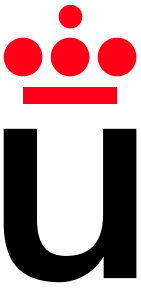


Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

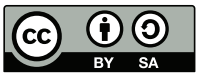
Grado en Ingeniería Informática

Curso 2024-2025

Trabajo Fin de Grado

DESARROLLO Y DESPLIEGUE DE UNA APLICACIÓN WEB PARA LA GESTIÓN DE CITAS EN UNA CLÍNICA DE NUTRICIÓN

Autor: Jorge Leal Gozalo
Tutor: Michel Maes Bermejo



©2025 Jorge Leal Gozalo
Algunos derechos reservados
Este documento se distribuye bajo la licencia
"Atribución-CompartirIgual 4.0 Internacional" de Creative Commons,
disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Agradecimientos

Quisiera expresar mi más sincero agradecimiento a todas las personas que han hecho posible la realización de este trabajo de fin de grado y me han brindado su apoyo a lo largo de mi formación académica.

En primer lugar, quiero agradecer a mi familia por todo el cariño y apoyo que me han dado siempre. Gracias por la educación que he recibido desde pequeño y por enseñarme el valor del esfuerzo y la perseverancia que me han llevado hasta donde estoy ahora.

Agradecer también a mis compañeros de clase, con quienes he compartido los momentos más exigentes de este proceso universitario. Su apoyo y compañía han sido fundamentales para superar esta etapa.

Por último, me gustaría dar las gracias al profesor Michel Maes, tutor de este trabajo de fin de grado, por su implicación, rigor académico y valiosas orientaciones a lo largo de todo el proceso. Su compromiso y disponibilidad han sido fundamentales para el desarrollo de este proyecto.

Resumen

La aplicación de gestión de citas para una clínica de nutrición surge de la necesidad de modernizar un sistema tradicional de citación basado en llamadas y mensajes, especialmente en un contexto donde la obesidad en España muestra una tendencia creciente. Este proyecto propone una solución web que reduce la carga administrativa del personal de la clínica y otorga autonomía a los pacientes.

El frontend es una *single-page application* (SPA) desarrollada en React, desplegada en un bucket de Amazon Simple Storage Service (S3) y distribuida mediante Amazon CloudFront. El backend se implementa con Spring Boot y se ejecuta en una instancia EC2 (Elastic Compute Cloud) gestionada por Elastic Beanstalk de Amazon Web Services (AWS). La base de datos relacional MySQL se aloja en Amazon Relational Database Service (RDS). La autenticación y autorización se gestionan a través de AWS Cognito, emitiendo tokens que validan cada petición al backend. Se utilizan funciones AWS Lambda que automatizan el alta de pacientes en la base de datos y Amazon Simple Email Service (SES) para el envío de correos electrónicos automáticos con la información de las citas.

La interfaz contempla cuatro roles diferenciados: pacientes, nutricionistas, auxiliares y administradores. Los pacientes pueden registrarse de forma autónoma a través de la aplicación o mediante un alta asistida por un miembro de la clínica, gestionar sus propias citas (reserva, consulta y cancelación) y recibir notificaciones por correos automatizados. Los auxiliares administran las agendas de todos los nutricionistas, mientras que estos últimos pueden acceder a su calendario y reprogramar citas si lo desean.

Para asegurar la calidad del software, el proyecto incorpora análisis estático y pruebas automatizadas. También se han adoptado principios DevOps mediante el uso de GitHub Actions para automatizar el proceso de integración y despliegue continuo.

Este documento detalla todas las etapas del proyecto: desde la introducción y la motivación inicial, pasando por la definición de objetivos y requisitos, el análisis de tecnologías y metodologías utilizadas, el diseño e implementación de la aplicación web, las pruebas y el análisis estático, hasta el despliegue en AWS, terminando con las conclusiones y propuestas de trabajos futuros.

Palabras clave:

- Gestión de citas
- AWS (Amazon Web Services)
- Java
- Spring Boot
- React
- Cognito
- RDS (Relational Database Service)
- MySQL
- Elastic Beanstalk
- DevOps

Índice de contenidos

Índice de figuras

Índice de códigos

1 Introducción y motivación

En este apartado se explica cuál es el contexto y alcance de este proyecto, así como el origen de la motivación para llevarlo a cabo.

1.1. Contexto y alcance

La idea de este proyecto surge de una experiencia personal al haber colaborado en la clínica de nutrición de un familiar. La realidad es que muchos centros de nutrición emplean sistemas de gestión de citas obsoletos, en los que el procedimiento para pedir una cita se basa habitualmente en llamadas telefónicas o mensajes. El proceso de citación suele desarrollarse del siguiente modo: el paciente llama al centro de nutrición cuando desea pedir una cita con un nutricionista y este es atendido por un auxiliar. El auxiliar es el encargado de revisar si la fecha y hora solicitadas por el paciente están disponibles en la agenda del nutricionista. Si hay disponibilidad en la franja horaria solicitada, el auxiliar registra la cita en la agenda. En caso contrario, tendrá que llegar a un acuerdo con el paciente para buscar otra fecha y hora que estén libres y que se ajusten a la disponibilidad del paciente.

El sistema es el mismo cuando un paciente desea cambiar la fecha y hora de una cita, cancelar la consulta porque no puede asistir o porque no recuerda la fecha y hora de la cita y tiene que contactar con el centro para consultarla.

En la última década, el número de casos diagnosticados de obesidad en España ha aumentado notablemente. Han pasado de haber 1.877.140 casos en 2013 a más del doble en 2023, con un total de 3.792.362 casos. Este incremento supo-

ne un grave problema sanitario por la cantidad de enfermedades asociadas a la obesidad, como la diabetes tipo 2, las enfermedades cardiovasculares, la hipertensión, las patologías respiratorias y determinados tipos de cáncer. A medida que aumenta el número de personas con obesidad, también lo hace la incidencia de estas enfermedades asociadas. En la figura 1.1 podemos observar la evolución del número de casos de obesidad en España.

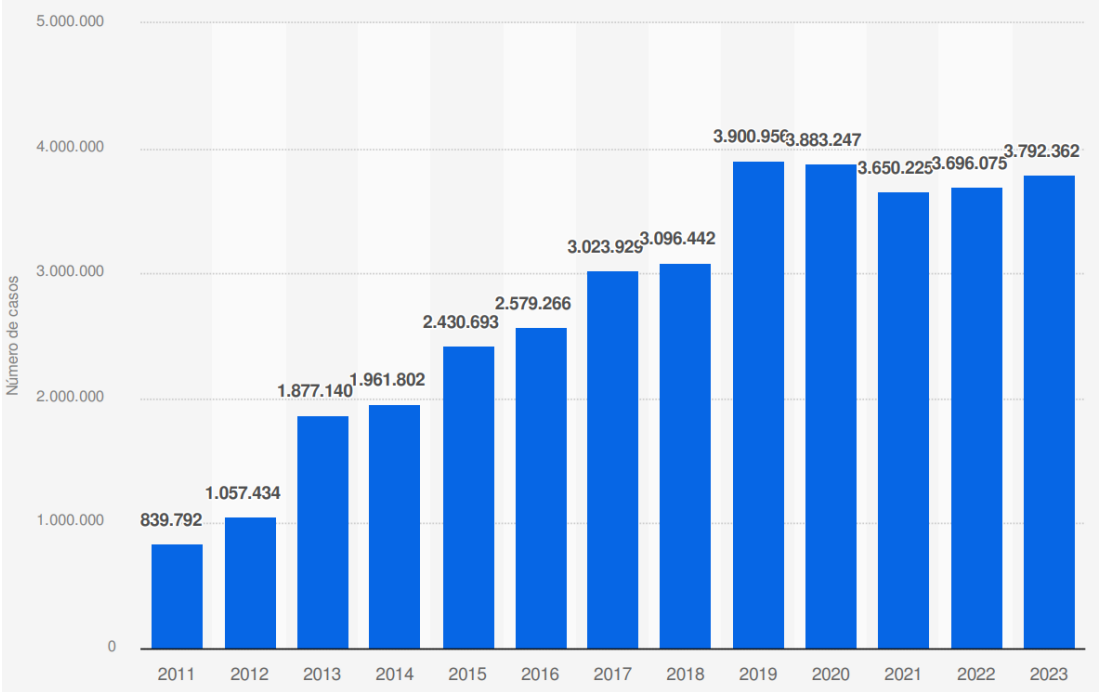


Figura 1.1: Número de casos de obesidad registrados en España de 2011 a 2023. [1]

En este contexto, el número de pacientes que acuden a las clínicas de nutrición está aumentando considerablemente año tras año, desbordando el trabajo de los auxiliares, quienes deben atender cada vez más llamadas y mensajes. Esta situación provoca que no todos los pacientes puedan ser atendidos, ya que se les imposibilita pedir, cancelar o modificar sus citas cuando lo deseen, lo que a su vez impide que otros pacientes aprovechen esos huecos o, peor aún, puede llevar al abandono de la dieta y la clínica debido a la rigidez y saturación del sistema de citación.

Para hacer frente a todos estos problemas, se ha desarrollado una aplicación web orientada a la gestión de citas en una clínica de nutrición. El objetivo principal consiste en prescindir de tener que llamar o escribir al centro de nutrición cada vez que se desee reservar, cancelar o modificar una cita y facilitar así el proceso de citación. De esta forma, se reduce drásticamente la carga de trabajo de los auxiliares y se otorga flexibilidad a los pacientes para que gestionen sus citas ellos mismos.

A través de la aplicación web, los pacientes podrán registrarse en línea, reservar una cita seleccionando la fecha y hora disponibles para un nutricionista concreto, consultar sus citas pendientes y la opción de poder cancelarlas si lo desean. Al reservar una cita, el sistema enviará automáticamente un correo electrónico a la dirección del paciente con la información de esta, de modo que los detalles de la cita sean accesibles para el paciente.

No obstante, se mantiene la posibilidad de que los usuarios que prefieren el método tradicional continúen llamando al centro de nutrición para gestionar sus citas. Por ello, la aplicación web dispone de una interfaz intuitiva y accesible que permite a los auxiliares administrar las agendas de todos los nutricionistas y gestionar las solicitudes de cita que se realicen por vía telefónica o por mensaje. Además, el propio nutricionista puede visualizar su agenda de citas y reprogramar consultas, lo que le permite mantenerse conectado con la plataforma de citación, estar al día de su planificación y realizar modificaciones puntuales en su calendario.

En cuanto a aspectos tecnológicos, la aplicación está completamente desplegada en la nube, sobre una arquitectura que integra numerosos servicios gestionados para garantizar su robustez y eficiencia. Gracias al uso de diversas herramientas *cloud*, la plataforma ofrece alta disponibilidad, de modo que los usuarios pueden acceder al sistema de gestión de citas en cualquier momento sin interrupciones. Además, la elasticidad de los servicios utilizados permite adaptar los recursos a medida que crece el número de usuarios, lo que facilita que la aplicación mantenga un buen rendimiento ante picos de demanda y optimice los costes cuando la carga disminuye.

En definitiva, esta aplicación web responde tanto a las necesidades de los pacientes como a las exigencias operativas del personal de la clínica, con el objetivo de permitir una gestión autónoma y eficiente de citas en un entorno sanitario especializado en nutrición.

2

Objetivos

2.1. Objetivos generales

El objetivo principal del proyecto es diseñar, implementar y desplegar una aplicación web de gestión de citas para una clínica de nutrición que permita prescindir del proceso tradicional basado en llamadas o mensajes, reduciendo drásticamente la carga de trabajo de los auxiliares de la clínica y otorgando flexibilidad a los pacientes para que gestionen sus citas de forma autónoma.

Adicionalmente, el proyecto plantea aprovechar las ventajas de la computación en la nube, de manera que la aplicación resultante sea altamente disponible, escalable y fiable.

2.2. Objetivos específicos

2.2.1. Implementación de funcionalidades clave

- **Sistema de reserva de citas**: Desarrollar la funcionalidad que permita a los pacientes reservar citas con su nutricionista de preferencia a través de la plataforma web, visualizando en tiempo real las fechas y horas disponibles en la agenda de cada especialista.
- **Registro y gestión de cuentas de usuario**: Implementar un mecanismo en el que los pacientes puedan crear su propia cuenta sin asistencia de un

miembro de la entidad. A su vez, el sistema debe permitir dar de alta a nuevos miembros de la clínica, así como activar o desactivar cuentas según sea necesario.

- **Consulta y cancelación de citas por parte del paciente**: Proporcionar a los pacientes el poder consultar sus citas pendientes y la información de estas. Añadir la opción de cancelar citas, para que el paciente pueda gestionar las cancelaciones de manera autónoma.
- **Administración de agendas**: Desarrollar un componente para que los auxiliares consulten y modifiquen las agendas de los nutricionistas de forma ágil (para atender solicitudes vía llamada o mensaje) y para que los nutricionistas puedan visualizar su agenda de citas y reprogramar consultas si lo desean.
- **Notificaciones automáticas por correo**: Automatizar el envío de correos mediante AWS SES que incluyan la información de la cita, de modo que sirva de recordatorio a los pacientes.
- **Interfaz**: Diseñar una interfaz *responsive* que se adapte a cualquier tamaño de pantalla en el caso de los pacientes, para que puedan usar la aplicación en cualquier dispositivo. En cambio, para los miembros de la clínica, la aplicación debe estar diseñada para entornos de escritorio al utilizarse en un espacio de trabajo.

2.2.2. Garantizar una infraestructura sólida

- **Distribución global del frontend**: Desplegar el frontend estático en un bucket S3 y distribuirlo con Amazon CloudFront para minimizar latencias.
- **Despliegue del backend**: Desplegar el backend en AWS Elastic Beanstalk sobre sobre una instancia EC2.
- **Base de datos**: Configurar Amazon RDS con motor MySQL en una subred privada, definiendo grupos de seguridad que restrinjan el acceso únicamente a los servicios autorizados.

2.2.3. Seguridad y control de acceso

- **Autenticación robusta de usuarios**: Integrar Amazon Cognito como proveedor de identidad, para que almacene las credenciales y gestione la recuperación de contraseña, registro e inicio de sesión de los usuarios.
- **Autorización basada en roles y privilegios mínimos**: Establecer un control de acceso basado en distintos roles, que delimite las acciones que

puede realizar cada tipo de usuario en la aplicación. Tras la autenticación, cada petición al backend incluirá un token JWT emitido por Cognito, que será validado y se asociará al usuario a un rol determinado.

- **Comunicación segura con certificados digitales**: Garantizar que toda interacción con la aplicación se realice bajo HTTPS/TLS.

2.2.4. Pruebas y asegurar la calidad del software

- **Pruebas unitarias e integración**: Implementar pruebas unitarias que garanticen que la lógica de negocio sea correcta. Realizar pruebas de integración donde se verifica el comportamiento esperado de la aplicación en su conjunto, haciendo uso de un perfil de test para no utilizar los recursos de desarrollo o producción.
- **Análisis estático de código**: Incorporar herramientas de análisis estático de código que detecten posibles errores, *bugs*, malas prácticas o desviaciones de estilo.

2.2.5. Automatización del ciclo DevOps (CI/CD)

Implementar un flujo de integración continua y despliegue continuo (CI/CD) utilizando GitHub Actions para automatizar las pruebas y el despliegue tanto en el backend como en el frontend.

3

Tecnologías, Herramientas y Metodologías

3.1. Lenguajes de programación

En esta sección se abordarán los lenguajes de programación empleados y en qué parte del proyecto se han utilizado.

3.1.1. Java

El lenguaje de programación utilizado en el *backend* ha sido Java (Figura 3.1) [2]. Es un lenguaje de programación multiplataforma orientado a objetos, conocido por su seguridad, portabilidad y robustez. Tiene una gran comunidad y ofrece una amplia cantidad de librerías y *frameworks*.

Figura 3.1: Logotipo de Java

3.1.2. JavaScript

El lenguaje de programación sobre el que se ha desarrollado principalmente el *frontend* ha sido JavaScript (Figura 3.2) [3]. Es un lenguaje de programación interpretado, orientado a objetos y basado en prototipos, diseñado inicialmente para añadir interactividad a las páginas web en el navegador. JavaScript garantiza interoperabilidad entre diferentes entornos y facilita el uso de múltiples librerías y frameworks (React, Vue, Angular) para la construcción de interfaces de usuario dinámicas y escalables. En el proyecto se ha integrado con React, framework del que hablaremos más adelante.

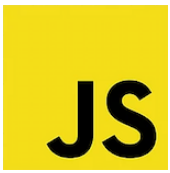


Figura 3.2: Logotipo de JavaScript

3.1.3. HTML

HTML5 (Figura 3.3) [4] es el lenguaje de marcado estándar para la creación y estructuración de contenido en páginas web. HTML define la semántica de una página mediante elementos que describen la función de cada sección, facilitando tanto la accesibilidad como la indexación por navegadores y motores de búsqueda. En el proyecto se ha utilizado para definir la estructura semántica de las vistas en los componentes React.

Figura 3.3: Logotipo de HTML

3.1.4. CSS

CSS (Figura 3.4) [5] es un lenguaje de hojas de estilo que permite diseñar y dar presentación a documentos HTML, controlando aspectos como tipografía, color, espaciado y distribución de elementos en la página. Ofrece módulos que permiten diseños *responsive* adaptables a diferentes tamaños de pantalla, así como la capacidad de animaciones y efectos.

Figura 3.4: Logotipo de CSS

3.1.5. YAML

YAML (YAML *Ain't Markup Language*) [6] es un formato de serialización de datos legible, diseñado para representar estructuras complejas mediante indentación y un esquema sencillo. Facilita la definición de configuraciones y flujos de trabajo sin la verbosidad de otros formatos, ofreciendo claridad gracias a su sintaxis basada en sangrías. En el proyecto se ha utilizado para definir dos workflows de GitHub Actions (`ci.yml` para la integración continua y `deploy.yml` para el despliegue continuo) y también para la configuración de propiedades de Spring Boot.

3.2. Tecnologías

3.2.1. Frontend

Para desarrollar el frontend se ha utilizado React (Figura 3.5) [7], un framework de JavaScript utilizado para construir interfaces de usuario interactivas y reutilizables. Basado en un modelo declarativo, permite definir componentes que reflejan el estado de la aplicación, de modo que al actualizarse dicho estado, la interfaz se renderiza automáticamente en pantalla sin necesidad de manipular el DOM[1] (*Document Object Model*) de forma directa. El uso del virtual DOM es una de sus principales ventajas, ya que mantiene una representación en memoria de la interfaz y compara los cambios con la versión anterior del virtual DOM, para modificar solo esos nodos en el DOM, mejorando el rendimiento y reduciendo el coste de las operaciones de renderizado. Utiliza una arquitectura basada en componentes que facilita la modularidad y el mantenimiento del código.

La aplicación web es una SPA (*Single Page Application*) [8], debido a que

[1]representación jerárquica de un documento HTML, al que se puede acceder y modificar mediante JavaScript.

ejecuta todo su contenido en una sola página. Al abrir la web descarga el contenido HTML, CSS y JavaScript por completo y solo necesita cargar el contenido nuevo de forma dinámica si este lo requiere, evitando así recargar la página por completo. Esto mejora los tiempos de respuesta y agiliza la navegación, favoreciendo la experiencia del usuario.

