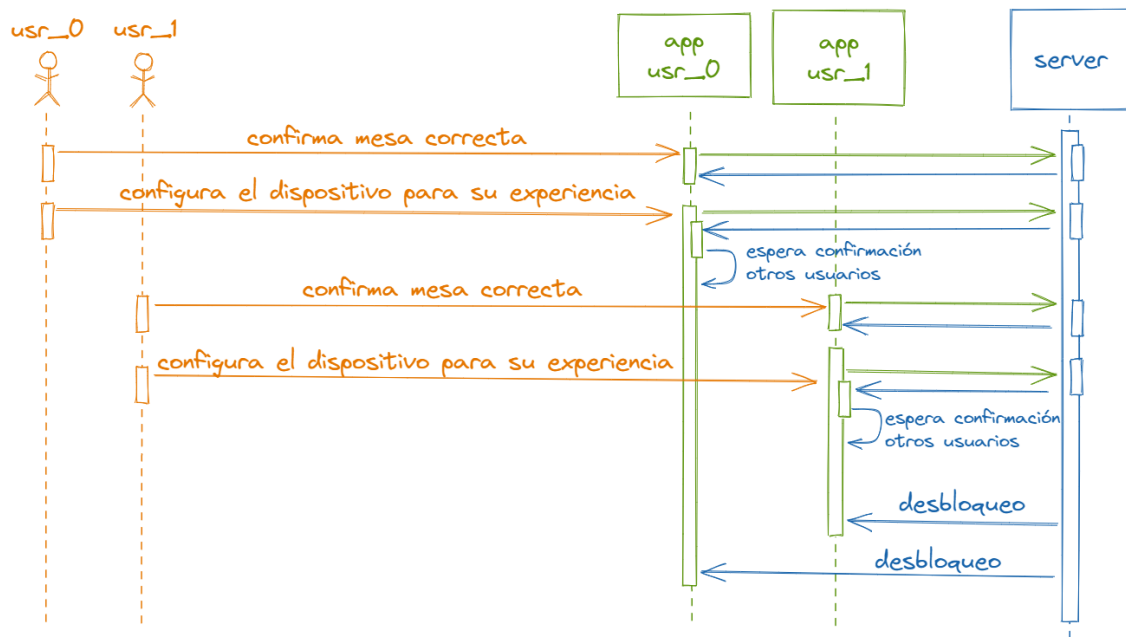


Figura 31. Diagrama secuencia bloqueo espera de confirmación de todos los usuarios en la mesa

Por un mayor entendimiento, en el ejemplo los dos actores son *usr_0* y *usr_1*, que hacen el papel de comensales, *app_usr_0* es la instancia del Front End ejecutándose en el dispositivo que maneja *usr_0* y *app_usr_1* es la instancia del Front End ejecutándose en el dispositivo que maneja *usr_1*, y *server* es la única instancia del Back End en ejecución.

Ambos actores confirman que estan en la mesa correcta, y solo uno de ellos configura su dispositivo para su experiencia personal, como el otro actor no ha llegado a confirmar todavía, se queda bloqueado a que todos los actores involucrados lo hagan. Cuando el segundo actor se configura para su experiencia personal, el servidor lo detecta y desbloquea a todos los actores involucrados, pudiendo así proseguir con la experiencia Alafia.

Sincronización para lanzar contenido multimedia al recibir el plato configurado

En este caso, el punto de sincronización se produce cuando un plato está listo para servirse.

Entran en juego de nuevo dos actores, el jefe de sala y un comensal. Como ya vimos, el jefe de sala tiene el cometido de organizar a los clientes cuando lleguen al restaurante, organizar a los gestores de experiencia, y tiene el control sobre el estado de todos los platos que se sirven.

