UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

SafeGuard: aplicación al caso de estudio de un grupo Scout.

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Rodríguez Gómez, Pablo

Tutor/a: Molina Marco, Antonio

CURSO ACADÉMICO: 2023/2024

Agradecimientos

Quiero agradecer a mi tutor Antonio Molina Marco por acompañarme a lo largo del desarrollo de este proyecto y ayudarme a mejorarlo; y a Pablo Benlloch Caballero por toda la información proporcionada, la formación impartida en varios temas del trabajo, y por siempre estar dispuesto a ayudarme.

Resum

El propòsit d'este TFG és la revisió i recopilació d'informació i documentació sobre arquitectures *software*, amb la finalitat de desenvolupar una arquitectura que servisca de base per a altres projectes d'organitzacions xicotetes i mitjanes. Aquesta arquitectura sorgix davant la necessitat de les petites i mitjanes organitzacions que no necessiten o no poden permetre's una aplicació amb massa recursos.

L'arquitectura desenvolupada garantix la seguretat de les dades tractades i l'escalabilitat del sistema davant la demanda, alhora que posseïx un cost gratuït i servix per a crear aplicacions multiplataformes.

Per al desenvolupament de l'arquitectura del sistema s'ha fet ús del *cloud computing* amb els servicis de la plataforma AWS i per a la comunicació entre els mateixos s'ha usat una arquitectura sofware REST. Així mateix, la metodologia seguida per a desenvolupar el projecte ha sigut una planificació per etapes guiada per un tauler Kanban.

Paraules clau: Cloud Computing, REST, Arquitectura, Software, Serverless

---

Resumen

El propósito de este TFG es la revisión y recopilación de información y documentación sobre arquitecturas *software*, con el fin de desarrollar una arquitectura que sirva de base para otros proyectos de organizaciones pequeñas y medianas. Esta arquitectura surge ante la necesidad de las pequeñas y medianas organizaciones que no necesitan o no pueden permitirse una aplicación con demasiados recursos.

La arquitectura desarrollada garantiza la seguridad de los datos tratados y la escalabilidad del sistema ante la demanda, a la vez que posee un costo gratuito y sirve para crear aplicaciones multiplataformas.

Para el desarrollo de la arquitectura del sistema se ha hecho uso del *cloud computing* con los servicios de la plataforma AWS y para la comunicación entre los mismos se ha usado una arquitectura de software REST. Asimismo, la metodología seguida para desarrollar el proyecto ha sido una planificación por etapas guiada por un tablero Kanban.

Palabras clave: Cloud Computing, REST, Arquitectura, Software, Serverless

---

Abstract

The purpose of this Bachelor's Final Project is the review and collection of information and documentation on software architectures, in order to develop an architecture that will serve as a basis for other projects of small and medium organizations. This architecture arises from the need of small and medium organizations that do not need or cannot afford an application with too many resources.

The developed architecture guarantees the security of the processed data and the scalability of the system to the demand, at the same time that it has a free cost and serves to create multiplatform applications.

For the development of the system architecture, cloud computing has been used with the services of the AWS platform and for the communication between them a REST software architecture has been used. Likewise, the methodology followed to develop the project has been a step-by-step planning guided by a Kanban board.

Key words: Cloud Computing, REST, Architecture, Software, Serverless

---

Índice general

Índice general 3
Índice de figuras 5
Índice de tablas 6

1 Introducción 1
1.1 Motivación . . . 1
1.2 Objetivos . . . 1
1.2.1 Objetivos generales . . . 1
1.2.2 Objetivos específicos . . . 2
1.3 Impacto esperado . . . 2
1.4 Metodología . . . 2
1.5 Estructura de la memoria . . . 2
1.6 Convenciones . . . 3
2 Estado del arte 5
2.1 Plataformas cloud . . . 5
2.1.1 AWS . . . 6
2.1.2 Azure . . . 6
2.1.3 GCP . . . 6
2.2 Comparativa plataformas *cloud* . . . 7
2.3 Análisis de las aplicaciones . . . 7
2.3.1 CiViCRM . . . 7
2.3.2 Gillweb . . . 11
2.4 Comparativa entre las aplicaciones . . . 15
2.5 Propuesta . . . 16
3 Análisis de AWS 17
3.1 Información general . . . 17
3.2 Servicios . . . 18
3.2.1 Informática . . . 18
3.2.2 Almacenamiento . . . 18
3.2.3 Base de datos . . . 18
3.2.4 Redes y entrega de contenido . . . 19
3.2.5 Seguridad, identidad y conformidad . . . 19
3.2.6 Móvil . . . 19
4 Análisis del problema 21
4.1 Árbol de utilidad . . . 22
4.2 Análisis de la seguridad . . . 23
4.3 Análisis del marco legal y ético . . . 23
4.4 Análisis de la escalabilidad . . . 24
4.5 Identificación y análisis de soluciones posibles . . . 24
4.5.1 Máquinas virtuales . . . 24
4.5.2 Servicios AWS Free tier . . . 25
4.5.3 Servicios AWS Pago . . . 27

4.5.4 Comparación entre las tres soluciones 28
4.6 Solución propuesta 28
4.7 Plan de Trabajo 29
4.8 Presupuesto 31
5 Diseño de la solución **33**
5.1 Arquitectura del Sistema 33
5.2 Diseño Detallado 33
5.2.1 Comunicación entre servicios 33
5.2.2 Servicios en profundidad 35
5.2.3 Despliegue de los servicios 36
5.3 Tecnología Utilizada 36
5.3.1 Lenguajes de desarrollo y Frameworks 36
5.3.2 Entorno de desarrollo integrado 36
5.3.3 Control de versiones 37
6 Desarrollo e implementación de la arquitectura **39**
6.1 Desarrollo de la arquitectura 39
6.2 Implementación de la aplicación web 40
6.2.1 Componentes 40
6.2.2 Vistas 40
6.2.3 Comportamiento 41
6.2.4 Estilos 41
6.2.5 Navegación 41
6.2.6 Guards 41
6.2.7 Interceptor HTTP 41
6.2.8 Despliegue 41
6.3 Implementación del servidor 41
6.3.1 Lambda Handler 42
6.3.2 Controladores 42
6.3.3 Servicios 42
6.3.4 Modelos 43
6.3.5 Despliegue 43
6.4 Cognito y DynamoDB 44
6.4.1 Cognito 44
6.4.2 DynamoDB 46
7 Pruebas **49**
7.0.1 Pruebas de seguridad 49
7.0.2 Pruebas de integración 52
7.0.3 Pruebas de carga 53
8 Prueba de concepto **57**
8.1 Consulta de información 57
8.2 Modificación de información 60
9 Conclusiones **61**
9.1 Relación del trabajo desarrollado con los estudios cursados 61
9.2 Trabajos futuros 62
10 Glosario **63**
Bibliografía **65**