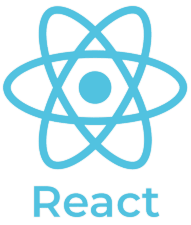


Figura 3.5: Logotipo de React

Figura 3.6: Logotipo de Bootstrap

Las principales librerías y dependencias del frontend han sido:

- **react-router-dom** [9]: Librería que permite la navegación entre diferentes componentes o páginas de una aplicación sin recargar la página completa.

- **react-oidc-context** [10]: Librería que gestiona la autenticación OpenID Connect (OIDC) en aplicaciones React SPA, la cual será explicada en profundidad en el apartado de autenticación y autorización. Al envolver la raíz de la aplicación con `AuthProvider`, crea un contexto[2] que permite almacenar toda la información relevante de la sesión actual como los datos del usuario, estado de autenticación y métodos para iniciar o cerrar sesión. Se encarga de iniciar sesión automáticamente y restablecer la sesión anterior.

- **bootstrap-react / bootstrap** (Figura 3.6) [11]: En el proyecto se ha utilizado el framework de Bootstrap para aprovechar sus estilos y utilidades CSS, así como la librería react-bootstrap [12], que utiliza los componentes de Bootsrap como componentes de React. Esta integración permite utilizar los estilos de Bootstrap junto con la sintaxis y lógica de React.

- **react-big-calendar** [13]: Librería que te permite utilizar un componente calendario con una gran variedad de funciones, como la gestión de eventos, navegación por fechas y personalización de las diferentes vistas (mes, semana, día, etc).

- **react-toastify** [13]: Librería que proporciona notificaciones personalizables a las aplicaciones de React. Permite una fácil integración, diseño personalizable, notificaciones no intrusivas y opciones de cierre flexibles.

[2] mecanismo que permite compartir datos o estado de forma global entre componentes, sin tener que pasarlos manualmente a través de props en cada nivel.

3.2.2. Backend

En cuanto a las tecnologías empleadas para el desarrollo del backend, se ha optado por usar Spring Boot (Figura 3.7) [14], una herramienta que ayuda a crear aplicaciones independientes basadas en Spring. Su facil configuración, simplicidad y reducción de código hacen que puedas empezar con con la mínima dificultad. Se ha utilizado la versión 3.4.6 en el proyecto.

Spring [15] es un framework de código abierto para el desarrollo de aplicaciones empresariales en Java. Permite a los desarrolladores centrarse en la implementación de la lógica de negocio, sin preocuparse por la forma en que se gestionan internamente aspectos como la inyección de dependencias, acceso a base de datos, la propagación de transacciones o el enrutamiento de mensajes.

Figura 3.7: Logotipo de Spring Boot

A continuación, se describen los principales módulos de Spring Boot utilizados en este proyecto:

- **Spring MVC** [16]: Sigue el patrón de diseño arquitectónico Modelo-Vista-Controlador, que funciona en torno al servlet Dispacher. Este servlet gestiona y envía todas las solicitudes HTTP entrantes al controlador correspondiente. La anotación `@Controller` define que una clase específica es un controlador y la anotación `@RequestMapping` asigna las solicitudes web a los métodos del controlador Spring.

- **Spring Data JPA** [17]: Facilita la implementación de repositorios basados en JPA (Java Persistence API). Ofrece interfaces como `JpaRepository<T, ID>` que permiten implementar operaciones CRUD y consultas comunes sin escribir código SQL manualmente. Utiliza Hibernate como proveedor JPA para gestionar el mapeo de entidades (`@Entity`, `@OneToMany`, `@ManyToOne`, etc.), la caché y las transacciones automáticamente, agilizando el desarrollo de aplicaciones basadas en Spring.

Para la seguridad del backend se ha utilizado Spring Security [18],un *framework* que intercepta las peticiones HTTP y aplica una cadena de filtros configurable antes de llegar a los controladores. Además de ofrecer soporte integral para autenticación y autorización, permite integrar fácilmente mecanismos basados en OAuth 2.0 o JWT (JSON Web Token), gestionar roles y permisos en los métodos

y configurar políticas de CORS (Cross-Origin Resource Sharing)[3].

Para completar la configuración de seguridad, se ha añadido además la dependencia de Spring Security OAuth2 Resource Server [19], una extensión de Spring Security que permite la protección de endpoints de una aplicación mediante el uso de Bearer Tokens que pueden ser emitidos por distintos proveedores de autorización. En el proyecto se han utilizado JWT como tokens y Amazon Cognito como proveedor.

Se han integrado en el backend los siguientes servicios de AWS:

- **Amazon Cognito** (Figura 3.8) [20]: Es un servicio de AWS que permite añadir rápidamente el registro, inicio de sesión y administración de usuarios a aplicaciones web y móviles. Proporciona un *User Pool* (Almacén de identidades) con soporte para cuentas locales y federadas (Google, Facebook, SAML, etc.), emite tokens JWT y permite gestionar grupos de usuarios, reglas de contraseñas y mecanismos de MFA.

- **Amazon Simple Email Service** (Figura 3.9) [21]: Es un proveedor de servicios de correo electrónico basado en la nube que se puede integrar en cualquier aplicación para la automatización de grandes volúmenes de correos electrónicos. Se puede integrar con otras herramientas de AWS y utiliza un modelo de pago bajo demanda donde solo se paga por lo que se utiliza.

Figura 3.8: Logotipo de AWS Cognito

Figura 3.9: Logotipo de AWS SES (Simple Email Service)

Finalmente, para evitar código repetitivo, se ha incorporado Lombok [22], una librería de Java que puede generar automáticamente los métodos `getters`, `setters`, `equals`, `hashCode` y `toString`. Además, se ha utilizado Java-Dotenv [23] para gestionar las variables de entorno en desarrollo. Esta biblioteca lee un archivo `.env` y exporta sus valores a la aplicación, de modo que las credenciales no queden hardcodeadas y puedan variar según el entorno (local o producción).

[3]mecanismo de seguridad del navegador que permite a una página web acceder a recursos de un dominio diferente al suyo.

3.2.3. Bases de datos

Para el entorno de producción se ha elegido Amazon RDS (Figura 3.10) [24] ejecutando MySQL 8.0.40. Es un servicio web que facilita la configuración, el funcionamiento y el escalado de bases de datos relacionales en la nube de AWS. Ofrece capacidad redimensionable y rentable para bases de datos relacionales estándar del sector y gestiona tareas comunes de administración de bases de datos.

Durante el desarrollo local se ha utilizado MySQL (Figura 3.11), lo que permite trabajar con un entorno idéntico al de producción sin depender de la infraestructura de AWS.

Para las pruebas unitarias y de integración se emplea H2 Database (Figura 3.12) [25], una base de datos muy ligera basada en Java. La principal ventaja de esta base de datos es que, al ejecutarse en memoria y ser muy ligera, permite realizar consultas y pruebas unitarias que requieren acceso a datos de manera sencilla y rápida, sin necesidad de instalar un sistema de gestión de bases de datos completo.

Figura 3.10: Logotipo de AWS RDS

Figura 3.11: Logotipo de MySQL

Figura 3.12: Logotipo de H2 Database

3.2.4. Pruebas

Análisis estático frontend

- **ESLint**(Figura 3.13) [26]: Es un proyecto de código abierto que ayuda a identificar y resolver problemas en el código JavaScript. Analiza estáticamente el código para detectar rápidamente errores, está integrado en la mayoría de los editores de texto y se puede ejecutar como parte del flujo de trabajo de integración continua. Muchos de los errores pueden corregirse de forma automática con ESLint y es posible personalizar escribiendo reglas que se ajusten a las necesidades específicas del proyecto.

- **Stylelint** (Figura 3.14) [27]: Es una herramienta de análisis estático (linter) para hojas de estilo CSS que ayuda a evitar errores y a aplicar convenciones de codificación. Ofrece un conjunto de reglas integradas para verificar la sintaxis y las mejores prácticas de CSS. Permite personalizar las reglas

según las necesidades del proyecto. Además, puede corregir automáticamente ciertos problemas y se integra fácilmente en flujos de integración continua, mejorando la calidad y consistencia del código CSS.

- **Prettier** (Figura 3.15) [28]: Formateador de código automatizado que garantiza un estilo de codificación coherente en proyectos de desarrollo. Es compatible con una amplia variedad de lenguajes. Reescribe el código fuente desde cero, aplicando reglas de formato predefinidas que consideran aspectos como la longitud de las líneas, la colocación de comillas y la indentación. Esto asegura que todo el código en un proyecto siga un estilo uniforme.

Figura 3.13: Logotipo de ESLint

Figura 3.14: Logotipo de Stylelint

Figura 3.15: Logotipo de Prettier

Análisis estático backend

- **Checkstyle** (Figura 3.16) [29]: Es una herramienta de análisis estático de código utilizada en el desarrollo de software para verificar que el código fuente en Java cumpla con las normas de codificación establecidas. Automatiza el proceso de revisión del código, asegurando la coherencia y calidad del código a lo largo del proyecto. Es altamente configurable y puede adaptarse a casi cualquier estándar de codificación.

- **PMD** (Figura 3.17) [30]: Es un analizador de código estático multilenguaje. Detecta errores comunes de programación, como variables no utilizadas, bloques `catch` vacíos, creación innecesaria de objetos, etc. Se centra principalmente en Java y Apex, pero es compatible con otros 16 lenguajes. Incluye más de 400 reglas integradas y se puede ampliar con reglas personalizadas. Además, puede detectar código duplicado dentro del proyecto.

- **SpotBugs** (Figura 3.18) [31]: Es una herramienta de análisis estático de código para proyectos Java que detecta errores comunes y posibles fallos de programación en el código fuente. Analiza el *bytecode* de los programas Java para identificar patrones problemáticos como posibles excepciones, problemas de concurrencia o errores de estilo, que podrían afectar la estabilidad y el rendimiento de la aplicación.

Figura 3.16: Logotipo de Checkstyle

Figura 3.17: Logotipo de PMD

Figura 3.18: Logotipo de SpotBugs

Pruebas unitarias

Para las pruebas unitarias se ha utilizado JUnit 5 (Figura 3.19) [32], el principal *framework* de pruebas unitarias en Java. Permite estructurar y ejecutar pruebas de manera sencilla y eficiente mediante anotaciones. Permite la configuración y limpieza de recursos entre pruebas mediante anotaciones como `@BeforeEach` y `@AfterEach`. Además, ofrece una serie de aserciones como `assertEquals` para comprobar que los resultados sean los esperados o `assertThrows` para verificar que se lanzan excepciones en condiciones específicas.

También se ha utilizado Mockito (Figura 3.20) [33] en las pruebas unitarias, una librería utilizada para crear stubs[4], mocks[5] y spies[6]. Su utilidad radica en aislar el comportamiento de una clase o método específico sin depender de sus interacciones con otras partes del sistema, como bases de datos o servicios externos, como podría ser AWS en el caso de este proyecto. Permite comprobar que ciertos métodos han sido invocados, la cantidad de veces que se han ejecutado o el orden en el que se han invocado.

Figura 3.19: Logotipo de JUnit 5

Figura 3.20: Logotipo de Mockito

Pruebas de integración

Para las pruebas de integración en el backend, se ha utilizado Spring Boot Test [34], un módulo de Spring Boot que permite arrancar el contexto completo de la aplicación en un entorno de pruebas. Al utilizar la anotación `@SpringBootTest`,

[4] simulaciones totales que requieren que se defina el comportamiento de sus métodos.
[5] iguales a los stubs, pero se verifica que los métodos especificados han sido llamados.
[6] pueden utilizar métodos reales o simulados.

Spring arranca todos los componentes de la aplicación, incluyendo servicios, repositorios y controladores, lo que permite realizar pruebas que cubren múltiples capas de la arquitectura. Se ha utilizado una base de datos H2, la cual Spring Boot configura automáticamente la conexión y la inicialización de los datos de prueba. Este enfoque asegura que las pruebas de integración validen el comportamiento real del sistema en su totalidad, utilizando componentes reales en lugar de simulaciones.

MockMvc [35] es un framework dentro de Spring Test que ofrece soporte para probar aplicaciones Spring MVC. Se utiliza para realizar pruebas de los endpoints REST sin necesidad de arrancar un servidor web real. Permite simular peticiones HTTP (`GET, POST, PUT, DELETE`), pudiendo añadir cabeceras de autorización, roles o cualquier otro dato necesario. Posteriormente, verifica las respuestas para comprobar que el código de estado y el contenido de la respuesta son los esperados.

Cobertura de código

Para el análisis de cobertura de código se ha utilizado JaCoCo [36], que proporciona métricas de cobertura sobre instrucciones, ramas, líneas, métodos y complejidad ciclomática. Se integra de forma sencilla y es compatible con todas las versiones de archivos Java publicadas.

3.2.5. Despliegue y alojamiento

- **Amazon Elastic Compute Cloud (EC2)** (Figura 3.21) [37]: Es un servicio de AWS que proporciona capacidad de computación escalable bajo demanda en la nube. Permite iniciar y configurar servidores virtuales, conocidos como instancias, sin necesidad de adquirir hardware físico. Cada instancia se ejecuta a partir de una Imagen de Máquina de Amazon (AMI), que incluye el sistema operativo y el software necesario. Los usuarios pueden elegir entre diferentes tipos de instancias, que ofrecen diversas combinaciones de recursos de computación, memoria, almacenamiento y capacidad de red, adaptándose a las necesidades específicas de cada aplicación.

- **AWS Elastic Beanstalk** (Figura 3.22) [38]: Es una plataforma como servicio (PaaS) proporcionada por AWS que facilita la implementación y gestión de aplicaciones en la nube sin necesidad de gestionar la infraestructura subyacente. Permite cargar la aplicación para administrar automáticamente los recursos necesarios, como instancias EC2, balanceadores de carga, escalado automático y supervisión del estado de la aplicación. Proporciona herramientas para monitorear el rendimiento de la aplicación y realizar ajustes según sea necesario. Tiene un modelo de precios bajo demanda donde solo se paga por lo que se consume.

- **Amazon Simple Storage Service (S3)** (Figura 3.23) [39]: Es un servicio de almacenamiento de objetos en la nube proporcionado por AWS. Permite almacenar y recuperar cualquier cantidad de datos desde cualquier lugar, ofreciendo una escalabilidad, disponibilidad y seguridad líderes en la industria. Los datos se organizan en *buckets*[7] y se gestionan como objetos.
- **Amazon CloudFront** (Figura 3.24) [40]: Es una red de entrega de contenido (CDN) proporcionada por AWS que acelera la distribución de contenido web estático y dinámico, como archivos HTML, CSS, JavaScript e imágenes, a los usuarios finales. Utiliza una red global de centros de datos denominados *edge locations* para entregar el contenido con la menor latencia posible. Cuando un usuario requiere contenido, la solicitud se redirige a la *edge location* que ofrece la mínima latencia, asegurando un rendimiento óptimo. Si el contenido no se encuentra en el *edge locations*, CloudFront lo recupera del origen configurado y lo almacena en caché para futuras solicitudes. Además de mejorar el rendimiento, CloudFront refuerza la seguridad mediante el cifrado del tráfico y controles de acceso personalizables.

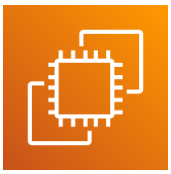


Figura 3.21: Logotipo de Ec2

Figura 3.22: Logotipo de Elastic Beanstalk

Figura 3.23: Logotipo de S3

Figura 3.24: Logotipo de CloudFront

3.2.6. Infraestructura y servicios auxiliares en AWS

- **Amazon Virtual Private Cloud** (Figura 3.25) [41]: Permite lanzar recursos de AWS dentro de una red virtual lógicamente aislada, definida por el usuario. Esta red virtual reproduce la arquitectura de un centro de datos tradicional, pero aprovecha la escalabilidad y fiabilidad de la plataforma AWS. Una vez creada la VPC, es posible añadir subredes, configurar tablas de enrutamiento y conectar *gateways* para controlar de forma detallada el flujo de tráfico hacia y desde los recursos desplegados.
- **AWS Identity and Access Management (IAM)** (Figura 3.26) [42]: Es un servicio web que facilita el control seguro del acceso a los recursos de AWS, gestionando quién está autenticado y autorizado para interactuar con cada servicio o recurso. IAM introduce el concepto de identidades (usuario raíz, usuarios de IAM, roles y entidades federadas) y políticas para definir con detalle los permisos de cada usuario.

[7] contenedor virtual que almacena objetos y sus metadatos.

- **AWS Lambda** (Figura 3.27) [43]: Es un servicio de computación sin servidor que permite ejecutar código sin aprovisionar ni administrar servidores. Lambda ejecuta el código en una infraestructura de alta disponibilidad, gestionando automáticamente el aprovisionamiento de capacidad, el mantenimiento del sistema operativo, el escalado automático y las funciones de registro. Para utilizarlo, se organiza el código en funciones independientes que se cargan en uno de los entornos de ejecución compatibles y se ejecutan únicamente cuando se invocan, escalando según la demanda. Con este modelo, solo se paga por el tiempo de cómputo consumido.
- **Amazon EventBridge** (Figura 3.28) [44]: Es un servicio de bus de eventos sin servidor que proporciona un flujo de datos en tiempo real desde aplicaciones propias, aplicaciones de software como servicio (SaaS) y otros servicios de AWS, enrutando esos datos a destinos como AWS Lambda. Permite crear arquitecturas basadas en eventos, con un acoplamiento flexible y distribuidas.

Figura 3.25: Logotipo de Amazon Virtual Private Cloud

Figura 3.26: Logotipo de IAM

Figura 3.27: Logotipo de AWS Lambda

Figura 3.28: Logotipo de Amazon EventBridge

- **Amazon Route 53** (Figura 3.29) [45]: Es un servicio de DNS en la nube que dirige el tráfico de Internet a recursos como sitios web o aplicaciones, traduciendo nombres de dominio en direcciones IP. Permite registrar dominios, gestionar zonas DNS y aplicar políticas de enrutamiento avanzadas.
- **AWS Certificate Manager (ACM)**(Figura 3.30) [46]: Es un servicio de Amazon Web Services que facilita la provisión, gestión de certificados SSL/TLS para asegurar las comunicaciones en la nube.
- **Amazon CloudWatch** (Figura 3.31) [47]: Es un servicio de monitoreo en tiempo real que permite supervisar los recursos y aplicaciones que se ejecutan en AWS. Proporciona métricas automáticas de diversos servicios de AWS y permite crear métricas personalizadas. Es posible configurar alarmas para recibir notificaciones o ejecutar acciones automáticas.

Figura 3.29: Logotipo de Amazon Route 53

Figura 3.30: Logotipo de AWS Certificate Manager

Figura 3.31: Logotipo de CloudWatch

3.3. Herramientas

3.3.1. Amazon Web Service (AWS)

Amazon Web Services (AWS) (Figura 3.32) [48] es una plataforma de computación en la nube que ofrece más de 200 servicios desde centros de datos globales. Proporciona soluciones escalables y seguras en áreas como cómputo, almacenamiento, bases de datos, análisis e inteligencia artificial. AWS permite a empresas de todos los tamaños reducir costes, aumentar agilidad e innovar rápidamente sin necesidad de gestionar infraestructura física.