Cuando un plato va a ser servido, una acción manual es requerida por el jefe de sala, para cambiarle el estado de *En espera* a *Servido*. En ese momento en el dispositivo del usuario al que le será servido el plato que ha sufrido el cambio de estado, navega hasta una pantalla en la cual se le muestra al usuario un video introductorio del plato en cuestión, historia, ingredientes, etc.

Al igual que con el anterior punto de sincronización, con un diagrama de secuencia visualizaremos mejor todo esto, **Figura 32**.

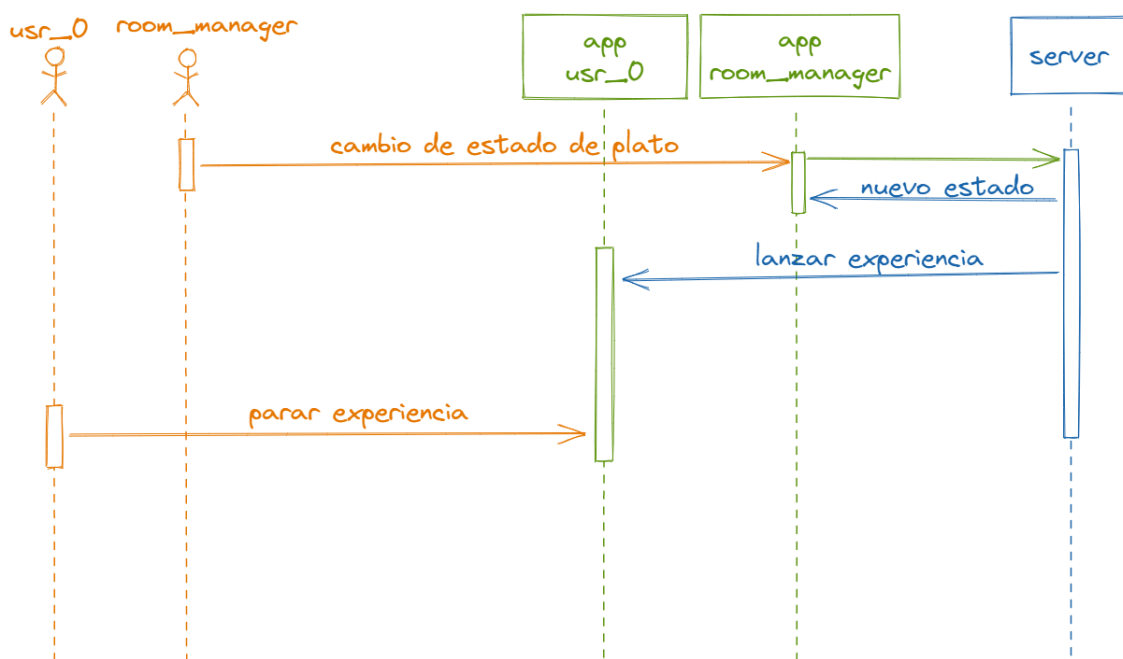


Figura 32. Diagrama secuencia lanzamiento de experiencia por jefe de sala

Sincronización para mostrar notificación cuando se solicita un gestor de experiencia

Este punto de sincronización se produce cuando un cliente que se encuentra en su menú principal de la aplicación, solicita la asistencia de un gestor de experiencia para ya sabemos que, mantener una conversación por alguna curiosidad que tuviera el cliente sobre su cultura o sobre su estancia en el restaurante. Para que esto sea posible, vamos a seguir con la línea de

trabajo definida en Alafia para que toda gestión se produzca a través del dispositivo personal de cada cliente.

Para ello, el cliente tiene en su menu principal la opción “*Gestor de experiencia*”, mediante la cual, solicitará la asistencia de algún gestor de experiencia para que le atienda en la mayor brevedad posible.

Esta petición no le llega directamente al gestor de experiencia, pues en esta primera iteración de la aplicación no cuentan con un dispositivo personal, si no que la petición le llega al jefe de sala, que es quien le indicará que mesa ha solicitado la atención de un gestor de experiencia.