Apéndice

A Logs Nmap **69**
B ODS **77**

Índice de figuras

2.1 División del mercado global de los proveedores de infraestructuras *cloud*. 6
2.2 Panel de control. Paso 1. *(Perteneciente a la Guía CRM para Grupo Scout)* . . 8
2.3 Panel de control. Paso 2. *(Perteneciente a la Guía CRM para Grupo Scout)* . . 8
2.4 Panel de control. Paso 3. *(Perteneciente a la Guía CRM para Grupo Scout)* . . 9
2.5 Crear y editar contactos de un grupo. *(Perteneciente a la Guía CRM para Grupo Scout)* 9
2.6 Crear y editar Membresías de un grupo. Paso 1. *(Perteneciente a la Guía CRM para Grupo Scout)* 10
2.7 Crear y editar Membresías de un grupo. Paso 2. *(Perteneciente a la Guía CRM para Grupo Scout)* 10
2.8 Vista del índice en Gillweb. 11
2.9 Desplegable de informes en vista de los usuarios en Gillweb. 12
2.10 Vista de usuarios en Gillweb. 12
2.11 Editar usuario en Gillweb. 13
2.12 Pestaña membresía en editar usuario en Gillweb. 13
2.13 Vista tesorería en Gillweb. 14
2.14 Vista albergues en Gillweb. 14
2.15 Vista de las rutas en Gillweb. 15

4.1 Características de calidad definidas según el estándar **ISO/IEC 25010**. . . . 22
4.2 Diagrama de arquitectura de la aplicación con instancias EC2 (Elaboración propia). 24
4.3 Diagrama de arquitectura de la aplicación con servicios del free tier de AWS (Elaboración propia). 25
4.4 Diagrama de arquitectura de la aplicación con servicios de pago de AWS (Elaboración propia). 27

5.1 Arquitectura Safeguard con componentes **REST** integrados. 34

6.1 Vista principal del servicio AWS. 44
6.2 Diálogo de creación de un grupo de usuarios. 45
6.3 Vista del grupo de usuarios Safeguard. 45
6.4 Vista del cliente del grupo de usuarios Safeguard. 46
6.5 Vista principal del servicio DynamoDB. 46
6.6 Vista del dialogo de crear tablas. 47

7.1 Prueba de seguridad: *Authorization* sin cabecera. 49
7.2 Prueba de seguridad: *Authorization* con cabecera y sin *token*. 50
7.3 Prueba de seguridad: *Authorization* con cabecera y con *token*. 50
7.4 Inyección SQL en el login de Safeguard. 51
7.5 Prueba de integración: Acceso a **DynamoDB** a través del *endpoint* y la API. 52
7.6 Prueba de carga con 5000 hilos en 60 segundos. 53
7.7 Prueba de carga con 4000 hilos en 60 segundos. 54
7.8 Prueba de carga con 3000 hilos en 60 segundos. 54

7.9 Prueba de carga con 2000 hilos en 60 segundos. 54
7.10 Prueba de carga con 1000 hilos en 60 segundos. 55
7.11 Prueba de carga con 500 hilos en 60 segundos. 55
7.12 Prueba de carga con 250 hilos en 60 segundos. 55

8.1 Prueba de concepto: Vista de la cuenta. 57
8.2 Prueba de concepto: Vista de los pagos. 60

Índice de tablas

2.1 Comparativa entre aplicaciones. 15

4.1 *Stakeholders* de la arquitectura y sus necesidades. 21
4.2 Árbol de utilidad de la arquitectura Safeguard. 22
4.3 Comparativa entre arquitecturas. 28
4.4 Plan de trabajo inicial en días. 29
4.5 Plan de trabajo inicial en horas. 30
4.6 Plan de trabajo actualizado en horas. 31
4.7 Plan de trabajo ejecutado en horas. 31

7.1 Valores de pruebas de carga. 53

B.1 Relación con los ODS. 77

CAPÍTULO 1
Introducción

En este trabajo se va a exponer qué es el *cloud computing* y cómo puede ser aplicado para desarrollar nuevas aplicaciones. Para ello, se revisarán las plataformas que ofrecen servicios *cloud* más conocidas; qué tipos de escalado existen en las aplicaciones y cómo nos puede ayudar el *cloud* a aplicarlos; y las dos vertientes de arquitecturas de sistemas existentes y cómo se complementan con el cloud. Tras esto se diseñará y desarrollará una arquitectura *software* basándonos en la información recopilada.

En este capítulo se hará hincapié en la motivación que ha llevado a la realización de este trabajo, así como sus objetivos generales y específicos a alcanzar, junto a la metodología de trabajo seguida.

1.1 Motivación

La motivación para realizar este Trabajo Final de Grado (o TFG por sus siglas) proviene de querer continuar con el último peldaño restante de mi formación universitaria.

Para ello, se ha seleccionado un tema que sirva al interés personal de aprender sobre tecnologías actuales, y con relación con algo presente en mi día a día como lo es el escultismo.

En concreto, el tema a tratar es el *cloud computing* y las cuestiones subyacentes al mismo, como lo son la ciberseguridad, las arquitecturas en la nube y en el *software*, además de los diferentes proveedores de servicios. Estas cuestiones poseen gran relevancia actualmente en el mundo tecnológico, por ello pienso que el tema elegido para el trabajo puede suponer un gran beneficio en mi carrera profesional, además de una gran fuente de aprendizaje.

1.2 Objetivos

1.2.1. Objetivos generales

Los desarrolladores y arquitectos de *software* enfrentamos diariamente retos al tener que evaluar las diferentes alternativas existentes, los costes o la elección de una determinada arquitectura. Esto, sumado a que la información suele estar dispersa en la red o mal documentada, suele resultar en un proceso tedioso y plagado de errores.

El objetivo general de este trabajo consiste en realizar una revisión de toda esta información y documentación en la red y ofrecer una solución que pueda servir como base para otros proyectos y desarrolladores, en particular, para organizaciones pequeñas y me-

dianas. Esta solución deberá garantizar la escalabilidad y la seguridad de las aplicaciones desarrolladas a base de la misma.

Para ello, se analizará la arquitectura de las aplicaciones usadas por grupos scouts o juveniles para tratamiento de datos sensibles y se desarrollará una arquitectura en la nube que sirva para cubrir las carencias que puedan tener esas aplicaciones, garantizando también la seguridad y la escalabilidad.