Figura 3.32: Logotipo de Amazon Web Services (AWS)

3.3.2. Control de versiones y repositorio

Se ha utilizado el sistema de control de versiones Git (Figura 3.33) [49], que realiza un seguimiento de los cambios en los archivos de un proyecto, permitiendo conservar un historial completo de commits y volver a cualquier versión anterior en cualquier momento. Permite crear copias locales del repositorio, realizar modificaciones de forma aislada y posteriormente, integrar sus aportes en la rama principal sin interferir con el trabajo de otros usuarios.

GitHub (Figura 3.34) [49] es una plataforma que utiliza Git como sistema de control de versiones y añade funcionalidades para almacenar, compartir y colaborar en proyectos de software. Al alojar el código en repositorios remotos, ofrece herramientas para realizar revisiones del mismo, seguimiento de incidencias y la automatización de flujos de trabajo.

Figura 3.33: Logotipo de Git

Figura 3.34: Logotipo de GitHub

3.3.3. Construcción y gestión de dependencias

Para la construcción y gestión de dependencias se ha utilizado Maven (Figura 3.35) [50], una herramienta de gestión de proyectos y compilación para Java basada en el Project Object Model (POM), donde el archivo `pom.xml` define dependencias, plugins y fases del ciclo de vida como `compile`, `test` y `package`. Maven automatiza la descarga de librerías y ejecuta tareas de construcción y pruebas desde repositorios remotos.

Figura 3.35: Logotipo de Maven

3.3.4. Visualización de la base de datos

Durante el desarrollo local se ha utilizado MySQL Workbench (Figura 3.36) [51], una herramienta que permite diseñar, modelar y generar bases de datos de forma visual. Entre sus múltiples funcionalidades, incluye un editor SQL, la posibilidad de crear esquemas ERD y de realizar copias de seguridad.

Figura 3.36: Logotipo de MySQL Workbench

3.3.5. Peticiones API

Postman (Figura 3.37) [52] es una plataforma integral para el desarrollo y prueba de APIs que facilita la creación, envío y automatización de solicitudes

HTTP. Su interfaz gráfica permite definir solicitudes complejas con facilidad (incluyendo autenticación, parámetros y cuerpos de petición) y visualizar las respuestas de manera estructurada.

En este proyecto, Postman se ha utilizado para organizar y gestionar las pruebas de los endpoints de la aplicación. Se han creado carpetas dentro de colecciones para agrupar solicitudes por funcionalidades específicas y se han configurado entornos que permiten definir variables específicas para desarrollo y producción.

Figura 3.37: Logotipo de Postman

3.3.6. Entorno de desarrollo integrado (IDE)

El proyecto se ha desarrollado utilizando como editor de código Visual Studio Code. Se ha utilizado la extensión Git Graph para visualizar el historial de commits del repositorio, mostrando ramas, merges y etiquetas en un diagrama. En la figura 3.38 se puede apreciar una parte de la representación gráfica del historial del repositorio, lo que facilita el seguimiento de la evolución del proyecto.

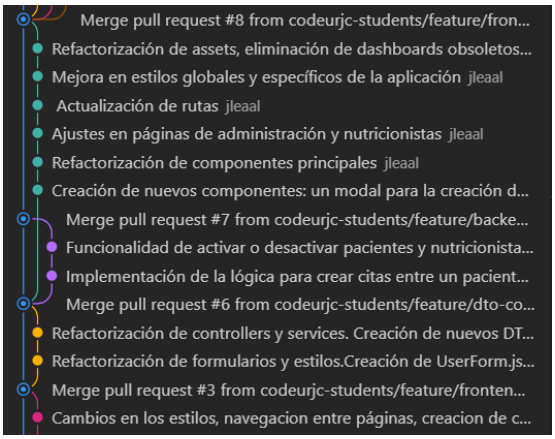


Figura 3.38: Diagrama del historial de commits, ramas y merges generado por la extensión Git Graph en Visual Studio Code.

3.3.7. Automatización y Despliegue

GitHub Actions [53] es el sistema de GitHub que permite definir flujos de trabajo como código en el propio repositorio, de modo que distintos eventos disparen procesos automatizados de compilación, pruebas y despliegue. En nuestra

arquitectura, GitHub Actions coordina la ejecución de análisis estático, pruebas, empaquetado de artefactos y el despliegue automático tanto del frontend (S3 y CloudFront) como del backend (Elastic Beanstalk).

AWS CLI [54] es una interfaz de línea de comandos unificada para gestionar servicios de AWS, que permite automatizar tareas como despliegues, gestión de recursos y sincronización de artefactos mediante comandos simples o scripts. En este proyecto, se ha empleado AWS CLI para automatizar operaciones tales como la subida de paquetes compilados a S3, la invalidación de cachés de CloudFront y la creación de versiones de aplicación en Elastic Beanstalk de forma consistente y sin intervención manual.

3.4. Metodologías

El proyecto se estructuró siguiendo una metodología iterativa e incremental. Se dividió el trabajo en iteraciones en las que se planificaron, diseñaron e implementaron las funcionalidades de la aplicación web. Durante el proceso de desarrollo, se tuvieron reuniones periódicas con el tutor, en las que se revisaron los avances de cada iteración y se validaron las funcionalidades implementadas.

3.4.1. Git Flow

El modelo de desarrollo utilizado en el proyecto ha sido Git Flow (Figura 3.39) [55], un modelo de gestión de ramas para Git que organiza y estructura el desarrollo en varias ramas de propósito específico, permitiendo gestionar de forma clara y escalable la evolución del código y las versiones de una aplicación.

En este modelo de desarrollo se han utilizado las siguientes ramas:

- **main**: Contiene el historial de versiones de producción. Se etiqueta con el número de versión correspondiente para reflejar el conjunto de versiones de la aplicación.
- **develop**: Rama de integración donde convergen todas las ramas `feature`. Representa el estado previo a la siguiente versión del proyecto. Cuando la rama `develop` está lista, se crea un `pull request` para fusionar los cambios con la rama `main`.
- **feature**: Se crean desde `develop` para trabajar en funcionalidades aisladas. Una vez completada la implementación, la rama se integra de nuevo en `develop`.
- **hotfix**: Se crea desde `main` para corregir fallos críticos en producción. Tras

realizar la corrección, se fusiona en `main` y en `develop` para incorporar los cambios en el desarrollo posterior.

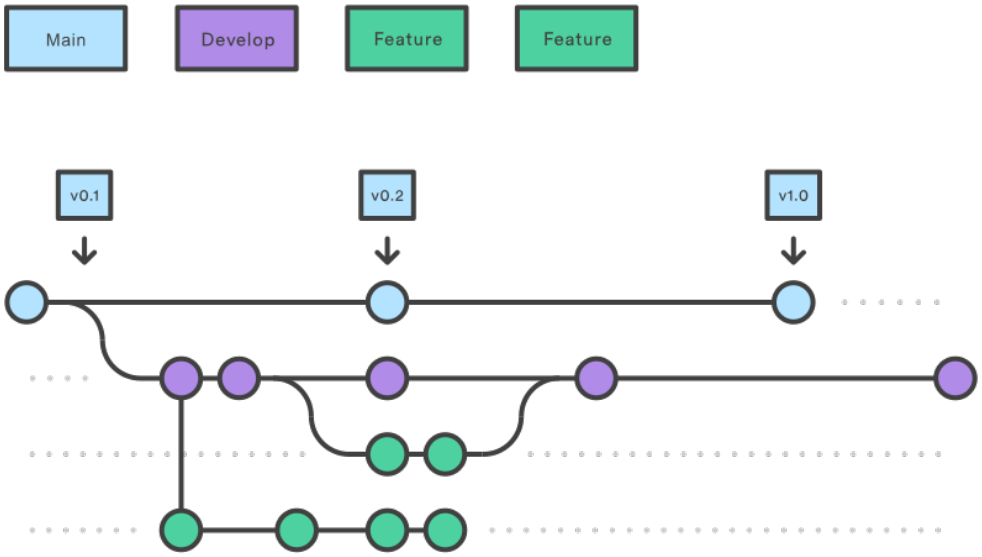


Figura 3.39: Flujo de trabajo Git Flow

3.4.2. Integración Continua y Despliegue Continuo (CI/CD)

La Integración Continua (CI) es una práctica de desarrollo de software que consiste en validar automáticamente cada cambio en el código al integrarlo en el repositorio remoto. Se debe comprobar que el proyecto con los nuevos cambios compila, que pasa el análisis estático y todas las pruebas antes de integrar las modificaciones. En este proyecto, el workflow de CI se activa con cada `push` a la rama `develop` y con cada *pull request* (PR) hacia `main`. De este modo, cualquier error de compilación, estilo o test en el backend y el frontend se detecta de inmediato, evitando que el código erróneo avance hacia producción.

El Despliegue Continuo (CD) es una práctica del desarrollo de software en la que, una vez el código ha superado todas las etapas de prueba, se empaqueta y se despliega de forma automática en producción sin intervención manual. En el proyecto, el workflow de CD se dispara con un `push` a la rama `main` (tras la fusión de la PR), donde se despliega el backend y después el frontend, sin intervención manual.

4

Descripción Informática

En este apartado se detalla la descripción técnica de la aplicación. En primer lugar, se presentan los requisitos funcionales, organizados por roles de usuario y los requisitos no funcionales. A continuación, se explica la arquitectura de la aplicación y el diseño, junto con algunas implementaciones de interés. Posteriormente, se detallan las pruebas y el análisis estático. Por último, se explica el flujo de integración continua y despliegue continuo mediante GitHub Actions, aportando una visión global del despliegue de la aplicación.

4.1. Requisitos

En esta sección se describen los requisitos que debe cumplir la aplicación web para garantizar su correcto funcionamiento y satisfacer las necesidades de todos los usuarios. Primero se detallan los requisitos funcionales, organizados por los distintos roles (auxiliar administrador, auxiliar, nutricionista y paciente), que definen las acciones que cada tipo de usuario puede realizar dentro del sistema. A continuación, se presentan los requisitos no funcionales, que establecen las características de calidad, seguridad, rendimiento y usabilidad.

4.1.1. Requisitos funcionales

Rol de auxiliar administrador (ADMIN)

- **RF1: Gestión de nutricionistas**: El auxiliar administrador podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los registros de nutricionistas.
- **RF2: Gestión de auxiliares**: El auxiliar administrador podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los registros de auxiliares.
- **RF3: Eliminación de pacientes**: El auxiliar administrador podrá eliminar registros de pacientes.
- **RF4: Control de acceso de nutricionistas**: El auxiliar administrador podrá activar o desactivar la cuenta de los nutricionistas para controlar su acceso.
- **RF5: Búsqueda de nutricionistas**: El auxiliar administrador podrá buscar a los nutricionistas por nombre, apellido, número de teléfono, correo electrónico y estado de la cuenta (activa o inactiva).
- **RF6: Búsqueda de auxiliares**: El auxiliar administrador podrá buscar a los auxiliares por nombre, apellido, número de teléfono y correo electrónico.
- **RF7: Ejecución de funcionalidades de auxiliar**: El auxiliar administrador podrá desempeñar todas las funcionalidades del rol de auxiliar.

Rol de auxiliar (AUXILIARY)

- **RF8: Modificación de perfil**: El auxiliar podrá modificar los datos de su perfil.
- **RF9: Visualización de agenda de nutricionistas**: El auxiliar podrá visualizar la agenda de cualquier nutricionista en forma de calendario.
- **RF10: Cambio de vista en la agenda**: El auxiliar podrá alternar la visualización de la agenda a vista diaria, semanal o mensual.
- **RF11: Ajuste de zoom en la agenda**: El auxiliar podrá ajustar el nivel de zoom de la agenda (ampliar o reducir el tamaño de las celdas).
- **RF12: Navegación a fecha concreta**: El auxiliar podrá ir directamente a una fecha concreta de la agenda.
- **RF13: Navegación temporal en la agenda**: El auxiliar podrá avanzar y retroceder días, semanas o meses en la agenda.

- **RF14: Gestión de bloqueos**: El auxiliar podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los bloqueos en la agenda de cualquier nutricionista.
- **RF15: Gestión de citas**: El auxiliar podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las citas de todos los nutricionistas.
- **RF16: Edición rápida de eventos**: El auxiliar podrá seleccionar y arrastrar las citas y los bloqueos en la agenda de cualquier nutricionista para modificar su fecha y hora de forma rápida.
- **RF17: Búsqueda de pacientes**: El auxiliar podrá buscar a los pacientes por nombre, apellido, número de teléfono, correo electrónico y estado de la cuenta (activa o inactiva).
- **RF18: Gestión de pacientes**: El auxiliar podrá crear, leer y actualizar los datos de pacientes.
- **RF19: Control de acceso de pacientes**: El auxiliar podrá activar o desactivar la cuenta de pacientes para controlar su acceso.
- **RF20: Alta asistida**: El auxiliar podrá ser dado de alta en la aplicación por un auxiliar administrador, recibiendo una contraseña por defecto que estará obligado a cambiar en su primer inicio de sesión.
- **RF21: Inicio de sesión**: El auxiliar podrá iniciar sesión en la aplicación con sus credenciales.
- **RF22: Cierre de sesión**: El auxiliar podrá cerrar sesión en la aplicación.
- **RF23: Recuperación de contraseña**: El auxiliar podrá recuperar su contraseña para poder acceder de nuevo a su cuenta.

Rol de nutricionista (NUTRITIONIST)

- **RF24: Modificación de perfil del nutricionista**: El nutricionista podrá modificar los datos de su perfil (incluyendo horario, duración de cita y número máximo de citas activas).
- **RF25: Visualización de agenda propia**: El nutricionista podrá visualizar únicamente su propia agenda de citas en forma de calendario.
- **RF26: Cambio de vista en la agenda**: El nutricionista podrá cambiar la visualización de la agenda a vista diaria, semanal o mensual.

- **RF27: Ajuste de zoom en la agenda**: El nutricionista podrá aumentar o disminuir el tamaño de las celdas de la agenda mediante una barra de zoom.
- **RF28: Navegación a fecha concreta**: El nutricionista podrá ir a una fecha concreta de la agenda.
- **RF29: Navegación temporal en la agenda**: El nutricionista podrá avanzar y retroceder días, semanas o meses en la agenda.
- **RF30: Configuración de duración de citas**: El nutricionista podrá definir el tiempo de sus citas, ajustando automáticamente el tamaño de las celdas de su agenda.
- **RF31: Restricciones de citas**: El nutricionista podrá establecer restricciones sobre el número máximo de citas pendientes por paciente.
- **RF32: Gestión de bloqueos propios**: El nutricionista podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los bloqueos en su propia agenda.
- **RF33: Gestión de citas propias**: El nutricionista podrá realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre las citas de su agenda.
- **RF34: Edición rápida de citas y bloqueos**: El nutricionista podrá seleccionar y arrastrar las citas y los bloqueos en su agenda para modificar rápidamente su fecha y hora.
- **RF35: Búsqueda de pacientes**: El nutricionista podrá buscar a los pacientes por nombre, apellido, número de teléfono, correo electrónico y estado de la cuenta (activa o inactiva).
- **RF36: Gestión de pacientes**: El nutricionista podrá crear, leer y actualizar los datos de pacientes.
- **RF37: Control de acceso de pacientes**: El nutricionista podrá activar o desactivar la cuenta de pacientes para controlar su acceso.
- **RF38: Alta asistida**: El nutricionista podrá ser dado de alta en la aplicación por un auxiliar administrador, recibiendo una contraseña por defecto que estará obligado a cambiar en su primer inicio de sesión.
- **RF39: Inicio de sesión**: El nutricionista podrá iniciar sesión en la aplicación con sus credenciales.
- **RF40: Cierre de sesión**: El nutricionista podrá cerrar sesión en la aplicación.

- **RF41: Recuperación de contraseña**: El nutricionista podrá recuperar su contraseña para poder acceder de nuevo a su cuenta.

Rol de paciente (PATIENT)

- **RF42: Modificación de perfil**: El paciente podrá modificar los datos de su perfil.
- **RF43: Creación de citas en agenda de nutricionistas**: El paciente podrá citarse en la agenda de cualquier nutricionista, seleccionando fecha y hora disponibles y respetando los límites de citas pendientes del nutricionista.
- **RF44: Notificación de reserva**: El paciente recibirá un correo electrónico al reservar una cita con la información de esta.
- **RF45: Visualización de citas pendientes**: El paciente podrá ver sus citas pendientes, con la información del nutricionista y la fecha y hora de la cita.
- **RF46: Cancelación de citas**: El paciente podrá cancelar sus citas pendientes.
- **RF47: Registro autónomo**: El paciente podrá darse de alta en la aplicación rellenando el formulario de registro.
- **RF48: Alta asistida**: El paciente podrá ser dado de alta en la aplicación por un trabajador de la entidad, recibiendo una contraseña por defecto que estará obligado a cambiar en su primer inicio de sesión.
- **RF49: Inicio de sesión**: El paciente podrá iniciar sesión en la aplicación con sus credenciales.
- **RF50: Cierre de sesión**: El paciente podrá cerrar sesión en la aplicación.
- **RF51: Recuperación de contraseña**: El paciente podrá recuperar su contraseña para poder acceder de nuevo a su cuenta.

4.1.2. Requisitos no funcionales

- **RNF1:** El sistema debe ser fácil de utilizar e intuitivo.
- **RNF2:** El sistema debe solicitar confirmación explícita al usuario antes de ejecutar cualquier acción decisiva o irreversible.
- **RNF3:** El sistema debe estar compuesto por un frontend SPA en React y un backend RESTful en Spring Boot.

- **RNF4:** El sistema debe autenticar y autorizar a los usuarios mediante JWT emitidos por AWS Cognito.
- **RNF5:** El sistema debe implementar control de acceso basado en roles usando los grupos de Cognito, donde se deben poder realizar acciones diferentes según el rol.
- **RNF6:** El sistema debe permitir peticiones CORS configuradas correctamente en Spring Security.
- **RNF7:** El sistema debe enviar correos electrónicos de forma asíncrona con AWS SES.
- **RNF8:** El sistema debe ofrecer una interfaz responsive para los pacientes.
- **RNF9:** El sistema debe contar con una batería de pruebas unitarias y de integración en el repositorio, integradas en un pipeline de CI/CD para garantizar la calidad del código.
- **RNF10:** El sistema debe utilizar Amazon RDS con motor MySQL para el almacenamiento centralizado de datos.

4.2. Arquitectura y análisis

4.2.1. Arquitectura de la aplicación

En la figura 4.1 se presenta una visión general de la arquitectura de la aplicación, mostrando cómo los distintos servicios de AWS se combinan para ofrecer una solución completa y segura.