Por tanto, la sincronización se produce cuando el cliente solicita la atención de un gestor de experiencia, pues en el dispositivo del jefe de sala se le debe indicar que en la mesa de ese cliente se ha producido una petición de un gestor de experiencia. Podemos observar con mayor claridad esta sincronización con el diagrama de la **Figura 33**.

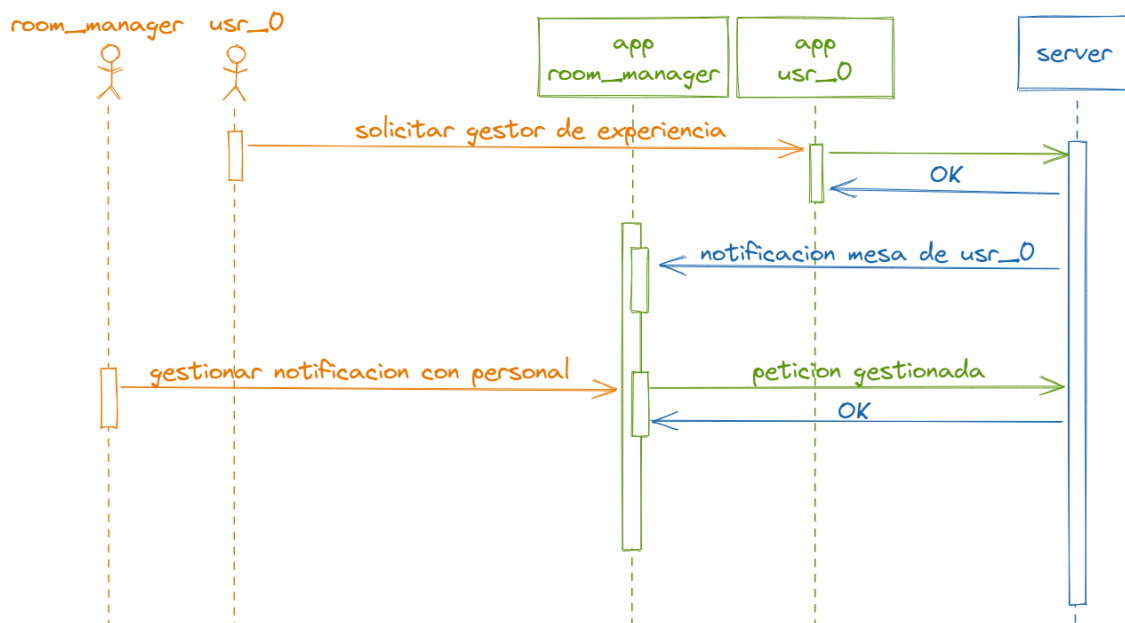


Figura 33. Solicitud de gestor de experiencia

Impacto social

Uno de los objetivos principales del proyecto Alafia, obviando el hecho del reto tecnológico para la creación de todo el sistema informático que se ha desarrollado en este documento, y de ofrecerle a la sociedad una experiencia gastronómica que permite “viajar” momentáneamente a un país con cultura africana, es medir el impacto social que tiene este tipo de experiencias en nuestra sociedad.

Para ello, a todos los clientes se les insta a que realicen un pequeño test al final de su experiencia en el restaurante para obtener datos que puedan ser utilizados posteriormente para la extracción de estadísticas que sirvan para la medición de cuanto y como les ha impactado en su mentalidad a los clientes el haber vivido la experiencia Alafia.

Se les plantean preguntas tipo si les ha gustado el ambiente, la música, decoración; si han sentido realmente que han viajado momentáneamente a un país con cultura africana; si han estado totalmente cómodos durante toda la experiencia; si les ha gustado la comida; si la interacción con los gestores de experiencia ha sido satisfactoria; si tras la experiencia ha habido algún tipo de evolución de mentalidad frente a la interacción con sociedades de diferentes culturas, etc.