1.2.2. Objetivos específicos

Concretamente, la arquitectura a desarrollar debe cumplir con los siguientes objetivos:

1. Revisar las plataformas de *cloud computing* actuales.
2. Diseñar una arquitectura que garantice:
    2.1 Tener un bajo coste o ser gratuita.
    2.2 Garantizar que el acceso a los datos esté restringido y estén protegidos de manera segura, es decir, que solo se pueda acceder desde dentro de la aplicación.
    2.3 Tener la capacidad de escalar ante aumentos de la demanda.
    2.4 Ser capaz de servir de base para desarrollar una aplicación multiplataforma.
3. Validar la arquitectura propuesta mediante una prueba de concepto en el contexto scout.

1.3 Impacto esperado

El impacto esperado de la arquitectura será establecer la base para desarrollar aplicaciones que garanticen la seguridad de los datos de los usuarios y el acceso a los mismos desde diferentes dispositivos; que puedan responder ante el incremento de la demanda puntual ante ciertos eventos; y que sean asequibles.

1.4 Metodología

Para la metodología de trabajo se han seguido los siguientes procedimientos:

- Planificación de etapas.
- Definición de objetivos y tareas para cada etapa.
- Desarrollo guiado por un tablero Kanban.
- Revisión periódica cada 2 semanas del trabajo realizado con el tutor.

1.5 Estructura de la memoria

En la memoria podemos encontrar diferentes capítulos, cada uno con su propia función.

- **Capítulo 2. Estado del arte:** En este capítulo veremos una introducción a la arquitectura en sistemas y cómo se relaciona con el *cloud computing*. Además, también veremos diferentes aplicaciones disponibles en nuestro ámbito de estudio, un análisis de las mismas y una propuesta a nivel macroscópico para solucionar las carencias encontradas.
- **Capítulo 3. Análisis de AWS:** Este capítulo recopilará la información disponible y las categorías de servicios existentes de AWS, haciendo hincapié en las relacionadas con nuestro trabajo.
- **Capítulo 4. Análisis del problema:** En el capítulo cuatro, se hará hincapié en el análisis de la situación desde el punto de vista de un arquitecto de *software*, y cómo los atributos de calidad nos permiten describir las necesidades de los diferentes roles involucrados en el uso de la solución. Asimismo, analizaremos diversos apartados relacionados con el problema y plantearemos varias soluciones y un análisis de las mismas. Tras esto, se presentará la solución elegida y el plan de trabajo para llevarla a cabo.
- **Capítulo 5. Diseño de la solución:** En el quinto capítulo, entraremos a detalle en el diseño de la solución elegida previamente. Veremos cómo se relacionan y comunican los diferentes componentes de la solución y qué papel juega la arquitectura REST en este proceso. Además, podremos ver cómo funciona cada componente, qué papel desempeña en la arquitectura diseñada y cómo desplegar el resultado final. Por último, veremos también las tecnologías a utilizar en el desarrollo de la solución elegida.
- **Capítulo 6. Desarrollo e implementación de la arquitectura:** A continuación, en el capítulo seis, veremos el proceso de desarrollo de la solución propuesta, qué problemas se han presentado y cómo se han solucionado. Tras esto, se explicarán los diferentes componentes del sistema y la implementación de cada uno haciendo uso del *cloud* y la integración continua.
- **Capítulo 7. Pruebas:** En el séptimo capítulo se describirán las pruebas llevadas a cabo para confirmar el correcto funcionamiento de la solución desarrollada.
- **Capítulo 8. Prueba de concepto:** En el octavo veremos una prueba de concepto de una aplicación web desarrollada utilizando como base la arquitectura propuesta.
- **Capítulo 9. Conclusiones:** Finalmente, nos quedaría ver las conclusiones extraídas del trabajo y cómo puede afectar el mismo a trabajos futuros.

Para poder seguir la lectura del trabajo expuesto en todo momento, se dispone de un glosario al final del trabajo donde se exponen diferentes conceptos que pueden ser de utilidad para el lector. Además, también se podrá encontrar un capítulo de referencias bibliográficas utilizadas a lo largo del trabajo.

1.6 Convenciones

Las citas textuales externas a la obra se encontrarán entrecomilladas si cuentan con menos de 40 palabras; en caso de tratarse de citas mayores de 40 palabras, se citarán en un párrafo aparte con un margen mayor. En cualquiera de ambos casos, se citará a una referencia que se listará en el capítulo **Bibliografía** al final del documento.

CAPÍTULO 2
Estado del arte

Actualmente, existen diversas aplicaciones *software* destinadas a ofrecer servicios de almacenamiento de datos sensibles para organizaciones pequeñas y medianas, como grupos juveniles o grupos scouts.

Las diferencias entre las diversas aplicaciones disponibles suelen radicar en el tipo de arquitectura de sistema empleada y las funcionalidades que ofrecen. Estas últimas suelen variar dependiendo de los requisitos de la organización, pero, con respecto a la arquitectura, se pueden reducir a dos vertientes principales: sistemas **monolíticos** y sistemas **distribuidos**.

Amazon Web Services nos define los sistemas **monolíticos** como "un modelo de desarrollo de *software* tradicional que utiliza un código base para realizar varias funciones empresariales. Todos los componentes de *software* de un sistema monolítico son interdependientes debido a los mecanismos de intercambio de datos dentro del sistema.". [1]

En contraposición, la empresa de *software* **Atlassian**, nos define los sistemas **distribuidos** "como un conjunto de programas informáticos que utilizan recursos computacionales en varios nodos de cálculo distintos para lograr un objetivo compartido común ... estos nodos suelen representar dispositivos de *hardware* físicos diferentes, pero también pueden representar procesos de *software* diferentes...". [2]

Teniendo ambas definiciones en cuenta, podemos destacar que una de las principales diferencias entre ambos abordamientos a la arquitectura del sistema recae en la escalabilidad. Los sistemas monolíticos presentan problemas relacionados con la escalabilidad al no ser lo suficientemente flexibles para responder a los cambios pequeños del mercado. En cambio, los sistemas distribuidos pueden escalar individualmente sus diferentes partes, para responder a las necesidades de los usuarios.