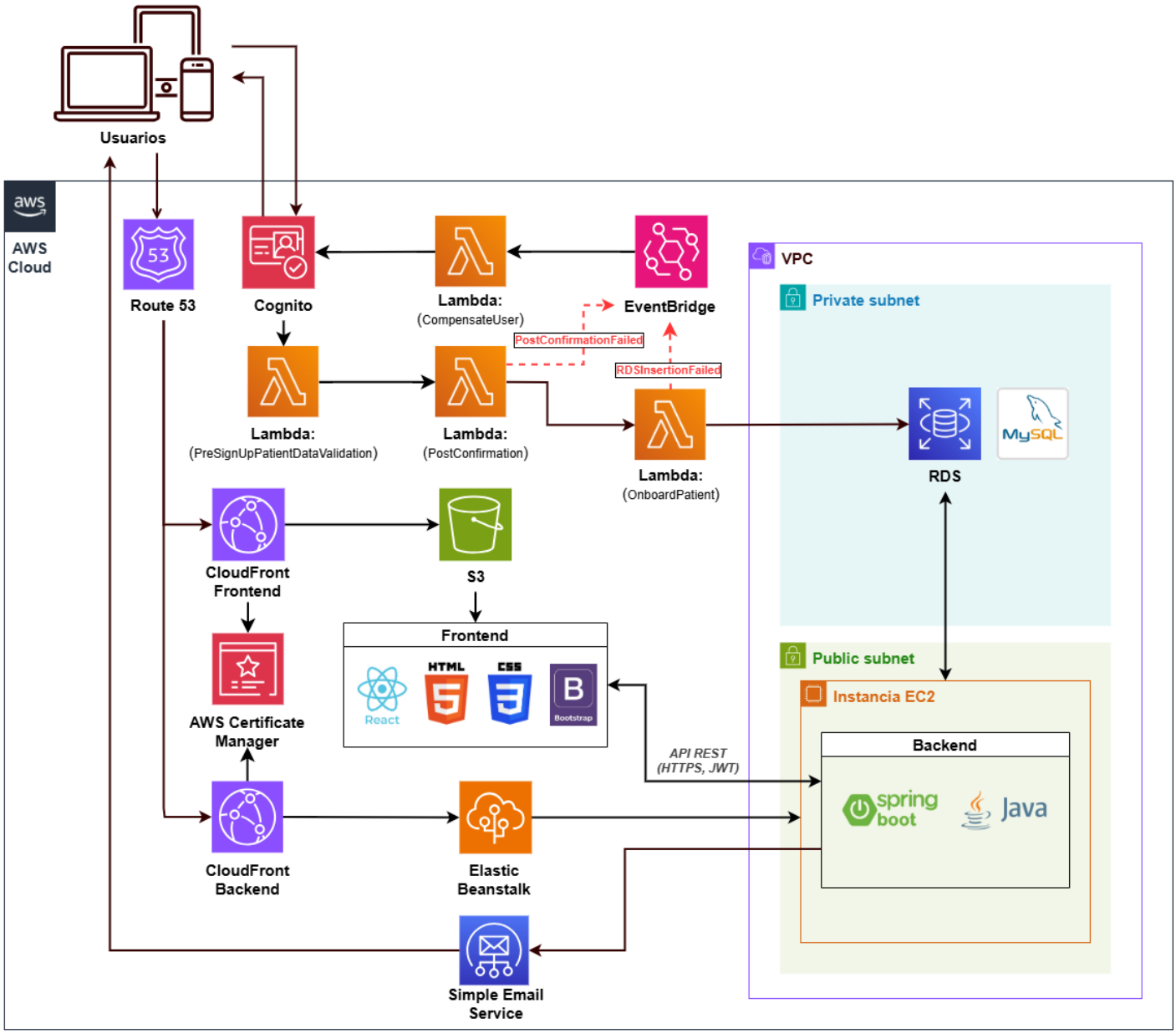


Figura 4.1: Arquitectura de la aplicación

Punto de entrada y enrutamiento

La capa de entrada al sistema se apoya en Route 53, el servicio de DNS de AWS que dirige el dominio `clinicanutricion.es` y sus subdominios hacia las distribuciones de CloudFront configuradas: una para el frontend (`www.clinicanutricion.es`) y otra para la API (`api.clinicanutricion.es`). Ambas distribuciones utilizan certificados TLS gestionados por AWS Certificate Manager (ACM), que emite y renueva automáticamente estos certificados, garantizando cifrado HTTPS sin necesidad de manejar manualmente ningún par de claves.

Frontend estático

La aplicación cliente es una SPA desarrollada con React, HTML, CSS y Bootstrap, almacenada en un *bucket* S3 configurado para servir contenido estático. CloudFront distribuye estos archivos desde sus *edge locations* globales, reduciendo la latencia y mejorando la experiencia de carga. Además, CloudFront aplica el certificado TLS de ACM y añade las cabeceras de seguridad necesarias.

Backend implementado en Java

El backend está implementado con Spring Boot sobre una instancia EC2 gestionada por Elastic Beanstalk, ubicada en una subred pública dentro de la Virtual Private Cloud (VPC). Elastic Beanstalk simplifica despliegues, el escalado manual, la configuración de variables de entorno y la visualización de los *logs* de la aplicación. Todas las políticas CORS se centralizan en el backend, evitando duplicidades y garantizando que solo los orígenes autorizados accedan a la API.

Cada petición a la API (`api.clinicanutricion.es`) pasa primero por una distribución de CloudFront configurada como *proxy* inverso[1] y que utiliza cifrado TLS. A continuación, se reenvía el token JWT al servicio de Spring Boot. Spring Security valida ese token consultando AWS Cognito para asegurar que el usuario esté autenticado y autorizado para la operación solicitada.

Para mayor seguridad, se habría podido colocar un balanceador de carga en una subred pública y situar la instancia del backend en subredes privadas sin IP pública. De esta forma, el tráfico pasaría primero por el balanceador de carga, evitando el acceso directo al backend. Sin embargo, esta opción no está incluida en la capa gratuita de AWS.

[1] servidor intermedio que recibe las solicitudes de los clientes y las reenvía a uno o varios servidores de origen, ocultando su identidad y gestionando tareas como balanceo de carga, cacheo y seguridad.

Base de datos

La base de datos se configura como una instancia RDS con motor MySQL, alojada en una subred privada de la misma VPC para aislarla del acceso público.

Autenticación y autorización

AWS Cognito centraliza el registro de usuarios, el inicio de sesión y la gestión de roles. Desde el frontend, la aplicación redirige al usuario al servicio de *login* de Cognito y, tras validar sus credenciales, obtiene un token JWT que se almacena en el navegador. Ese token se envía en cada petición al backend, donde se comprueba su validez y se extraen los permisos del usuario para decidir a qué partes de la API puede acceder.

Integraciones con AWS Lambda y EventBridge

Se han creado varias funciones Lambda asociadas a los *triggers* de Cognito o a eventos de error publicados en EventBridge, que forman parte del flujo de auto-registro de pacientes a través de la UI de Cognito y se explicarán en profundidad en el apartado de "Procesos de Registro de Usuarios en la Aplicación". Cada función Lambda ejecuta con su propio IAM Role, que contiene únicamente los permisos necesarios para interactuar con Cognito o la base de datos RDS, siguiendo el principio de mínimos privilegios. Todos los registros de ejecución y los errores de las funciones Lambda se envían a CloudWatch Logs, lo que centraliza toda la información y facilita el diagnóstico en tiempo real de fallos.

Envío de correos electrónicos

El envío de correos electrónicos con la información de la cita de nutrición se realiza mediante Amazon SES. La instancia EC2 asume un IAM Role con permisos para SES y el backend invoca un método asíncrono que construye y envía el correo electrónico al paciente.

4.2.2. Frontend

El frontend se ha desarrollado como una SPA con React. Los estilos se han gestionado mediante CSS y Bootstrap. A continuación, se detalla la organización del frontend, dividida en módulos principales, en los que se describe el propósito de cada uno y sus componentes más relevantes.

Recursos estáticos (`assets`)

Este directorio contiene todos los recursos estáticos de imagen utilizados en la interfaz, como el logotipo de la clínica de nutrición, iconos genéricos o para la barra lateral de la aplicación.

Componentes reutilizables (`components`)

En este directorio se agrupan elementos reutilizables de la interfaz de usuario que no pertenecen a una ruta concreta. Algunos de los componentes principales son:

- **NutritionistCalendar.js**: Componente que muestra un calendario interactivo con *drag-and-drop* de citas para nutricionistas. Carga eventos desde la API y permite crear, mover y borrar citas o bloqueos a través de menús contextuales y modales.
- **SearchComponent.js**: Componente que centraliza la búsqueda de usuarios (pacientes, nutricionistas o auxiliares). Proporciona un formulario de filtros (nombre, apellidos, teléfono, correo y estado de la cuenta) y muestra una tabla paginada con acciones como editar, eliminar o cambiar el estado de la cuenta. Incluye un modal de confirmación para las eliminaciones, notificaciones de éxito y error, validación de filtros y controles de paginación para navegar entre páginas.
- **Sidebar.js**: Componente que gestiona la navegación en una barra lateral en la aplicación web. Muestra iconos con las rutas disponibles según el rol del usuario para acceder a cada sección (perfil, agenda, gestión de usuarios y cierre de sesión).
- **UserForm.js**: Formulario reutilizable para crear o editar usuarios, que ajusta los campos según el rol.

Vistas principales (`pages`)

Este directorio organiza las vistas de la aplicación según el perfil y la funcionalidad, centralizando en `pages/management/` las pantallas de gestión de usuarios y agendas utilizadas por administradores y auxiliares, e incluso algunas vistas empleadas también por los nutricionistas para la gestión de pacientes.

- **management**: Agrupa las pantallas de gestión de usuarios y de las agendas de los diferentes nutricionistas.

- **ManagementAgendas.js**: Es la pantalla encargada de gestionar las agendas de los nutricionistas. Dispone de un campo de búsqueda predictiva que, a medida que se escribe, muestra sugerencias basadas en coincidencias de nombre y apellido de nutricionistas. Al escoger un nutricionista, se utiliza el componente `NutritionistCalendar.js` con la configuración del horario y la duración de cita de ese nutricionista.
- **ManageUsers.js**: Contiene la pantalla para administrar usuarios, mostrando botones que enlazan a las subrutas de pacientes, nutricionistas y auxiliares.
- **ManageAuxiliaries.js**, **ManageNutritionists.js**, **ManagePatients.js**: Pantallas para gestionar auxiliares, nutricionistas y pacientes, respectivamente. Cada una incluye botones para navegar a las rutas de creación y búsqueda de la entidad correspondiente.

- **nutritionist**: Este directorio contiene únicamente `NutritionistAgenda.js`, que muestra la agenda personal del nutricionista autenticado.

- **patient**: Dentro de este directorio se encuentran todas las pantallas que forman parte del flujo de creación y gestión de citas por parte de los pacientes.

  - **MainPatientScreen.js**: Página principal a la que accede el paciente tras iniciar sesión. Incluye botones para ver los datos de su perfil, solicitar una cita o consultar las citas pendientes.
  - **NutritionistSelection.js**: Se muestra al pulsar "Pedir cita". Permite seleccionar una franja horaria (mañana, mediodía y tarde), actualizando un desplegable que muestra los nutricionistas que trabajan en la franja horaria seleccionada.
  - **TimeSelection.js**: Después de seleccionar la franja horaria y el nutricionista, se muestra un calendario con los huecos disponibles de cada día en la franja elegida. Al escoger día y hora, el paciente navega a la pantalla de confirmación de cita.
  - **AppointmentConfirmation.js**: Muestra los datos de la cita (paciente, nutricionista, fecha y hora) y un botón para confirmar y reservar.
  - **PendingAppointments.js**: Presenta un listado de las citas próximas del paciente, donde se muestra la información de cada reserva y un botón para cancelar la cita.

Rutas protegidas y autorización (`routes`)

- **ProtectedRoute.js**: Es un componente de React que protege rutas mediante autenticación. Verifica si el usuario está autenticado y dispone de al

menos uno de los roles permitidos en su perfil OIDC[2]. En caso contrario, redirige a la página de acceso no autorizado.

- **Routes.js**: Organiza la navegación declarando las rutas dinámicas de la SPA, asignando componentes de vistas según las URLs. Integra lógica de roles para pacientes, nutricionistas, auxiliares y administradores, redirigiendo según el rol del usuario y renderizando la barra lateral de navegación.

Organización de hojas de estilo (`styles`)

El directorio agrupa y organiza las hojas de estilos en tres ámbitos claramente diferenciados. En `global.css` se definen los estilos y normas generales. En el subdirectorio `styles/components/` cada componente reutilizable (agenda, formularios, modal de citas, barra lateral, notificaciones, etc.) dispone de su propio archivo CSS. Por último, en `styles/pages/` cada pantalla principal (*login*, panel de paciente, selección de nutricionista, confirmación de cita, gestión de usuarios, gestión de agendas, etc) incorpora estilos específicos de diseño.

Componentes principales en la raíz del proyecto

- **App.js**: Actúa como componente principal que gestiona la experiencia del usuario en función del estado de autenticación. Emplea `react-oidc-context` para detectar si el usuario está autenticado y, de no ser así, muestra una pantalla de inicio con un botón de inicio de sesión y otro que redirige al formulario de registro. Tras la autenticación, `App.js` invoca `AppRoutes` para establecer la navegación protegida por roles.

- **index.js**: Define los parámetros OIDC para AWS Cognito en `cognitoAuthConfig` y utiliza `AuthProvider` para proporcionar el contexto de autenticación a todos los componentes. Con `BrowserRouter` envuelve la aplicación en un `Router` para manejar rutas de React. El componente `AuthRedirect` detecta cambios en el estado de autenticación, guarda el token JWT en `localStorage` y redirige al usuario según su grupos (*admin*, *nutritionist*, *patient*, *auxiliary*) o a la página de no autorizado.

4.2.3. Backend

El backend se ha desarrollado utilizando Spring Boot siguiendo el patrón arquitectónico MVC. Los controladores exponen la API REST, la capa de servicios

[2]conjunto de atributos de identidad (*claims*) que un proveedor OpenID Connect incluye en el token.

gestiona la lógica de negocio y las integraciones con servicios externos y los repositorios manejan la persistencia en MySQL. La figura 4.2 muestra la estructura de carpetas del backend de la aplicación.

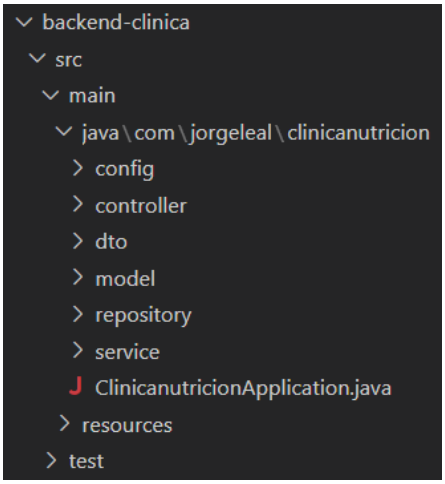


Figura 4.2: Estructura Backend

1. **Seguridad**: La configuración de seguridad se encarga de proteger toda la API definiendo reglas de origen, autenticación y autorización de forma centralizada. Se habilita CORS, restringiendo orígenes, cabeceras y métodos HTTP al frontend autorizado. Las rutas públicas, como el endpoint de salud o el de registro y *login*, quedan abiertas, mientras que el resto exige un token JWT válido. A continuación, se aplican permisos basados en grupos extraídos de Cognito, asignados a cada agrupación de recursos (pacientes, citas, nutricionistas, etc). Por último, esos grupos se convierten en autoridades de Spring Security, de modo que el propio framework valida el token y controla el acceso a cada endpoint.

2. **Controladores**: Los controladores actúan como interfaz entre las peticiones HTTP y la lógica de negocio. Exponen *endpoints* REST para entidades (pacientes, nutricionistas, auxiliares, administradores y citas), aplican validación de seguridad y delegan operaciones a la capa de servicios.

3. **Servicios**: Los servicios implementan la lógica de negocio y evitan la duplicación de código. Se utilizan para transmitir información al repositorio e interactuar con clientes externos, como Cognito o SES.

4. **Repositorios**: Los repositorios actúan como puente entre la lógica de negocio y la base de datos, ofreciendo de forma automática operaciones CRUD (creación, lectura, actualización y borrado). Se utiliza para definir consultas personalizadas mediante la convención de nombres de Spring Data.

5. **Modelo**: Las clases de entidad JPA (`User`, `Patient`, `Nutritionist`, `Auxiliary`, `AdminAuxiliary` y `Appointment`) definen los atributos y relaciones (uno a uno, uno a muchos) del dominio. Se utilizan enumeraciones como `UserType`, `Gender` y `AppointmentType` para restringir valores. Este modelo constituye el esquema central que mapea los objetos de la aplicación a las tablas de la base de datos.

6. **DTOs**: Los DTOs (*Data Transfer Objects*) encapsulan y transportan información entre capas. Su uso permite validar los datos de entrada, evitar exponer directamente las entidades de dominio y estructurar la salida en el formato JSON deseado.

4.2.4. Base de datos

La base de datos en producción es una RDS con motor MySQL, un sistema de gestión de bases de datos relacional. A continuación, se detallarán las tablas utilizadas y se mostrará un diagrama entidad-relación de la base de datos en la figura 4.3.

- **user**: entidad principal de la aplicación, donde se almacena la información común a todos los tipos de usuarios.
- **patient**: corresponde a los usuarios con perfil de paciente.
- **nutritionist**: representa a los nutricionistas de la clínica.
- **auxiliary**: corresponde a los auxiliares de la clínica.
- **admin_auxiliary**: representa a los administradores de la clínica.
- **appointment**: almacena las citas entre pacientes y nutricionistas.