Destacar también que se ha creado un modelo de negocio novedoso, ya que Alafia incorpora nuevos elementos para un servicio de restauración que no se ve en casi ningún negocio actual, pues toda la gestión se realiza a través de un dispositivo personal para cada cliente, de forma que los errores de comunicación se reducen al mínimo.

Conclusiones

Para concluir con este documento, quería dejar una reflexión de lo que ha supuesto su desarrollo y realización.

La clave ha sido la investigación y puesta en marcha de una aplicación web desarrollada con las últimas tecnologías que más se usan actualmente a nivel profesional en el mercado. Para ello ha sido necesario el estudio y desarrollo de múltiples tecnologías y frameworks como la de Angular para el desarrollo del Front End o como la de Spring Boot para el desarrollo del Back End.

Ha sido de suma importancia también la metodología de trabajo que se ha llevado a cabo, pasando por diferentes etapas de desarrollo del software como la definición de funcionalidades que se querían tener para una primera versión MVP, y de otras que quedan fuera de este scope pero que aportarían mejoras significativas para versiones futuras. Siguiendo con un desarrollo estructurado del código. El estilo del código es algo en lo que pongo mucho hincapié, pues creo que favorece una lectura más clara, y se suele entender mucho mejor, al igual que en el desarrollo de test, pues son de vital importancia para tener la seguridad de que el código escrito hace lo que se pretende que haga.

En el grado a veces se tiene la sensación de que no se está trabajando con las tecnologías que realmente se requieren para el mundo laboral actual, y la realización de proyectos como este, hacen que te pongas al día y que salgas un poco más preparado al mundo laboral.

Este proyecto se puede considerar un MVP (Minimum Viable Product) de una aplicación gestora de un restaurante, pues tiene margen de mejora. Funcionalidades que se han quedado fuera de esta primera versión como la incorporación de una pasarela de pagos para la liquidación de la cuenta a través del dispositivo, por poner un ejemplo.

Además de la implantación de la aplicación en un servicio de cloud computing, como por ejemplo *AWS (Amazon Web Services)* o *Google Cloud*, para disponer de acceso a la aplicación desde cualquier lugar con conexión a internet y no tener que montar un servidor para el restaurante. Esta migración a la nube también permitiría atender a N restaurantes, pues todas las instancias compartirían base de datos al atacar a la misma instancia del servidor desplegado en la nube, por lo que una expansión del restaurante sería completamente viable.

Bibliografía

- <https://www.credencys.com/blog/front-end-technologies/>
- <https://cadabra.studio/blog/best-backend-technologies-list-comparison-examples>
- <https://medium.com/javarevisited/10-best-frontend-and-backend-frameworks-for-java-python-ruby-and-javascript-developers-cce3c951787a>
- <https://www.hostinger.es/tutoriales/que-es-mysql>
- <https://www.genbeta.com/desarrollo/mongodb-que-es-como-functiona-y-cuando-podemos-usarlo-o-no>
- <https://www.mongodb.com/es>
- <https://aws.amazon.com/es/redis/>
- <https://www.oracle.com/es/mysql/>
- <https://angular.io/docs>
- <https://www.chakray.com/es/gradle-vs-maven-definiciones-diferencias/>
- <https://spring.io/web-applications>
- <https://spring.io/projects/spring-boot>
- <https://ant.apache.org/>
- https://es.wikipedia.org/wiki/Lenguaje_espec%C3%ADfico_de_dominio
- <https://mvitinnovaciontecnologica.wordpress.com/2020/02/06/guia-de-anotaciones-de-spring-framework/>
- <https://developer.mozilla.org/es/docs/Web/HTTP/Methods>
- <https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference>

Diagramas de la aplicación:

- <https://excalidraw.com/#json=NIAHXAX0mHU53HrDzY1cR,6HUKB9OmXzY2ad4WzltrQ>