2.1 Plataformas cloud

Actualmente, una de las tecnologías punteras que nos permiten autoescalar nuestro sistema en tiempo real es el ***cloud computing***. Esta tecnología nos permite solventar uno de los requisitos de los sistemas distribuidos, la necesidad de poseer conocimientos sobre redes para conectar los servicios del sistema. Además, existen diversas plataformas *cloud* disponibles actualmente, siendo las principales, Amazon Web Services, Google Cloud o Microsoft Azure, según el gráfico 2.1 desarrollado por **Statista** con base en los datos de **Synergy Research Group**.[1]

[1] https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/

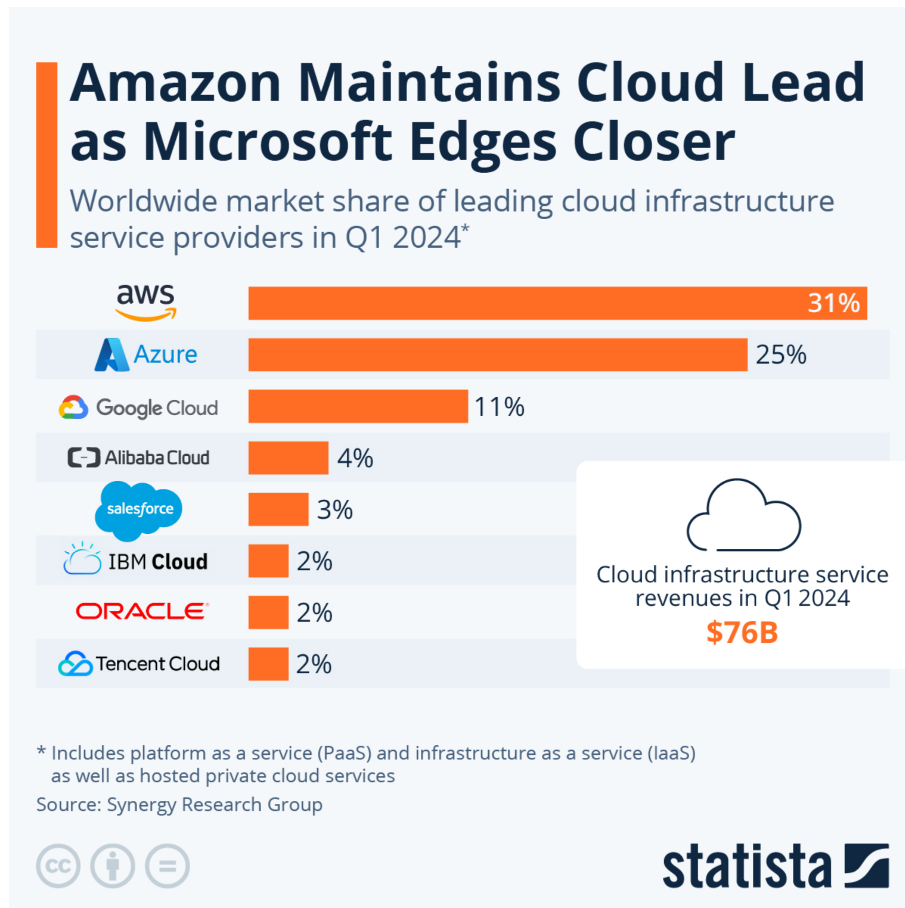

Figura 2.1: División del mercado global de los proveedores de infraestructuras *cloud*.

Las tres plataformas líderes en proveer servicios *cloud* presentan características similares en los ámbitos de servicios, medidas de seguridad, escalabilidad y costes, pero hay algunas diferencias principales entre ellas.

2.1.1. AWS

Amazon Web Services (AWS) posee una amplia red de centros esparcida a lo largo de todo el mundo, facilitando la presencia global y la escalabilidad. Además, posee una gran variedad de servicios disponibles y una base de usuarios madura y extensa.

2.1.2. Azure

Microsoft Azure, en cambio, tiene integración con el resto de servicios de Microsoft y ofrece soluciones *cloud* híbridas. Por otra parte, el foco de la plataforma recae en la seguridad, haciendo un gran énfasis en la misma.

2.1.3. GCP

Google Cloud Platform (GPC) ofrece soluciones centradas en el análisis de datos y el tratamiento de los mismos. También ofrece buen rendimiento asegurando una baja latencia, resultando en bajos tiempos de respuesta.

2.2 Comparativa plataformas *cloud*

Como hemos visto anteriormente, cada plataforma tiene un enfoque principal diferente, aunque el núcleo de todas sea prácticamente el mismo. Esto lo podemos ver gracias a una comparativa[2] hecha por Google entre GPC, AWS y Azure, en la cual podemos ver que la amplia mayoría de servicios ofrecidos por una plataforma encuentra su equivalente en las otras dos.

Ante la necesidad de elegir entre uno de estos tres proveedores, la elección depende de muchos factores. Tal y como destaca Scott P. en el artículo "AWS vs Azure vs GCP: The big 3 cloud providers compared"[3], "no hay una respuesta universal porque todas las situaciones son diferentes. Sugiero realizar algunos proyectos piloto con estos proveedores... Tal vez pruebe a hacer el mismo proyecto en múltiples proveedores y evaluar aspectos como la facilidad de uso, la solidez de las ofertas, la calidad de la documentación, etc".

2.3 Análisis de las aplicaciones

Revisando las aplicaciones usadas actualmente por organizaciones del carácter mencionado anteriormente, podemos encontrar dos principales:

2.3.1. CiViCRM

CiVi Customer Relationship Management[3] es un *software* utilizado a nivel nacional por la Federación de Scouts-Exploradores de España (ASDE) para almacenar los datos de los educandos y monitores adscritos a la federación a nivel nacional. Este *software* es de carácter *open-source*, y se centra en facilitar el manejo de la información (clientes, proveedores, afiliados, socios, etc.) de diversas organizaciones.

Dentro de la página web de ASDE podemos encontrar una guía[4] sobre cómo usar los diferentes apartados del *software*. En ella podemos encontrar explicaciones sobre las siguientes funcionalidades principales:

> **"Panel de control de informes**
>
> Cuando entras por primera vez en el CiviCRM te vas a encontrar con la pantalla de inicio en la cual podrás acceder a todo el menú del programa
>
> Podrás configurar el panel de control para que muestre los informes ("Dashlets") disponibles en el sistema.

[2] https://cloud.google.com/docs/get-started/aws-azure-gcp-service-comparison?hl=es-419

[3] https://civicrm.org

[4] https://www.scout.es/wp-content/uploads/2021/06/Guia-CRM-para-Grupo-Scout.pdf