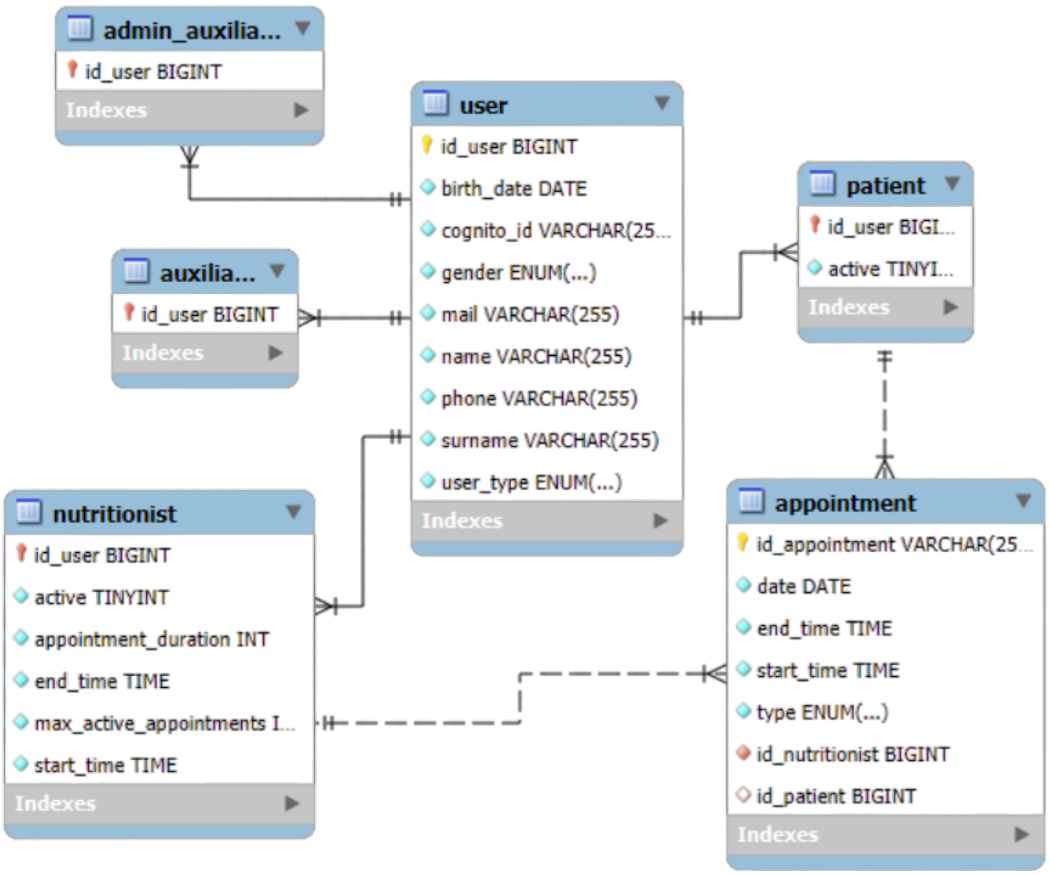


Figura 4.3: Diagrama entidad-relación (ER)

4.3. Diseño e implementación

4.3.1. Frontend

Implementación del Calendario de Nutricionista

Una vez explicada la estructura del frontend y el propósito de cada módulo, me gustaría explicar la implementación del componente principal `NutritionistCalendar.js`. Este componente ofrece una interfaz interactiva de un calendario para gestionar las citas y bloqueos en la agenda de un nutricionista. En la figura 4.4 se muestra el componente integrado en la pantalla `ManagementAgendas.js`. Las propiedades `nutritionistId`, `appointmentDuration`, `startTime` y `endTime` permiten configurar dinámicamente el calendario según cada nutricionista, ajustando la duración de las citas y su jornada laboral. Para sincronizar el estado interno del componente con el backend y la vista del usuario, se han combinado diversos *hooks*. Todas las peticiones realizadas al backend incluyen el token de autenticación para verificar la identidad y permisos del usuario.

Figura 4.4: Vista de la pantalla `ManagementAgendas.js`

- **Carga de citas**: Al iniciar el calendario, cambiar de nutricionista o modificar la fecha, se invoca automáticamente `fetchAppointments`. Esta función realiza una petición al *endpoint* correspondiente, procesa la respuesta JSON mapeándola con la estructura necesaria para crear eventos en el calendario y actualiza el estado interno del componente para mostrar las citas y bloqueos del nutricionista.

- **Arrastrar y soltar (*Drag and Drop*)**: Se ha utilizado la librería `react-dnd` para permitir que en el calendario se puedan mover citas con arrastrar y soltar. Al cambiar de posición un evento, se comprueba que la nueva fecha no sea anterior a la actual, se actualiza el estado local del componente y se envía la petición al servidor con los nuevos horarios. Si la operación falla porque coincide con otra cita, se revierte el estado del componente y se muestra una notificación de error.

- **Menú contextual**: Al hacer clic derecho sobre un evento o un espacio libre, `handleRightClick` despliega un menú emergente en la posición del cursor. El componente intercepta el clic derecho para evitar que aparezca el menú del navegador y en su lugar captura la posición exacta y la información de si el elemento es un evento existente o un hueco libre. Si se ha hecho clic sobre un evento, el menú muestra la opción de "Eliminar" y si es un hueco libre, muestra la opción de "Nueva Cita" y "Nuevo Bloqueo". El menú se cierra al seleccionar una opción o al hacer clic en cualquier otro lugar.

- **Creación y eliminación de eventos**: Al seleccionar "Nueva Cita" o "Nuevo Bloqueo" en el menú, se calcula automáticamente la hora de fin sumando la variable del nutricionista `appointmentDuration` y se prepara un objeto con todos los datos (nutricionista, fecha, horario y en el caso de que sea una cita y no un bloqueo, el paciente). Si se ha seleccionado "Nueva Cita", la función `handleNewAppointment` mostrará el componente `NewAppointmentModal`, donde se selecciona al paciente filtrándolo por nombre, apellido, correo o teléfono. Tras confirmar, se envía una petición HTTP `POST` al backend y si es exitosa, se recargan los eventos del calendario. Al pulsar "Eliminar" en el menú, la función `handleDeleteEvent` envía una petición de borrado al backend y actualiza las citas para reflejar los cambios.

- **Navegación**: Se incluye un selector de fecha de `react-datepicker`, un botón "Hoy" y flechas de navegación que avanzan o retroceden la vista según el modo activo (diario, semanal o mensual).

- **Zoom y altura de franjas horarias**: El slider de Bootstrap ajusta el estado `zoom`, que modifica la variable CSS `--calendar-slot-height` mediante un `useEffect`. De este modo, la altura de cada franja horaria crece o disminuye proporcionalmente, permitiendo acercar o alejar la vista de forma sencilla para el usuario.

Confirmación de acciones y notificaciones

Con el objetivo de reforzar la seguridad y evitar la eliminación accidental de recursos por parte de los usuarios, se han utilizado modales implementados con el componente `<Modal>` de `react-bootstrap`. Por ejemplo, en el componente de

búsqueda de usuarios (`SearchComponent.js`) se ofrece la opción de poder activar o desactivar la cuenta de pacientes y nutricionistas. Esta es la opción más habitual, ya que así se mantiene el histórico de citas; en cambio, los administradores también tienen la opción de eliminar al usuario por completo, borrando sus citas asociadas. Para prevenir borrados involuntarios, se muestra el modal de confirmación que aparece en la figura 4.5. De igual modo, en la pantalla de citas pendientes del paciente (`PendingAppointments.js`), al pulsar el botón para eliminar una cita, aparece el modal de la figura 4.6, que obliga al paciente a confirmar o descartar los cambios antes de enviar la petición al servidor.

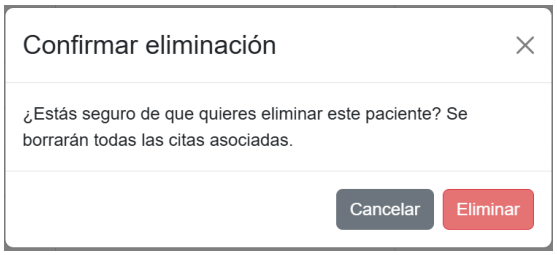


Figura 4.5: Modal de confirmación de eliminación de paciente y sus citas asociadas.

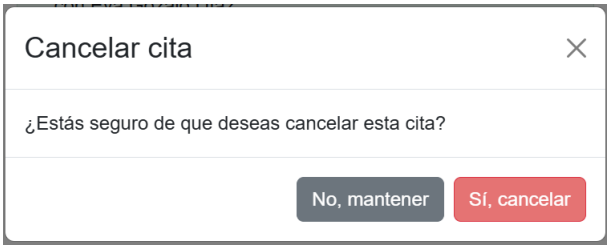


Figura 4.6: Modal de confirmación de cancelación de cita.

En la aplicación se han utilizado notificaciones de tipo *toast* mediante la librería `react-toastify`. Estas notificaciones emergentes sirven para informar al usuario de forma inmediata y no intrusiva sobre el resultado de sus acciones. Al aparecer en la esquina de la pantalla y desaparecer tras unos segundos, no se necesitan usar modales adicionales ni hace falta recargar la página. Por ejemplo, tras confirmar una cita en la pantalla de `AppointmentConfirmation.js`, se muestra la notificación de éxito de la figura 4.7 y al intentar mover un evento en `NutritionistCalendar`, aparece la notificación de error por solapamiento mostrada en la figura 4.8.

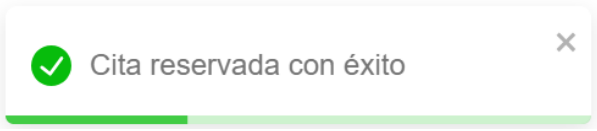


Figura 4.7: Notificación de cita confirmada con éxito.

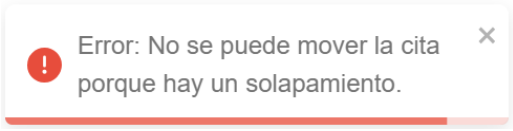


Figura 4.8: Notificación de error por solapamiento de cita.

4.3.2. Backend

Configuración por entornos: perfiles de Spring y Elastic Beanstalk

En el backend se ha optado por externalizar la configuración de la aplicación para garantizar la máxima flexibilidad y seguridad. Las variables de configuración y las credenciales sensibles no se incluyen directamente en el código fuente, sino que se inyectan desde ficheros de perfil o desde variables de entorno en tiempo de ejecución.

Para el entorno de pruebas se ha definido un perfil test, que se activa mediante la notación `@ActiveProfiles("test")`. Al usar este perfil, Spring carga `application-test.yml`, en el que se configura una base de datos H2 en memoria. Esto permite ejecutar las pruebas de forma aislada sin hacer uso de los recursos de desarrollo o producción.

Cuando no se especifica ningún perfil, Spring utiliza el perfil por defecto (desarrollo), utilizando el archivo `application.yml` y rellenando sus parámetros mediante la clase `EnvLoader`, obteniendo valores de las variables de entorno definidas en el archivo `.env`. En este entorno se utiliza una base de datos MySQL y un *pool* de Cognito de desarrollo. Sin embargo, se ha optado por no definir un perfil de producción específico, aprovechando que Elastic Beanstalk inyecta variables de entorno que tienen que ser definidas en su configuración. En producción, se utiliza una base de datos RDS con motor MySQL y un *pool* de Cognito de producción.

Envío de correos electrónicos mediante Amazon SES

En este proyecto, el envío de los correos de confirmación de cita se ha delegado a Amazon SES para evitar mantener un servidor de correo propio. La clase `EmailService` crea un `SesClient` del SDK v2 de AWS configurado con la misma región que el resto de la arquitectura. El `SesClient` permite interactuar con Amazon Simple Email Service (SES), gestionando internamente la región y las credenciales para enviar correos electrónicos de forma sencilla y segura.

Para permitir que la aplicación desplegada en Elastic Beanstalk utilice SES sin almacenar claves, se ha tenido que configurar previamente un rol IAM con una política que permita usar Amazon SES. Este rol se asigna a la instancia de la aplicación, de modo que al desplegar en producción, el SDK de AWS detecta automáticamente las credenciales. En el entorno de desarrollo, obtiene las credenciales del archivo `$HOME/.aws/credentials` tras configurar el AWS CLI.

A continuación, se muestra la inicialización del cliente SES con obtención automática de credenciales en el código 4.3.1.

```
private final SesClient ses;

public EmailService(@Value("${AWS_REGION}")
    String awsRegion) {
     this.ses = SesClient.builder()
         .region(Region.of(awsRegion))
         .build();
}
```

Código 4.3.1: Inicializa el cliente SES obteniendo credenciales automáticamente

La lógica de envío se encapsula en un método asíncrono, de modo que la operación no bloquea el hilo principal de la aplicación. Este método recibe como parámetros el correo del paciente, su nombre, la fecha y la hora de la cita, y construye un HTML que incluye un saludo personalizado y los datos de la cita.

Cabe destacar que se ha utilizado Amazon SES en modo *sandbox*, lo que restringe el envío de correos únicamente a direcciones que han sido verificadas previamente en la sección de identidades verificadas de la consola de SES. Para eliminar esa limitación, simplemente hay que solicitar el acceso a producción en la consola de SES, permitiendo enviar a cualquier destinatario.

Implementación del servicio de usuarios con AWS Cognito

La finalidad de esta clase es abstraer las operaciones administrativas sobre el *User Pool* de Cognito: creación, actualización, eliminación, activación y desactivación de usuarios, así como el cierre de sesión global.

Para conectar con Cognito, la clase inicializa un cliente `CognitoIdentityProviderClient` del SDK v2 de AWS. La obtención de credenciales es idéntica a la usada en Amazon SES, por lo que se ha definido un rol IAM con una política que permita a la instancia EC2 del Elastic Beanstalk hacer uso de Cognito.

El método para crear un usuario en Cognito construye una petición `AdminCreateUserRequest` con los atributos del usuario, asigna una contraseña temporal que debe cambiarse obligatoriamente en el primer inicio de sesión y añade al usuario al grupo correspondiente en Cognito (*patient*, *nutritionist*, *auxiliary* o *admin*).

Los métodos para actualizar, eliminar, habilitar o deshabilitar un usuario, así como para realizar el cierre de sesión global, funcionan de forma similar. Primero se construye el objeto de petición correspondiente, como `AdminUpdateUserAttributesRequest`, aportando los parámetros `userPoolId` y `username` para identificar la cuenta. A continuación, se invoca el método del

cliente de Cognito (`adminUpdateUserAttributes`), de modo que la complejidad de la llamada HTTP, la autenticación y el manejo de respuestas y errores queda gestionada por el SDK.

4.3.3. Seguridad

Autenticación y autorización mediante AWS Cognito

En el frontend, la integración con AWS Cognito se basa en OpenID Connect (OIDC) y JWT para delegar la autenticación sin tener que gestionar contraseñas ni sesiones en el propio servidor. Al iniciar la aplicación, la librería `react-oidc-context` se configura con el dominio de Cognito, el identificador de cliente y las URI de redirección. Cuando el usuario quiere iniciar sesión, se le redirige al dominio de Cognito, donde se muestra la interfaz de autenticación. Dado que `react-oidc-context` no proporciona un método para redirigir al proceso de registro de Cognito, ha sido necesario construir la URL de forma manual.

Una vez autenticado, Cognito devuelve un token JWT al frontend. Este token incluye información del usuario, como su `sub` (ID de Cognito) y el grupo. El token se almacena en `localStorage` para incluirlo en las cabeceras de las solicitudes al backend, de modo que el servidor valide que el usuario está autenticado.

La autorización en el frontend se basa en roles, que se recuperan de Cognito mediante el *claim* `cognito:groups`. En el código 4.3.2 se muestra el componente `ProtectedRoute`, que verifica si el usuario está autenticado y pertenece a alguno de los roles permitidos; en caso contrario, redirige a la pantalla de inicio de sesión o a la página de acceso no autorizado.

```
1  const ProtectedRoute = ({ allowedRoles }) => {
2    const auth = useAuth();
3    if (!auth.isAuthenticated || !auth.user) {
4      return <Navigate to="/" />;
5    }
6    const roles =
       auth.user?.profile['cognito:groups'] || [];
7    if (!roles.some((role) =>
       allowedRoles.includes(role))) {
8      return <Navigate to="/unauthorized" />;
9    }
10   return <Outlet />;
11 };
12 export default ProtectedRoute;
```

Código 4.3.2: Componente en React que verifica autenticación y autoriza el acceso según los roles permitidos.

En el backend, la API se configura como *Resource Server* OAuth 2.0 mediante Spring Security para que todas las peticiones requieran un token JWT emitido por Cognito. Para ello se define en el fichero de configuración de propiedades el `issuer-uri`, que identifica al emisor del token de Cognito, y el `jwk-set-uri`, que proporciona las claves públicas para que el backend pueda verificar la firma de los tokens JWT. De esta forma, Spring obtiene automáticamente dichas claves, comprueba la firma y verifica la validez temporal de cada token.

Por defecto, Spring extrae las autoridades de los *claims* `scope` o `authorities`, pero como Cognito almacena los grupos en `cognito:groups`, se ha implementado un `JwtAuthenticationConverter` (Código 4.3.3) que transforma esos grupos en *granted authorities* con prefijo `ROLE_`. De este modo, Spring reconocerá automáticamente los roles definidos en Cognito y los asociará a las peticiones entrantes.

```
@Bean
public JwtAuthenticationConverter jwtAuthenticationConverter() {
    JwtGrantedAuthoritiesConverter grantedAuthoritiesConverter
       = new JwtGrantedAuthoritiesConverter();

    JwtAuthenticationConverter jwtConverter = new
       JwtAuthenticationConverter();
    jwtConverter.setJwtGrantedAuthoritiesConverter(jwt -> {
        Collection<String> authorities =
           jwt.getClaimAsStringList("cognito:groups");
        if (authorities == null) {
            return grantedAuthoritiesConverter.convert(jwt);
        }
        return authorities.stream()
                .map(role -> new SimpleGrantedAuthority("ROLE_"
                   + role.toUpperCase()))
                .collect(Collectors.toList());
    });

    return jwtConverter;
}
```

Código 4.3.3: Bean `JwtAuthenticationConverter`

La autorización de rutas se configura en el método `securityFilterChain`. Primero, se aplica la política de CORS y se desactiva la protección CSRF. A continuación, en la llamada `authorizeHttpRequests`, se definen los patrones de URL y las autoridades necesarias para cada ruta; el resto requiere autenticación con `.anyRequest().authenticated()`. Seguidamente, se integra el `JwtAuthenticationConverter`, que transforma los grupos de Cognito en *granted authorities*. Para reforzar la seguridad a nivel de método, se habilita `@EnableMethodSecurity` y se utilizan anotaciones como `@PreAuthorize("hasRole('ROLE_NUTRITIONIST')")` en los controladores, para que cada operación solo se ejecute si el usuario posee el rol correspondiente.

CORS

La configuración de CORS se realiza de forma centralizada en la clase `SecurityConfig`, aprovechando el soporte nativo de Spring Security y su integración con Spring MVC. Se define un *bean* de tipo `CorsConfigurationSource` que encapsula todos los detalles de política CORS. En este se establece como único origen permitido la URL del frontend, se especifican las cabeceras que el navegador puede enviar y las que la API expondrá en la respuesta, se habilitan los métodos HTTP básicos y se activa el envío de cabeceras de autenticación en peticiones CORS.

Para aplicar la configuración, se integra la política de CORS dentro de la cadena de filtros de seguridad de Spring. Se ha permitido expresamente que las peticiones `OPTIONS` lleguen sin autenticación, ya que este requisito es obligatorio para que los navegadores completen correctamente el *preflight* de CORS.

Con este enfoque, cualquier llamada que el frontend realice a la API respetará la política definida, recibiendo únicamente respuestas si su origen coincide con la URL configurada, utiliza métodos permitidos y envía cabeceras autorizadas.

Grupos de seguridad

Se han configurado grupos de seguridad que permiten definir reglas de acceso basadas en protocolos, rangos de direcciones IP o incluso otros grupos de seguridad. De este modo, se garantiza que cada componente del sistema únicamente establezca comunicaciones con los servicios autorizados. Por ejemplo, la RDS permite únicamente tráfico entrante en el puerto 3306 desde el grupo de seguridad de la instancia EC2 que contiene el backend y desde el grupo de seguridad de la Lambda que accede a la base de datos.

4.3.4. Procesos de registro de usuarios en la aplicación

La aplicación contempla dos mecanismos principales para dar de alta nuevos usuarios. A continuación se detallan ambos métodos, haciendo énfasis en su funcionamiento y en las situaciones en que se recomienda cada uno.

Auto-registro de pacientes a través de la UI de Cognito

El auto-registro permite que el paciente cree su propia cuenta mediante la interfaz web alojada de Amazon Cognito. Este proceso resulta adecuado para la mayoría de los pacientes, dado que es completamente autogestionado y no requiere

intervención del personal de la clínica, lo que reduce la carga administrativa. La figura 4.9 muestra el diagrama de flujo del auto-registro de pacientes.