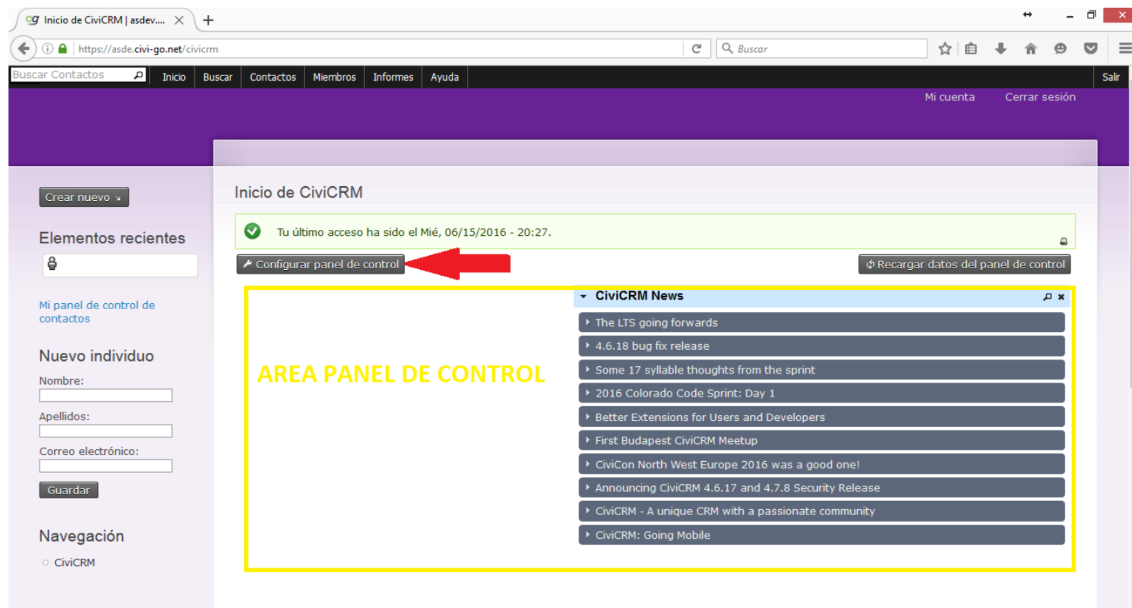

Figura 2.2: Panel de control. Paso 1. *(Perteneciente a la Guía CRM para Grupo Scout)*

Los "Dashlets"podrás colocarlos en las columnas simplemente arrastrándolos a ellas y pulsando el botón “Finalizado”.

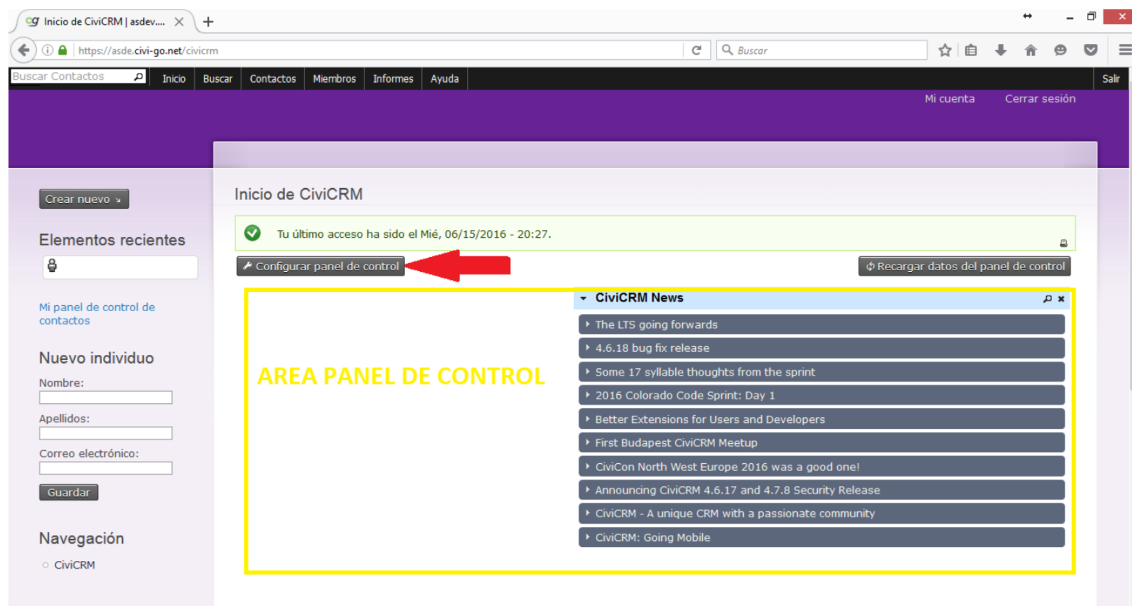

Figura 2.3: Panel de control. Paso 2. *(Perteneciente a la Guía CRM para Grupo Scout)*

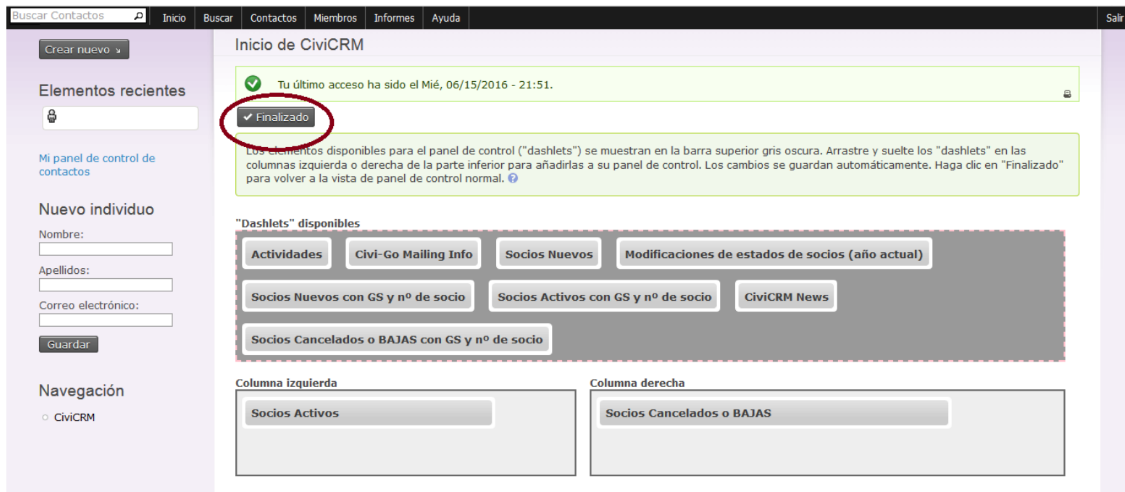

Figura 2.4: Panel de control. Paso 3. *(Perteneciente a la Guía CRM para Grupo Scout)*

Crear y editar contactos de un grupo

Dentro del Civi de ASDE hay tres tipos de contactos:

- Monitores y beneficiarios: socios. Todas las personas miembros de un Grupo Scout: Se consideran “Monitores” a los Scouters y Responsables Adultos y beneficiarios a Castores, Lobatos, Scouts, Escultas/Pioneros y Rovers/Compañeros. Los Responsables Adultos son aquellas personas que no realizan tareas educativas directas pero son socios.
- Individuos. Son aquellas personas que pertenecen al entorno del grupo scout pero no son socios.

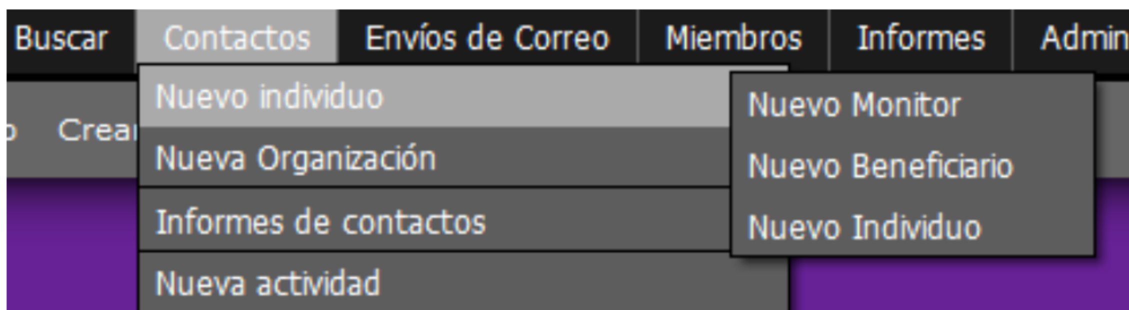

Figura 2.5: Crear y editar contactos de un grupo. *(Perteneciente a la Guía CRM para Grupo Scout)*

Accedes al menú de contactos y escoges uno de los subtipos de individuo disponibles para crear un nuevo contacto en el sistema.

Rellenas los campos pertinentes (los campos mínimos obligatorios son: Nombre + Apellidos, o bien, un Correo electrónico)

Con estos pasos, has creado un contacto en el CRM, en el caso que el contacto deba tener una membresía (Castores, Lobatos, Rovers,...) deberás crear al contacto una Membresía.

Crear y editar membresías de un grupo

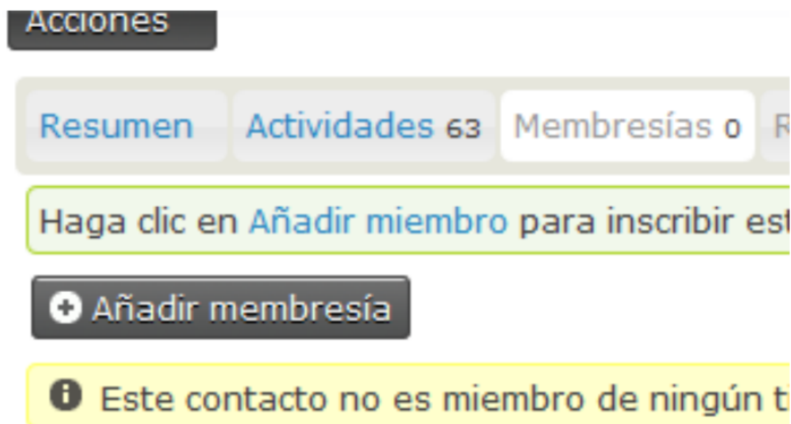

Figura 2.6: Crear y editar Membresías de un grupo. Paso 1. *(Perteneciente a la Guía CRM para Grupo Scout)*

En la ficha del nuevo contacto accederemos a la pestaña de membresías para añadirle una nueva membresía.

Nuevo Miembro
Guardar Guardar y Nuevo Cancelar
Seleccione contacto O - crear nuevo contacto -
Organización y tipo de miembro ASDE España - seleccionar -
- seleccionar -
Castores
Escultas/Pioneros
Lobatos
Responsable Adulto
Rovers/Compañeros
Scouters
Scouts
Número de términos 1
Origen
Miembro desde 08/01/2016
¿Cuándo fue la primera vez que este contacto se hizo miembro?
Fecha Inicio (Limpiar)
Primer día del período actual como miembro. Si no selecciona una fecha la Fecha de Inicio se ajustará automáticamente en función del tipo de miembro.
Fecha finalización (Limpiar)
Último día del período actual como miembro. Si no selecciona una fecha, la Fecha de Fin se establecerá automáticamente en función del tipo de miembro.
¿Sobrescribir estado?
Datos membresía
Número de socio
Grupo Scout relacionado
Nombre específico Sección
Fecha Integración (Limpiar)
Fecha Participación (Limpiar)
Fecha Animación (Limpiar)
Fecha Promesa (Limpiar)
Observaciones a la progresión
Guardar Guardar y Nuevo Cancelar

Figura 2.7: Crear y editar Membresías de un grupo. Paso 2. *(Perteneciente a la Guía CRM para Grupo Scout)*

Seleccionas el tipo de membresía de entre las distintas secciones disponibles. La única fecha obligatoria es la de Miembro desde, puesto que las otras 2 se rellenan de forma automática al grabar la membresía.

Tendrás que tener en cuenta que, de forma automática, pondrá como Fecha de inicio el 1 de septiembre de la ronda en curso. Para que aparezca la fecha concreta, debes introducirla.

La Fecha finalización se establece de forma automática por un periodo de 100 años.” - *Guia CRM para Grupo Scout.*

Arquitectura

Por otra parte, en lo referente a la arquitectura de la solución implementada a nivel nacional, esta aloja en un mismo dispositivo físico, la página web, el servidor y la base de datos. Por lo que se trata de un sistema monolítico que no posee capacidad de escalado de forma sencilla.

2.3.2. Gillweb

Gillweb[5] es un portal de secretaría virtual en línea perteneciente y usado por ASDE-Exploradores de Castilla y León para tratar los datos de educandos y monitores adscritos a la organización federada.

Esta herramienta cuenta con las funcionalidades de la usada a nivel nacional (CiViCRM), y agrega una amplia variedad de funcionalidades a la aplicación, que no se limitan únicamente a almacenar datos. Además, la misma se encuentra presente en diversas plataformas, facilitando así el acceso al público.

Las funcionalidades presentes en el *software* nacional se encuentran aquí bajo el módulo Gillweb que podemos ver en la figura 2.8.

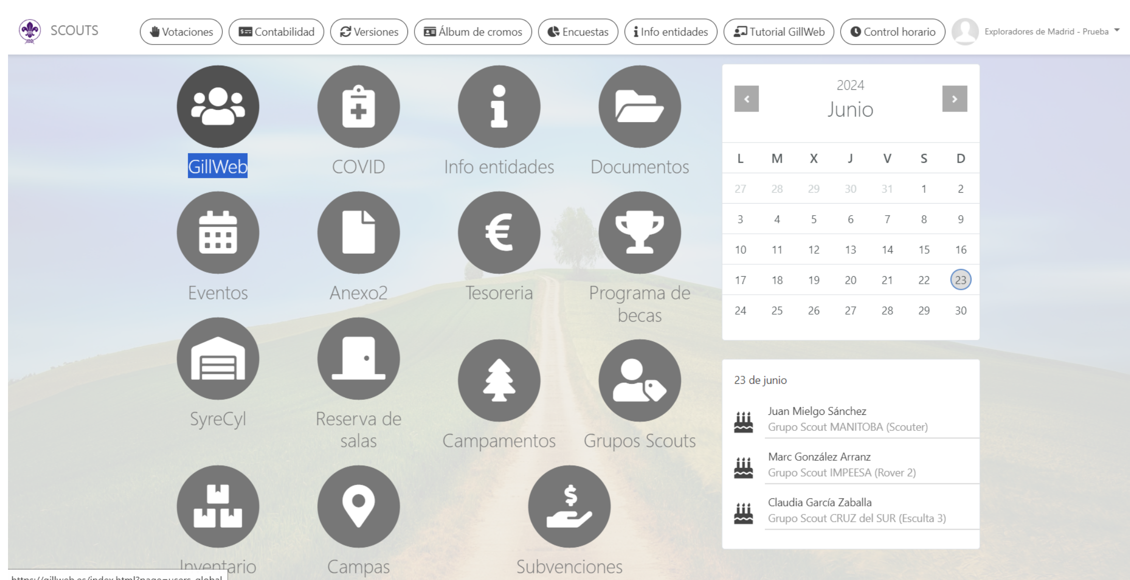

Figura 2.8: Vista del índice en Gillweb.