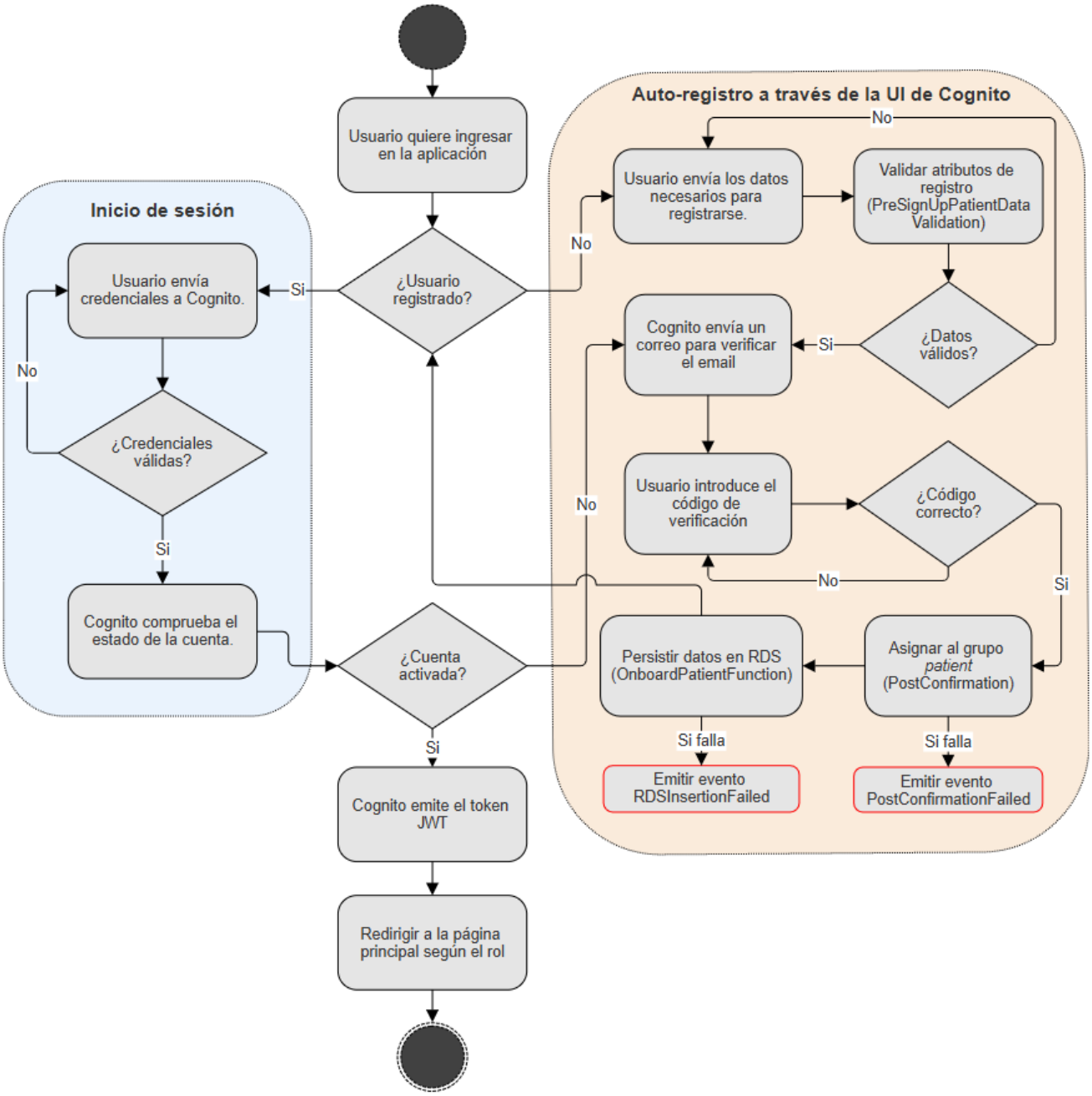


Figura 4.9: Diagrama de flujo del auto-registro de pacientes.

Cuando un paciente no dispone de cuenta y quiere registrarse, primero rellena el formulario integrado en la *Hosted UI* de Cognito. Al enviarlo, realiza validaciones básicas comprobando el formato de email válido, requisitos mínimos de contraseña según la política del *User Pool*, etc.

Para superar las limitaciones de la *Hosted UI* (que solo admite campos en texto plano y carece de desplegables y validaciones avanzadas) se ha implementado la función Lambda `PreSignUpPatientDataValidation` y se ha asociado al disparador `PreSignUp_SignUp` de Cognito. Esta función añade las mismas validaciones a los atributos que en el backend de la aplicación.

A continuación, Cognito registra provisionalmente al usuario y envía un correo de verificación con un código de confirmación de varios dígitos. Hasta que el paciente introduzca correctamente dicho código, la cuenta permanecerá en el estado "no verificada". Si el código es incorrecto o ha expirado, Cognito no confirmará la cuenta y el usuario deberá volver a intentarlo o solicitar reenviar el correo de verificación. Cuando el código sea correcto, Cognito marca el email como verificado y confirma al nuevo usuario.

Una vez confirmada la cuenta, se activa el disparador `PostConfirmation` de Cognito, desencadenando la siguiente cadena de acciones:

1. **Asignación de grupo (`PostConfirmation`)**: La función recibe el nombre de usuario, el ID del *User Pool* y los atributos del usuario, y ejecuta un comando para asignar al usuario recién confirmado al grupo *patient*. Si esta operación falla, publica un evento `PostConfirmationFailed` en EventBridge y finaliza la ejecución sin proceder con los pasos siguientes.

2. **Persistencia en RDS (`OnboardPatientFunction`)**: Si la asignación al grupo se realiza con éxito, la función `PostConfirmation` lanza de forma asíncrona la Lambda `OnboardPatientFunction`, que se ejecuta en una subred privada para acceder a la RDS. Se ha utilizado la biblioteca `mysql2/promise` para crear un *pool* de conexiones, de modo que se mantenga un conjunto de conexiones activas y se reutilicen bajo demanda, evitando así el coste de establecer nuevas conexiones. Primero inserta un registro en la tabla `user` con atributos de Cognito (cognito_id, nombre, email, género, etc.) y posteriormente añade los datos en la tabla paciente. Si ocurre un error, emite el evento `RDSInsertionFailed` en EventBridge.

3. **Compensación de errores (`CompensateUserFunction`)**: Los eventos de error `PostConfirmationFailed` y `RDSInsertionFailed` publicados en EventBridge desencadenan la función Lambda `CompensateUserFunction`. Esta función elimina al usuario recién creado del *User Pool* de Cognito. De este modo, se evita que queden registros sin grupo de Cognito o sin perfil en la base de datos, manteniendo sincronizadas Cognito y la RDS.

Con todas las fases anteriores completadas con éxito, el paciente puede iniciar sesión con sus credenciales. Si la cuenta está verificada, Cognito emitirá los tokens JWT de autenticación y la aplicación redirigirá al usuario a la pantalla principal del paciente.

Registro de usuarios a través de un miembro de la entidad

El segundo método de alta es el registro asistido por personal de la entidad, dirigido a pacientes con poca familiaridad tecnológica o que prefieren el registro asistido, incorporación de nuevo personal en la clínica o casos de soporte en los que algún paciente puede experimentar problemas con el método de auto-registro. La figura 4.10 muestra el diagrama de flujo de registro asistido por entidad.

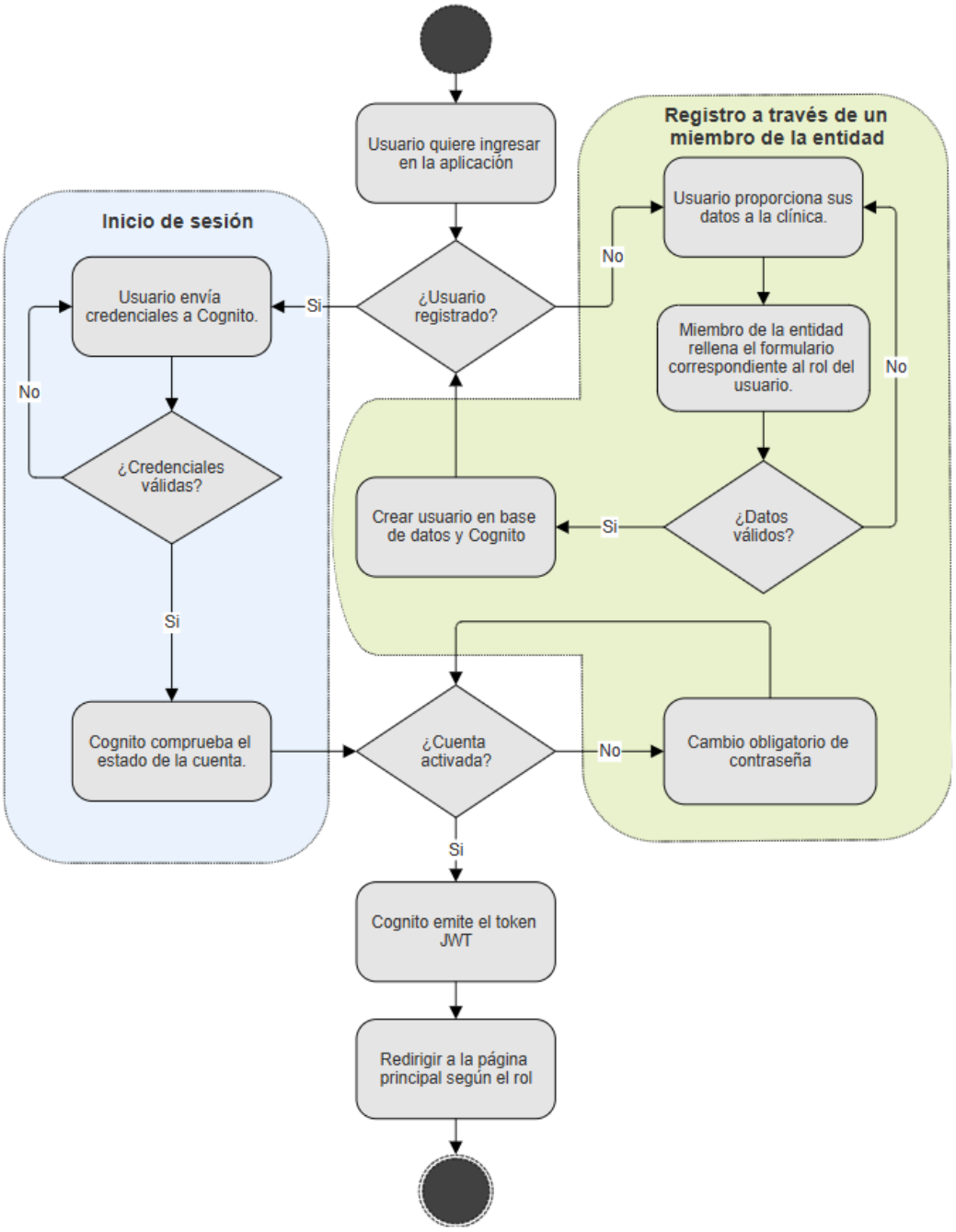


Figura 4.10: Diagrama de flujo de registro asistido por entidad.

Cuando el usuario indica que no dispone de cuenta y desea registrarse de forma asistida, el personal de la clínica recopila los datos necesarios y los introduce en el formulario de alta del rol correspondiente. Se debe tener en cuenta que los auxiliares y nutricionistas solo pueden dar de alta pacientes, mientras que los administradores también pueden registrar nutricionistas y auxiliares.

Al enviar el formulario, el sistema valida los datos tanto en la interfaz como en el backend. Si alguna comprobación falla, se notifica al miembro de la entidad para que corrija la información. Una vez superadas todas las validaciones, se utiliza la operación `AdminCreateUser` de Cognito para crear al usuario en la *User Pool* utilizando el email como nombre de usuario. De este modo, se asigna una contraseña temporal (`Contraseña123!`) y se marca el email del usuario como verificado. El personal de la clínica es quien facilita al nuevo usuario sus credenciales iniciales de manera segura (por teléfono, en persona o mediante un correo manual personalizado). Después de la creación, la cuenta queda en estado de "requiere cambio de contraseña", indicando que el usuario deberá establecer una nueva contraseña en su primer inicio de sesión.

Tras crear el usuario, el sistema agrega esa cuenta al grupo de Cognito que corresponda según el rol seleccionado, mediante la llamada `AdminAddUserToGroup`. Inmediatamente después, el backend inserta al nuevo usuario en la base de datos RDS invocando al repositorio correspondiente. Durante el primer acceso con la contraseña temporal, Cognito detecta que la cuenta requiere cambio de contraseña y no dará acceso directo al usuario, sino que le mostrará una interfaz donde debe introducir la nueva contraseña (Figura 4.11). Cognito establece la contraseña como definitiva y marca la cuenta como activada, dando lugar a que se emita el token JWT de sesión y se redirija al usuario a la pantalla principal correspondiente a su rol.

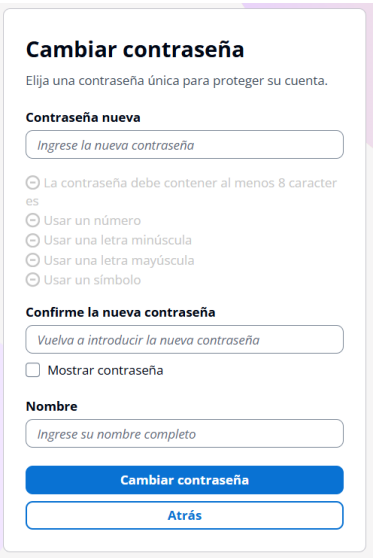


Figura 4.11: Pantalla cambio obligatorio contraseña (*Hosted UI* de Cognito)

4.3.5. Base de datos

Estructura de tablas y relaciones

A continuación se presenta la descripción de las tablas y sus interrelaciones, según el diagrama entidad-relación de la figura 4.3, del apartado de arquitectura y análisis de la base de datos.

- **user**: Almacena los datos comunes a todos los usuarios, independientemente de su rol.
  - **id_user** (`PK`, `BIGINT`): Identificador único para cada usuario.
  - **cognito_id** (`VARCHAR`, único): identificador de AWS Cognito.
  - **name**, **surname** (`VARCHAR`); **birth_date** (`DATE`); **phone** (`VARCHAR`), **mail** (`VARCHAR`, único, inmutable); **gender** (`ENUM`): Datos sobre el usuario.
  - **user_type**: Tipo de usuario o rol. Sus posibles valores son `NUTRITIONIST`, `AUXILIARY`, `ADMIN` y `PATIENT`.

  La tabla `user` se vincula uno a uno con las tablas de los distintos tipos de usuario mediante `id_user` como clave foránea única.

- **nutritionist**: Contiene los datos específicos de los nutricionistas.
  - **id_user** (`PK`, `BIGINT`, `FK`): Es la clave primaria de la tabla y clave foránea que referencia al usuario en la tabla `user`.
  - **start_time**, **end_time** (`TIME`): Hora de inicio y fin de la jornada laboral del nutricionista.
  - **appointment_duration** (`INT`): Duración de las citas en minutos.
  - **max_active_appointments** (`INT`):Número máximo de citas pendientes que puede tener un paciente con el mismo nutricionista.
  - **active** (`BOOLEAN`): Indicador del estado de un nutricionista en el sistema (activo o inactivo).

  Un nutricionista puede tener múltiples citas en `appointment` mediante `id_nutritionist`, estableciendo así una relación uno a muchos.

- **auxiliary** y **admin_auxiliary**: Representan a auxiliares y auxiliares administradores, respectivamente.
  - `id_user` (`PK`, `BIGINT`, `FK`): Es la clave primaria de la tabla y clave foránea que referencia al usuario con una relación uno-a-uno en la tabla `user`.

- **patient**: Contiene los datos específicos de los pacientes.
  - **id_user** (`PK`, `BIGINT`, `FK`): Identificador del paciente, utilizado como clave primaria y como clave foránea que referencia al usuario en la tabla `user`.
  - **active** (`BOOLEAN`): Indicador del estado de un paciente en el sistema (activo o inactivo).

  Un paciente puede tener varias citas en `appointment` mediante `id_patient`, que puede ser nulo para bloqueos de horario.

- **appointment**: Registra las citas agendadas entre pacientes y nutricionistas, así como bloques de tiempo no disponibles. Cada registro representa una cita concreta o un bloqueo de horario.
  - **id_appointment** (`PK`, `UUID/VARCHAR`): Identificador único.
  - **date** (DATE), **start_time** (TIME), **end_time** (`TIME`): Información sobre la fecha, hora de inicio y de finalización de la cita.
  - **type** (`ENUM`): Define el tipo de evento (`APPOINTMENT` o `BLOCKOUT`).
  - **id_nutritionist** (`BIGINT`, `FK`): Referencia al nutricionista que atiende la cita, apuntando a su id de usuario a través de la tabla `nutritionist`.
  - **id_patient** (`BIGINT`, `FK`, Permite valores nulos): Referencia al paciente asociado a la cita, apuntando a su id de usuario mediante la tabla `patient`. Puede ser nulo cuando `type` indica un bloqueo de agenda.

  Las relaciones de `appointment` con `nutritionist` y `patient` son de muchos a uno. Un nutricionista puede tener muchas citas programadas y un paciente puede tener múltiples citas agendadas.

Aspectos relevantes de implementación

- **Generación de claves primarias**: En la tabla `user`, la clave `id_user` se genera con estrategia autoincremental (`GenerationType.IDENTITY`). Sin embargo, en `appointment` se utiliza `GenerationType.UUID` para generar un identificador único global (`UUID`) para `id_appointment`. Inicialmente `id_user` era de tipo `UUID`, pero se modificó a `BIGINT` para utilizar una estrategia autoincremental que facilitase la inserción de nuevos pacientes en la RDS durante el flujo de auto-registro de pacientes con AWS Lambda y EventBridgede. De esta forma se evita tener asignar manualmente el identificador.

- **Modelo de datos y herencia**: Las entidades de rol (`Nutritionist`, `Patient`, `Auxiliary`, `AdminAuxiliary`) no extienden de `User`, sino que

cada una se relaciona uno a uno con `User` utilizando la misma clave primaria en ambas tablas. Para ello se han utilizado las anotaciones JPA `@OneToOne` junto con `@MapsId`, que indica que la entidad de rol reutiliza `id_user`. En `User` se definen referencias inversas con `mappedBy` y `cascade=CascadeType.ALL` para propagar operaciones de crear, actualizar y eliminar al registro relacionado.

- **Uso de Lombok y anotaciones auxiliares**: Se hace uso de las anotaciones `@Getter`, `@Setter`, `@NoArgsConstructor` y `@AllArgsConstructor` para eliminar código repetitivo de getters, setters y constructores básicos. Todas las entidades están anotadas con `@JsonIdentityInfo` indicando `property = idUser`. Esta configuración para serialización JSON utiliza el id del usuario como identificador de referencia para evitar problemas de referencias cíclicas al convertir las relaciones bidireccionales a formato JSON.

4.4. Pruebas y análisis estático

En este apartado se describen las pruebas realizadas y el análisis estático aplicado durante el desarrollo de la aplicación. A continuación, se detallan las herramientas empleadas, su funcionalidad y el enfoque adoptado para las diferentes pruebas.