[5]https://gillweb.es

Ver y generar informes

Para generar informes, una vez estemos dentro del módulo Gillweb, deberemos hacer clic en el desplegable "Informes" y seleccionar la opción que nos interese de entre las disponibles.

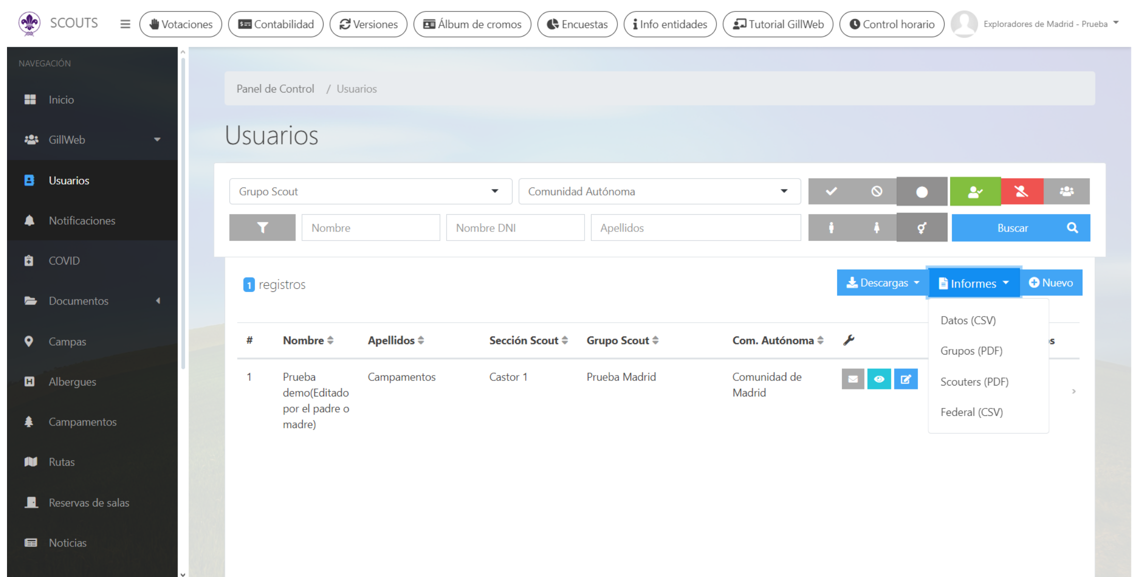

Figura 2.9: Desplegable de informes en vista de los usuarios en Gillweb.

Crear y editar contactos de un grupo

La funcionalidad de crear contactos de un grupo puede ser accedida haciendo clic en el botón con el texto "Nuevo", lo que nos permitirá crear un nuevo usuario. Y la de editar haciendo clic en el botón del lápiz en el usuario correspondiente.

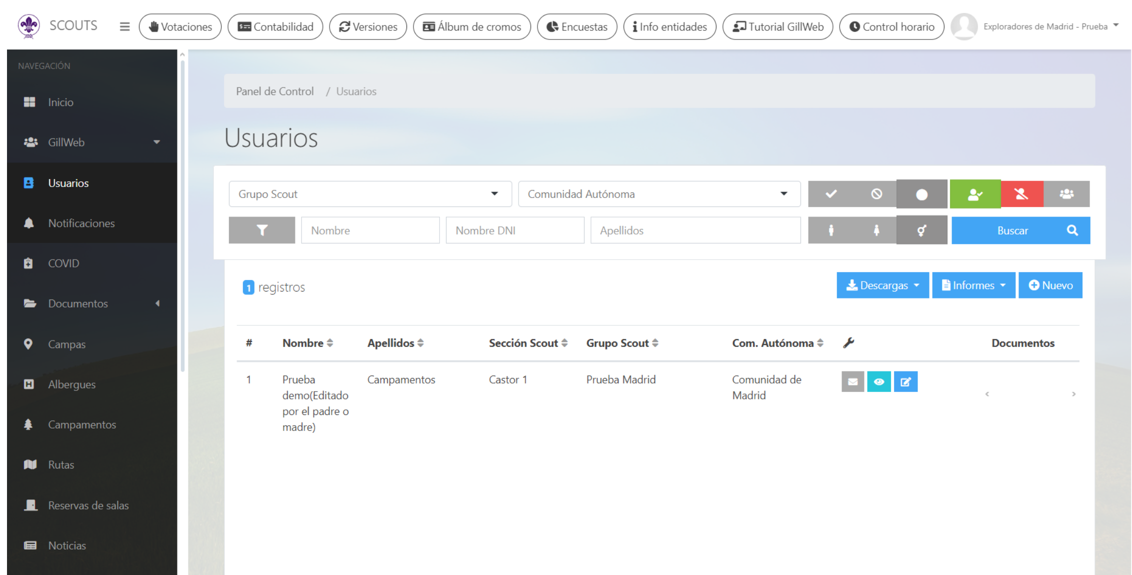

Figura 2.10: Vista de usuarios en Gillweb.

Tras acceder a cualquiera de ambas funciones, el panel resultante será el mismo pero con algunos textos cambiados. Aquí podremos modificar o añadir una amplia variedad de información.