4.4.1. Análisis estático

El análisis estático consiste en examinar de forma automática el código fuente o compilado sin ejecutar el programa. Su objetivo es identificar de forma anticipada errores, vulnerabilidades, incumplimientos de estilo y malas prácticas de programación. Esto permite mejorar la calidad del código y reducir fallos en etapas posteriores del desarrollo. En el proyecto, el análisis estático se utiliza tanto para el frontend como para el backend, garantizando el cumplimiento de estándares y la coherencia en las buenas prácticas.

Frontend

Para el frontend de la aplicación, desarrollada en JavaScript y CSS, se han utilizado las siguientes herramientas de análisis estático que ayudan a mantener la calidad del código:

- **ESLint**: Se emplea para detectar errores y malas prácticas en el código JavaScript. En el archivo de configuración `.eslintrc.json`, se han habilitado las reglas recomendadas de ECMAScript, que proporcionan una base

común para las implementaciones de JavaScript. También se ha activado el plugin de React y se ha integrado Prettier como formateador de reglas, para que se ejecute dentro del proceso de ESLint, centralizando así todas las comprobaciones. Se ejecuta en el proyecto mediante: `npm run lint`.

- **Stylelint**: Analiza los archivos CSS del proyecto y comprueba que cumplan un conjunto de reglas para mantener la consistencia y las buenas prácticas en las hojas de estilo. En el archivo de configuración `.stylelintrc.json`, se ha definido que se utilicen todas las reglas del paquete `stylelint-config-standard`, donde se abordan convenciones de sintaxis, orden de propiedades, valores permitidos, etc. En el proyecto se ejecuta mediante: `npm run stylelint`.
- **Prettier**: Garantiza un formateo consistente del código. Puede comprobar si el formato del código es correcto según las convenciones definidas mediante el uso de `npm run format:check` o reescribir cada archivo que encuentra ajustándolo al estilo configurado utilizando `npm run format`.

Toda esta lógica se centraliza en los scripts del `package.json`, donde se han definido los alias de los comandos mencionados para ejecutar las herramientas de forma sencilla, evitando así los comandos largos. Además, en el workflow de CI de GitHub Actions se incluyen pasos que ejecutan estas herramientas, garantizando la calidad del código.

Backend

En el backend también se ha empleado análisis estático con herramientas especializadas para código Java. Estas se integran en el proceso de desarrollo y en el workflow de GitHub Actions para reforzar la calidad del código del lado del servidor.

- **Checkstyle**: Mediante el plugin `maven-checkstyle-plugin`, se carga un fichero de configuración (`checkstyle.xml`), que aplica normas de estilo al código Java. Controla la longitud máxima de cada línea, detecta imports no usados y verifica las convenciones de nomenclatura para clases, métodos y variables, entre otras reglas. Durante el desarrollo, se ha ejecutado periódicamente `mvn checkstyle:check` para detectar y corregir los avisos generados, manteniendo así la consistencia en el estilo del código.
- **PMD**: Se ha integrado el plugin `maven-pmd-plugin` con el conjunto de reglas por defecto (`maven-pmd-plugin-default.xml`), orientadas a detectar *code smells* y malas prácticas en el código Java. Esto incluye la identificación de código duplicado, alta complejidad ciclomática, bloques `catch` e `if` vacíos, así como el uso ineficiente de bucles, entre otros problemas que

pueden señalar un diseño poco robusto o propenso a fallos. Durante el desarrollo, se ha ejecutado `mvn pmd:check` para comprobar si la construcción del proyecto fallaba por las infracciones y se ha ejecutado `mvn pmd:pmd` para generar informes que alertan sobre estas situaciones, permitiendo refactorizar el código antes de que los problemas se manifiesten en tiempo de ejecución.

- **SpotBugs**: Mediante el plugin `spotbugs-maven-plugin`, se analiza el *bytecode* compilado para detectar errores de ejecución potenciales que pasan desapercibidos en tiempo de compilación. Entre las comprobaciones más relevantes se incluyen la detección de posibles dereferencias nulas, flujos de control donde un bloque `finally` anula excepciones, condiciones de carrera en código concurrente y excepciones capturadas pero no gestionadas. Para evitar falsos positivos, se han utilizado filtros de exclusión y anotaciones `@SuppressFBWarnings` en casos concretos. Durante el desarrollo se ha ejecutado periódicamente `mvn spotbugs:check`, que detiene el proceso de construcción del proyecto si SpotBugs detecta errores de severidad media o alta e indica las incidencias encontradas.

Además, en el flujo de integración continua se agrupan las comprobaciones de estilo y calidad, ejecutando todos los análisis estáticos del backend (Checkstyle, PMD y SpotBugs) mediante el comando `mvn verify -DskipTests=true`.

4.4.2. Pruebas automáticas

En este apartado se describen las pruebas automáticas implementadas en el proyecto, clasificadas en pruebas unitarias y de integración. Las pruebas unitarias validan la lógica de negocio aislando dependencias, mientras que las pruebas de integración evalúan la correcta interacción entre las distintas capas de la aplicación.

Pruebas unitarias

El backend dispone de pruebas unitarias desarrolladas con JUnit 5 y Mockito. Se han implementado tests para cada servicio de la aplicación, aislando sus dependencias mediante dobles de prueba. Los repositorios y clientes externos se simulan para centrar la prueba en la correcta ejecución de las reglas de cada método.

Se ha empleado un método anotado con `@BeforeEach` para crear objetos de prueba con valores concretos antes de cada test. Las aserciones de JUnit confirman que los resultados satisfacen las condiciones establecidas, ya sea devolver un valor

concreto, lanzar una excepción o devolver una lista con el número esperado de elementos.

Se ha seguido una nomenclatura basada en combinar el nombre del método probado, la condición de ejecución y el resultado esperado, separados por guiones bajos, con el objetivo de que los tests sean descriptivos y fáciles de identificar. Se han elaborado un total de 71 pruebas unitarias. La figura 4.12 muestra algunas pruebas unitarias superadas con éxito.

Figura 4.12: Resultados de los tests unitarios de la clase `NutritionistService`.

Pruebas de integración

Para evaluar la interacción entre capas, se han creado pruebas de integración con Spring Boot Test y MockMvc. Todas las clases de prueba están anotadas con `@SpringBootTest(webEnvironment = RANDOM_PORT)` y `@AutoConfigureMockMvc`, lo que inicia el contexto completo de Spring en un puerto aleatorio y expone un bean de MockMvc para enviar peticiones HTTP simuladas. Mediante `@ActiveProfiles("test")` se activa el perfil de pruebas, que carga una configuración específica que utiliza una base de datos H2 en memoria definida en el archivo `application-test.yml`.

Antes de cada prueba, con `@BeforeEach` se eliminan las tablas clave y se insertan las entidades mínimas necesarias mediante los repositorios JPA, asegurando un estado limpio y escenarios reproducibles. En todos los tests se utiliza la clase `SimpleGrantedAuthority` de Spring Security para simular los distintos niveles de acceso y representar permisos individuales asignados a un usuario.

Las pruebas abarcan los cuatro métodos HTTP principales:

- **POST**: Crea un recurso a partir de un DTO enviado como JSON, comprueba el código de respuesta, el identificador generado y que se han invocado los métodos esperados de las dependencias. Se muestra un ejemplo en el código 4.4.1.

```
mockMvc.perform(post("/patients")
        .with(jwt().authorities(new
            SimpleGrantedAuthority("ROLE_ADMIN")))
        .contentType(MediaType.APPLICATION_JSON)
        .content(objectMapper.writeValueAsString(dto))
    )
    .andExpect(status().isOk())
    .andExpect(jsonPath("$.user.idUser").isNumber())
    .andExpect(jsonPath("$.user.mail").value("nuevo@ejemplo.com"));

verify(cognitoService,times(1)).createCognitoUser(any(UserDTO.class));
```

Código 4.4.1: Fragmento del test de creación de paciente.

- **GET**: Recupera datos tanto de rutas genéricas como de rutas con parámetros (por ejemplo, `/appointments/id` o `/appointments?patientId=123`), validando la estructura y los valores del JSON con `jsonPath`.
- **PUT**: Actualiza un recurso, comprueba el éxito de la operación y verifica que la entidad refleja los cambios en la base de datos.
- **DELETE**: Borra un recurso, comprueba el éxito de la operación y verifica que ya no exista en el repositorio.

Además, la batería de pruebas incluye escenarios de error y autorización: se validan peticiones sin credenciales (respuesta `Unauthorized`), intentos con rol insuficiente (`Forbidden`) y solicitudes con datos inválidos (`Bad Request`). A nivel de lógica de negocio, también se comprueba el lanzamiento de excepciones esperadas y, mediante el uso de verificaciones, se asegura si ciertos métodos de repositorios o servicios externos se invocan o no y cuántas veces, según cada caso.

Se han elaborado un total de 118 pruebas de integración. En la figura 4.13 se pueden observar algunas de ellas pasando correctamente.

Figura 4.13: Resultados de los tests de integración de `NutritionistController`.

4.4.3. Cobertura

La cobertura de pruebas es una métrica esencial que cuantifica el porcentaje de código fuente ejecutado al menos una vez durante la ejecución de las pruebas. Esta métrica proporciona una visión global del alcance con el que se ha evaluado la lógica de la aplicación.

En este proyecto, las pruebas unitarias y de integración alcanzan una cobertura del 82 % de instrucciones (3.248 de 3.953) y del 72 % de ramas (113 de 155) (Figura 4.14). Estos valores reflejan el grado de exposición de la mayor parte de la lógica de negocio a escenarios de prueba, garantizando que la mayoría de los flujos de ejecución hayan sido validados.

Para generar estos informes, se ha utilizado JaCoCo, una herramienta que analiza el código durante la ejecución de las pruebas y muestra las líneas, ramas y métodos que han quedado cubiertos. Gracias a estos informes, es posible identificar áreas con menor cobertura, con la finalidad de centrar en ellas futuros esfuerzos de testing.

Element	Missed Instructions	Cov.	Missed Branches	Cov.
com.jorgeleal.clinicanutricion.model		100 %		n/a
com.jorgeleal.clinicanutricion.config		92 %		25 %
com.jorgeleal.clinicanutricion.controller		86 %		71 %
com.jorgeleal.clinicanutricion.service		79 %		89 %
com.jorgeleal.clinicanutricion		33 %		n/a
com.jorgeleal.clinicanutricion.dto		28 %		0 %
Total	705 of 3.953	82 %	42 of 155	72 %

Figura 4.14: Cobertura de los tests

4.5. Distribución y despliegue

4.5.1. Despliegue con AWS Elastic Beanstalk

Para el despliegue del backend, se ha utilizado Elastic Beanstalk, aprovechando que soporta directamente plataformas como Java, Python y Node.js sin requerir el uso de contenedores personalizados. De este modo, se ha optado por prescindir de Docker, ya que Elastic Beanstalk proporciona el entorno de ejecución necesario y gestiona automáticamente las dependencias de la aplicación, lo que reduce la complejidad operativa y permite centrar los esfuerzos en el desarrollo.

Se ha configurado Elastic Beanstalk para que utilice una única instancia EC2 tipo `t3.micro`, una opción de baja capacidad y coste reducido adecuada para cargas ligeras. Además, se ha habilitado el uso de *Spot Instances* para disminuir el coste de computación mediante el aprovechamiento de capacidad sobrante de AWS. Esta configuración se ha diseñado para mantenerse dentro de la capa gratuita.

Se han definido variables de entorno dentro del apartado de configuración de Elastic Beanstalk para parametrizar el backend, de modo que la aplicación pueda leer dichos valores en tiempo de ejecución sin exponer datos sensibles en el código fuente.

4.5.2. Alojamiento estático con Amazon S3 y CloudFront

El frontend, al ser una aplicación de contenido estático, se ha aprovechado la combinación de Amazon S3 y CloudFront para su alojamiento y distribución. El *bucket* S3 se ha configurado de forma que bloquee el acceso público y permita únicamente el acceso a CloudFront, de forma que ningún otro origen pueda acceder a los archivos.

En la configuración de CloudFront, se ha creado un origen apuntando al *bucket* S3 del frontend. El comportamiento principal de la distribución está definido para servir todo el contenido estático desde este origen, aprovechando la caché de CloudFront para mejorar el rendimiento y reducir la latencia.

Con esta configuración, los recursos se entregan de forma segura y eficiente a los usuarios globales, ya que CloudFront almacena en caché los archivos en ubicaciones cercanas al usuario final (*Edge Locations*).

4.5.3. Flujo CI/CD con GitHub Actions

Se ha implementado un flujo de Integración Continua y Despliegue Continuo (CI/CD) utilizando GitHub Actions para automatizar las pruebas y el despliegue tanto del backend como del frontend. La metodología seguida comprende varias fases:

1. **Pruebas automáticas en rama de desarrollo**: Cada vez que se realiza un `push` a la rama `develop`, GitHub Actions desencadena un workflow de CI que ejecuta automáticamente la batería de pruebas y el análisis estático de código. De esta forma, cualquier error introducido en el código se detecta de forma inmediata en la rama de desarrollo, evitando que avance hacia producción.

2. **Validación en *pull request* a main**: Siguiendo una estrategia GitFlow simplificada, cuando la rama `develop` incorpora nuevas funcionalidades estables tras haber superado las pruebas, se abre una *pull request* (PR) hacia la rama `main`. El workflow de CI se ejecuta con los cambios fusionados virtualmente con el contenido actual de `main`, para asegurar que la combinación de cambios entre develop y main no introduzca regresiones ni errores nuevos. Solo si todas las pruebas finalizan con éxito se autoriza la fusión.

3. **Despliegue automático tras *merge* en main**: Una vez aprobada y fusionada la PR en la rama `main`, se ejecuta un segundo workflow de CD que solamente incluye las etapas de despliegue. En primer lugar, se despliega el backend y si esta operación es exitosa, se inicia el despliegue del frontend. Este enfoque mantiene la coherencia entre backend y frontend sin intervención manual.

4.5.4. Configuración de los workflows CI/CD

El proceso descrito anteriormente se ha codificado en dos workflows de GitHub Actions diferenciados: uno dedicado a la Integración Continua (CI) y otro al Despliegue Continuo (CD). Se han utilizado GitHub Secrets para almacenar la

información sensible que no debe aparecer en el código ni en los logs. A continuación, se detalla la composición de cada workflow, destacando cómo se emplean las herramientas involucradas y la función de cada sección.

Workflow de Integración Continua (CI)

Este workflow, definido en `ci.yml`, se encarga de ejecutar las comprobaciones automáticas cada vez que se efectúa un `push` a la rama `develop` o se abre un *pull request* hacia `main`. Incluye análisis estático de código y pruebas tanto en el backend como en el frontend.

```yaml
on:
  push:
    branches: [develop]
  pull_request:
    branches: [main]
```

Código 4.5.1: Eventos que disparan el workflow de CI (`ci.yml`).

El *job* `test-backend` (Figura 4.15) utiliza un runner en Ubuntu que clona el repositorio e instala Java 17. A continuación, con Maven se realiza el análisis estático con Checkstyle, PMD y SpotBugs, seguido de los tests unitarios y de integración. Finalmente, se suben los informes de prueba como artefactos, para que estén disponibles incluso si algún test falla.

El *job* `test-frontend` (Figura 4.16) depende de que los tests del backend se hayan superado con éxito. Tras configurar el entorno de Node.js e instalar las dependencias, se ejecuta ESLint para JavaScript, Stylelint para CSS y Prettier en modo *check*, asegurando que el formato del código cumple con el estándar definido. De esta forma, ningún cambio de frontend puede avanzar al despliegue si introduce errores de estilo o formato.

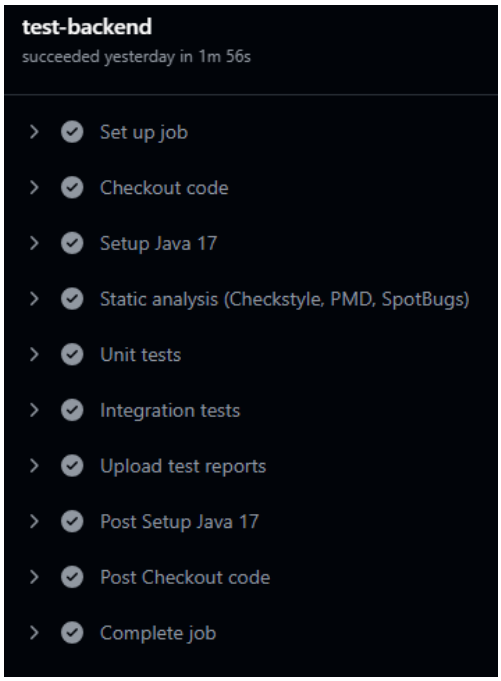


Figura 4.15: Ejecución exitosa del job `test-backend` en GitHub Actions.

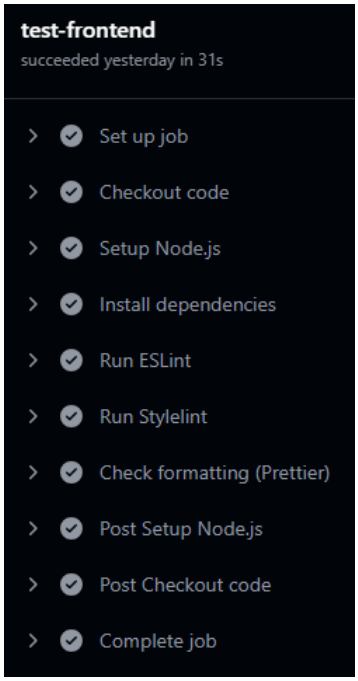


Figura 4.16: Ejecución exitosa del job `test-frontend` en GitHub Actions.

Workflow de Despliegue Continuo (CD)

El workflow de CD, definido en `deploy.yml`, se activa únicamente con un `push` a la rama `main` (Código 4.5.2) y se encarga de publicar la versión estable en producción sin reejecutar pruebas.

```yaml
on:
  push:
    branches: [main]
```

Código 4.5.2: Evento que dispara el workflow de CD (`deploy.yml`).

El *job* `deploy-backend` (Figura 4.17) clona el repositorio y configura las credenciales de AWS desde GitHub Secrets, de modo que las llamadas posteriores a la CLI de AWS estén autenticadas. Para gestionar Elastic Beanstalk, se instala la EB CLI y como muestra el código 4.5.3, se genera dinámicamente la etiqueta de versión basada en el número de ejecución y el SHA del commit. A continuación, con `aws s3 cp` se sube el JAR al *bucket* de S3 configurado en AWS y con `aws elasticbeanstalk create-application-version` creamos una nueva versión de la aplicación. Finalmente, `aws elasticbeanstalk update-environment` aplica la nueva versión al entorno de producción.

```yaml
- name: Upload and register new EB version
  working-directory: backend-clinica
  run: |
    VERSION_LABEL="v-${{ github.run_number }}-${{ github.sha }}"
    echo "Usando version-label: $VERSION_LABEL"
    aws s3 cp target/clinicanutricion-0.0.1-SNAPSHOT.jar \
      s3://${{ secrets.EB_S3_BUCKET
      ↪ }}/clinicanutricion-0.0.1-SNAPSHOT.jar
    aws elasticbeanstalk create-application-version \
      --application-name ${{ secrets.EB_APPLICATION_NAME }} \
      --version-label "$VERSION_LABEL" \
      --source-bundle S3Bucket="${{ secrets.EB_S3_BUCKET
      ↪ }}",S3Key="clinicanutricion-0.0.1-SNAPSHOT.jar"

- name: Update EB environment
  working-directory: backend-clinica
  run: |
    aws elasticbeanstalk update-environment \
      --application-name ${{ secrets.EB_APPLICATION_NAME }} \
      --environment-name ${{ secrets.EB_ENVIRONMENT_NAME }} \
      --version-label "v-${{ github.run_number }}"
```

Código 4.5.3: Publicación y despliegue de versión en Elastic Beanstalk.

Se ha incluido un smoke test tras el despliegue (Código 4.5.4), realizando una petición al endpoint de salud como última comprobación para asegurar que la aplicación está en ejecución y responde correctamente.

```yaml
- name: Smoke test
  run: curl -f https://api.clinicanutricion.es/health
```

Código 4.5.4: Smoke Test.

El *job* `deploy-frontend` (Figura 4.18) se ejecuta una vez que se ha completado con éxito el despliegue del backend. Primero se clona el repositorio y se prepara el entorno de Node.js. A continuación, se inyectan las variables de entorno necesarias (las URLs de la API, configuración de Cognito, etc.), de modo que la compilación ya incorpore la configuración de producción.

Una vez construido el proyecto, se configuran las credenciales de AWS mediante GitHub Secrets y, como se muestra en el código 4.5.5, se sube el contenido estático con `aws s3 sync` al *bucket* correspondiente, eliminando los archivos obsoletos. Por último, se lanza una invalidación de CloudFront para que los usuarios

obtengan inmediatamente la nueva versión, evitando que se sirva contenido almacenado en caché.

```
- name: Deploy to S3
  run: aws s3 sync frontend-clinica/build/
   ↪  s3://react-nutrition-clinic-app --delete

- name: Invalidate CloudFront cache
  run: aws cloudfront create-invalidation --distribution-id ${{
   ↪  secrets.CLOUDFRONT_DISTRIBUTION_ID }} --paths "/*"
```

Código 4.5.5: Publicación del *build* en S3 e invalidación de caché en CloudFront.

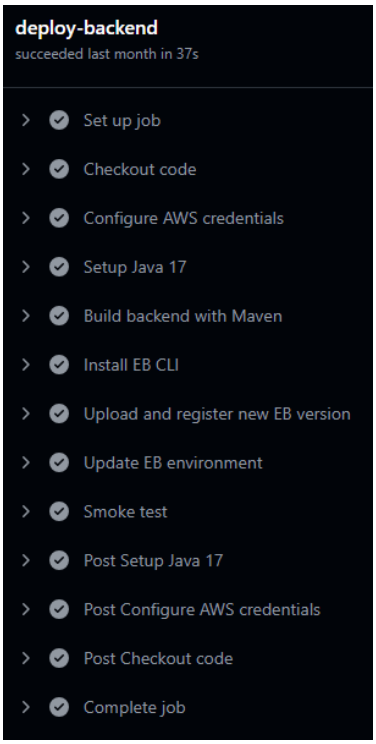


Figura 4.17: Ejecución exitosa del job `deploy-backend` en GitHub Actions.

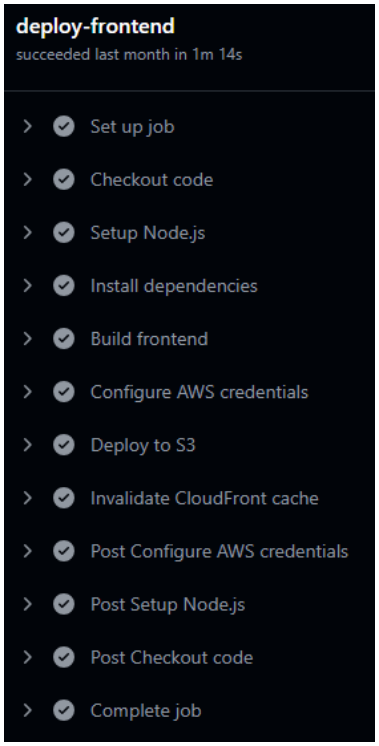


Figura 4.18: Ejecución exitosa del job `deploy-frontend` en GitHub Actions.

5

Conclusiones y trabajos futuros

5.1. Conclusiones del proyecto

En el proyecto se ha cumplido el objetivo principal planteado al inicio, que consistía en diseñar, implementar y desplegar una aplicación web de gestión de citas para una clínica de nutrición. Además, tal como se propuso desde un principio, la aplicación se ha desplegado íntegramente en AWS, aprovechando las ventajas de la computación en la nube.

En cuanto a la implementación de funcionalidades clave, todas han sido desarrolladas de forma satisfactoria, cumpliendo con los requisitos funcionales y técnicos definidos en la planificación del proyecto.

Desde el punto de vista de la infraestructura, el frontend estático se aloja en Amazon S3 y se distribuye con CloudFront, lo que permite reducir la latencia. El backend se ejecuta en AWS Elastic Beanstalk sobre una instancia EC2 y se ha utilizado CloudFront como *proxy* inverso con cifrado TLS. La base de datos RDS se encuentra en subredes privadas, protegida por grupos de seguridad que solo permiten conexiones desde servicios autorizados. El hecho de no haber implementado anteriormente una infraestructura similar ha añadido gran complejidad al trabajo, pero sin duda me ha permitido aprender en profundidad sobre cada una de las herramientas utilizadas y sobre el funcionamiento de AWS.

En términos de seguridad y control de acceso, se ha integrado Amazon Cognito como proveedor de identidad, lo que ha supuesto uno de los mayores retos durante el desarrollo, al haber diferentes formas de llevar a cabo la integración según el

tipo de aplicación y las necesidades de autenticación y autorización. Cognito se ha utilizado para externalizar la gestión de credenciales, así como para gestionar el registro de usuarios y el inicio de sesión, mediante la emisión de tokens JWT. Gracias al uso de Spring Security, se ha aplicado un modelo de autorización basado en roles, asegurando el principio de mínimos privilegios.

Respecto a la calidad del código, se han implementado pruebas unitarias para validar la lógica de negocio y pruebas de integración para evaluar la interacción entre capas. Se ha alcanzado un 82 % de cobertura de instrucciones, lo que aporta un alto grado de confianza en la robustez del sistema. Además, se han incorporado herramientas de análisis estático tanto en el frontend como en el backend, con el objetivo de detectar vulnerabilidades y malas prácticas de forma temprana. Estas tareas se ejecutan automáticamente mediante un workflow de CI/CD con GitHub Actions, que automatiza la compilación, las pruebas y el despliegue.

5.2. Aspectos pendientes y trabajos futuros

A pesar de haber alcanzado los objetivos propuestos, existen funcionalidades adicionales y posibles ampliaciones que podrían abordarse para seguir mejorando la aplicación:

- Implementar un apartado donde los pacientes pueden consultar el historial de sus citas y filtrarlas por fecha y por nutricionista.
- Desarrollar un módulo de estadísticas donde cada nutricionista pueda visualizar el número de pacientes atendidos a lo largo de los meses.
- Permitir añadir una fotografía al perfil de cada usuario.
- Incorporar un sistema de comunicación directa entre el paciente y el nutricionista asignado para la próxima cita, con el fin de resolver posibles dudas relacionadas con la dieta.
- Implementar un sistema donde los nutricionistas puedan guardar y gestionar las dietas, haciendo uso de un *bucket* S3 para el almacenamiento de archivos y una nueva tabla en la base de datos RDS de la aplicación, para guardar los datos de cada dieta y el enlace a la ubicación de cada fichero almacenado en S3.
- Implementar autenticación multifactor (MFA) utilizando Cognito, para reforzar la seguridad de la aplicación.
- Sustituir el *Hosted UI* de Cognito por el uso del SDK de Amazon Cognito Identity JS, debido a las limitaciones del primero en cuanto a personalización del formulario de registro y validaciones. Esta alternativa permitiría

utilizar formularios propios y gestionar de forma personalizada los flujos de registro, confirmación y autenticación desde el backend.

Estas y otras mejoras podrían implementarse para aumentar la complejidad del proyecto y proporcionar a la aplicación una funcionalidad más completa y profesional.

En cuanto a la arquitectura utilizada, se ha diseñado con el objetivo de mantenerse dentro de los límites de la capa gratuita de AWS y minimizar los costes. No obstante, para un entorno con alta disponibilidad, lo más adecuado sería desplegar varias instancias EC2 en subredes distintas, gestionadas por un balanceador de carga. Esta arquitectura permitiría soportar picos de tráfico, reducir el impacto de fallos de instancias individuales y habilitar el autoescalado.

Además, el uso del balanceador de carga añade un componente de seguridad adicional, ya que puede situarse en una subred pública y aislar las instancias del backend en subredes privadas sin IP pública. De esta forma, el tráfico entrante pasaría primero por el balanceador de carga, evitando el acceso directo al backend.

5.3. Conclusiones personales

La elaboración de este trabajo de fin de grado ha sido una experiencia muy enriquecedora que me ha permitido ampliar más aún mi formación académica. He tenido la oportunidad de consolidar y adquirir nuevos conocimientos en ingeniería del software, aprendiendo cómo funcionan herramientas que no había tenido la ocasión de utilizar, como puede ser React y perfeccionando las que sí había utilizado durante mi formación, como Spring Boot, el uso de workflows mediante GitHub Actions o la implementación de pruebas unitarias e integración.

No obstante, donde verdaderamente he aprendido ha sido trabajando con los servicios de AWS. He adquirido conocimientos prácticos sobre cómo desplegar y gestionar servicios en la nube, adquiriendo competencias en el uso de herramientas como Elastic Beanstalk, S3, RDS, CloudFront, Cognito, Lambda, entre otras. Este proyecto me ha permitido descubrir un área de la informática en la que no había profundizado hasta ahora y en la que quiero seguir especializándome en un futuro, con el objetivo de consolidar mi experiencia en el uso de herramientas y arquitecturas en la nube.

Bibliografía

[1] M. de Sanidad (España), "Número de casos de obesidad registrados en españa de 2011 a 2023 [gráfica]," Statista, jan. 2025, recuperado el 03 de junio de 2025. [Online]. Available: https://es.statista.com/estadisticas/1046199/numero-de-casos-de-obesidad-en-espana/

[2] Oracle, "Java, lenguaje de programación," https://www.oracle.com/java/technologies/introduction-to-java.html, 1995.

[3] M. Contributors, "Javascript — mdn," https://developer.mozilla.org/en-US/docs/Web/JavaScript, 2024.

[4] HTML.com, "¿qué es html?" https://html.com/#What_is_HTML, 2024.

[5] M. Contributors, "Css," https://developer.mozilla.org/es/docs/Web/CSS, 2024.

[6] R. Hat, "What is yaml," https://www.redhat.com/en/topics/automation/what-is-yaml, 2024.

[7] R. Team, "React: Biblioteca de javascript para construir interfaces de usuario," https://es.react.dev/, 2024, recuperado el 03 de junio de 2025.

[8] Digital55, "¿qué son las single page application (spa)?" https://digital55.com/blog/que-son-single-page-application-spa-desarrollo-elegido-por-gmail-linkedin/, 2025, recuperado el 04 de junio de 2025.

[9] 10Code, "¿qué es react router dom, rutas dinámicas y anidadas, navegación y redirección," https://10code.es/en/react-router-dom/, jul. 2024, recuperado el 04 de junio de 2025.

[10] AuthTS, "react-oidc-context," https://github.com/authts/react-oidc-context, 2025, recuperado el 04 de junio de 2025.

[11] Bootstrap, "Documentación de bootstrap," https://getbootstrap.esdocu.com/, 2025, recuperado el 05 de junio de 2025.

[12] UXPin, "Bootstrap vs react bootstrap: ¿cuál es la diferencia?" https://www.uxpin.com/studio/blog/bootstrap-vs-react-bootstrap/#:~:text=Bootstrap%20relies%20on%20jQuery%20for,a%20lighter%2C%20more%20efficient%20application., 2025, recuperado el 05 de junio de 2025.

[13] Lando, "Creating your first react big calendar," https://lando.hashnode.dev/creating-your-first-react-big-calendar, 2025, recuperado el 05 de junio de 2025.

[14] S. Boot, "Spring boot documentation," https://docs.spring.io/spring-boot/index.html, 2025, recuperado el 05 de junio de 2025.

[15] I. Pivotal Software, "Spring framework," https://spring.io/projects/spring-framework, 2025, recuperado el 05 de junio de 2025.

[16] GeeksforGeeks, "Spring mvc framework," https://www.geeksforgeeks.org/spring-mvc-framework/, 2025, recuperado el 05 de junio de 2025.

[17] S. Data, "Spring data jpa," https://spring.io/projects/spring-data-jpa, 2025, recuperado el 05 de junio de 2025.

[18] Spring Security, "Spring security project," https://spring.io/projects/spring-security, 2025, recuperado el 05 de junio de 2025.

[19] ——, "Oauth 2.0 resource server (servlet)," https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/index.html, 2025, recuperado el 05 de junio de 2025.

[20] Amazon Web Services, "Amazon cognito – funcionalidades," https://aws.amazon.com/es/cognito/features/#topic-0, 2025, recuperado el 06 de junio de 2025.

[21] ——, "Amazon simple email service (ses)," https://aws.amazon.com/es/ses/, 2025, recuperado el 05 de junio de 2025.

[22] P. Lombok, "Lombok artifact," https://mvnrepository.com/artifact/org.projectlombok/lombok, 2025, recuperado el 05 de junio de 2025.

[23] D. Contributors, "dotenv: Loads environment variables from .env," https://www.npmjs.com/package/dotenv, 2025, recuperado el 05 de junio de 2025.

[24] A. W. Services, "Amazon rds user guide," https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Welcome.html, 2025, recuperado el 06 de junio de 2025.

[25] A. Java, "H2 database," https://www.arquitecturajava.com/h2-database-java-y-acceso-remoto/, 2025, recuperado el 05 de junio de 2025.

[26] OpenJS Foundation, "Eslint," 2025, recuperado el 5 de junio de 2025. [Online]. Available: https://eslint.org/

[27] Stylelint, "Stylelint: A mighty, modern linter that helps you avoid errors and enforce best practices in your stylesheets," https://stylelint.io/, 2025, recuperado el 05 de junio de 2025.

[28] Prettier, "Prettier: An opinionated code formatter," https://prettier.io/, 2025, recuperado el 05 de junio de 2025.

[29] Checkstyle, "Checkstyle: A development tool to help programmers write java code that adheres to a coding standard," https://checkstyle.sourceforge.io/, 2025, recuperado el 05 de junio de 2025.

[30] PMD, "Pmd: A source code analyzer," https://pmd.github.io/, 2025, recuperado el 05 de junio de 2025.

[31] Baeldung, "Spotbugs: Detect bugs in your code," https://www.baeldung.com/spotbugs-detect-bugs-code, 2025, recuperado el 05 de junio de 2025.

[32] JUnit, "Junit: A programmer-oriented testing framework for java," https://junit.org/, 2025, recuperado el 05 de junio de 2025.

[33] Mockito, "Mockito: A testing framework for java," https://site.mockito.org/, 2025, recuperado el 05 de junio de 2025.

[34] S. Team, "Spring boot testing documentation," https://docs.spring.io/spring-boot/reference/testing/index.html, 2025, recuperado el 05 de junio de 2025.

[35] S. Framework, "Spring framework testing - mockmvc," https://docs.spring.io/spring-framework/reference/testing/mockmvc.html, 2025, recuperado el 05 de junio de 2025.

[36] JaCoCo, "Jacoco: Java code coverage library," https://www.jacoco.org/jacoco/trunk/doc/mission.html, 2025, recuperado el 27 de junio de 2025.

[37] A. W. Services, "Amazon ec2," https://docs.aws.amazon.com/es_es/AWSEC2/latest/UserGuide/concepts.html, 2025, recuperado el 05 de junio de 2025.

[38] ——, "Bienvenido a aws elastic beanstalk," https://docs.aws.amazon.com/es_es/elasticbeanstalk/latest/dg/Welcome.html, 2025, recuperado el 05 de junio de 2025.

[39] ——, "Amazon s3: Simple storage service," https://aws.amazon.com/es/s3/, recuperado el 05 de junio de 2025.

[40] ——, "Guía de desarrollo de amazon cloudfront," https://docs.aws.amazon.com/es_es/AmazonCloudFront/latest/DeveloperGuide/Introduction.html?utm_source=chatgpt.com, 2025, recuperado el 05 de junio de 2025.

[41] ——, "¿qué es amazon vpc? guía del usuario," https://docs.aws.amazon.com/es_es/vpc/latest/userguide/what-is-amazon-vpc.html, 2025, recuperado el 05 de junio de 2025.

[42] ——, "Aws identity and access management (iam) – guía de introducción," https://docs.aws.amazon.com/es_es/IAM/latest/UserGuide/introduction.html, 2025, recuperado el 05 de junio de 2025.

[43] ——, "Guía de inicio de aws lambda," https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html, 2025, recuperado el 05 de junio de 2025.

[44] ——, "Amazon eventbridge documentation," https://docs.aws.amazon.com/eventbridge/, 2025, recuperado el 16 de junio de 2025.

[45] ——, "Guía de desarrollo de amazon route 53," https://docs.aws.amazon.com/es_es/Route53/latest/DeveloperGuide/Welcome.html, 2025, recuperado el 05 de junio de 2025.

[46] ——, "Aws certificate manager," https://aws.amazon.com/es/certificate-manager/, 2025, recuperado el 05 de junio de 2025.

[47] ——, "¿qué es amazon cloudwatch?" https://docs.aws.amazon.com/es_es/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html, 2025, recuperado el 05 de junio de 2025.

[48] ——, "¿qué es aws?" https://aws.amazon.com/es/what-is-aws/, 2025, recuperado el 05 de junio de 2025.

[49] GitHub, "¿qué son git y github?" https://docs.github.com/es/get-started/start-your-journey/about-github-and-git, 2025, recuperado el 05 de junio de 2025.

[50] A. Maven, "What is apache maven?" https://maven.apache.org/what-is-maven.html, 2025, recuperado el 05 de junio de 2025.

[51] O. Corporation, "Mysql workbench," https://www.mysql.com/products/workbench/, 2025, recuperado el 05 de junio de 2025.

[52] Postman, "¿qué es postman?" https://www.postman.com/product/what-is-postman/, 2025, recuperado el 05 de junio de 2025.

[53] GitHub, "Github actions," https://github.com/features/actions, 2025, recuperado el 08 de junio de 2025.

[54] A. W. Services, "Aws command line interface (cli)," https://aws.amazon.com/es/cli/, 2025, recuperado el 08 de junio de 2025.

[55] Atlassian, "Gitflow workflow," https://www.atlassian.com/es/git/tutorials/comparing-workflows/gitflow-workflow, 2025, recuperado el 08 de junio de 2025.

Apéndices

A

Repositorio GitHub

A.1. Enlace al repositorio

El proyecto ha sido alojado en la plataforma GitHub.

El enlace es el siguiente: https://github.com/codeurjc-students/2025-ClinicaNutricion