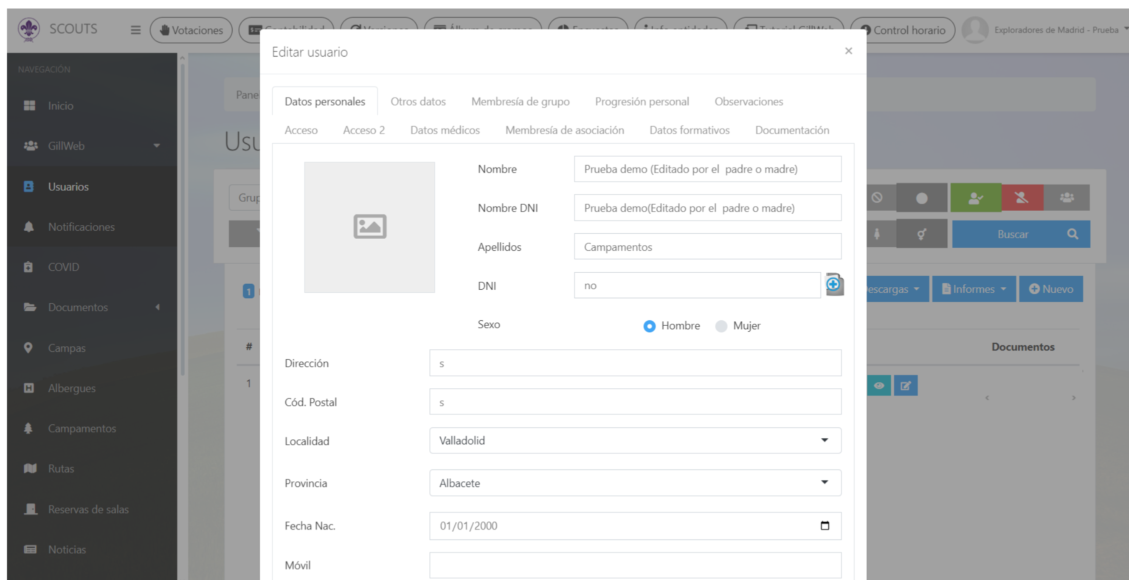

Figura 2.11: Editar usuario en Gillweb.

Crear y editar membresias

Para modificar la membresía de los miembros de los grupos, los pasos a seguir serían los mismos que en la funcionalidad anterior, con la adición de ingresar a la pestaña "Membresía". En esta pestaña podremos modificar los datos relativos a la membresía del usuario.

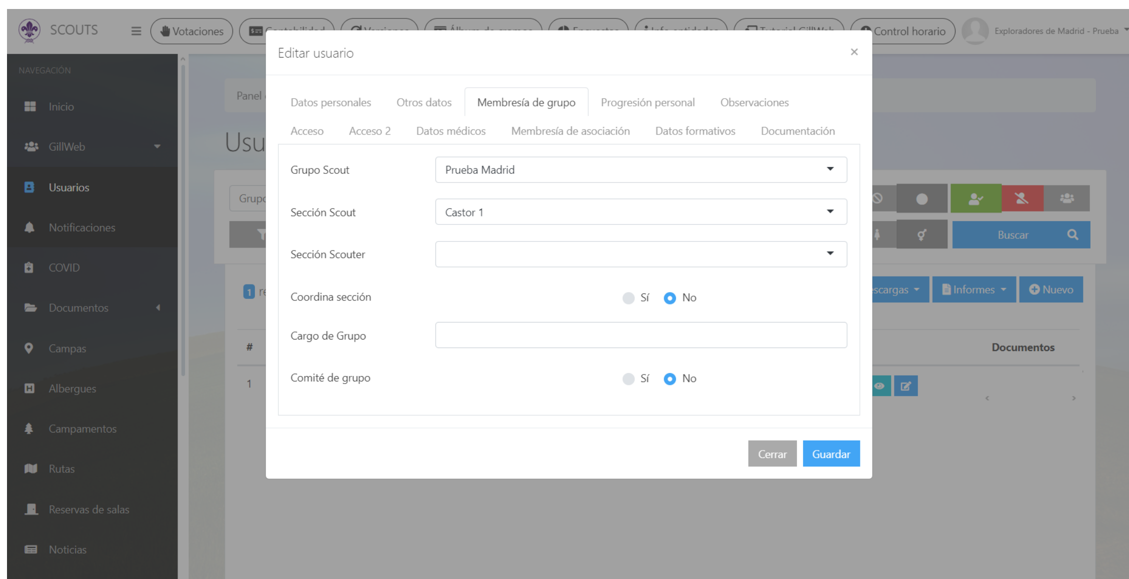

Figura 2.12: Pestaña membresía en editar usuario en Gillweb.

Tesorería

Otra de las funciones principales de Gillweb es la tesorería. En el módulo de tesorería se pueden gestionar y realizar los pagos de los grupos con la federación o viceversa, esto nos permite mantener un registro de las transacciones realizadas.

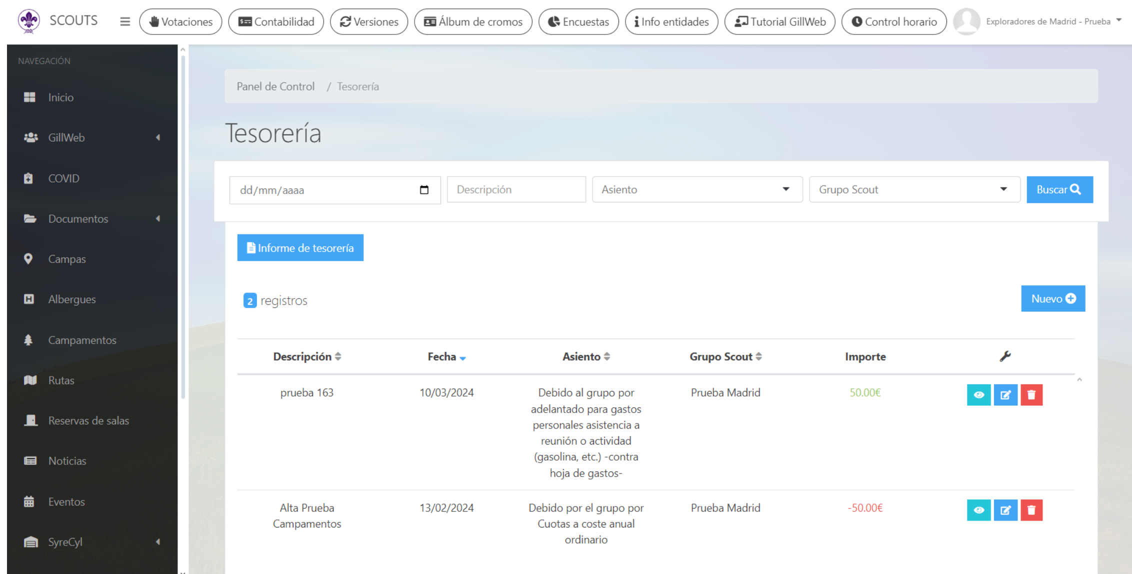

Figura 2.13: Vista tesorería en Gillweb.

Albergues y rutas

Otros dos módulos interesantes son el de albergues y el de rutas. Ambos funcionan de manera similar pero guardan información diferente.

En el primero podremos encontrar una colección de albergues creada de forma comunitaria entre los diferentes grupos con el fin de poder ser consultados y tener la información fácilmente accesible a la hora de planificar actividades.

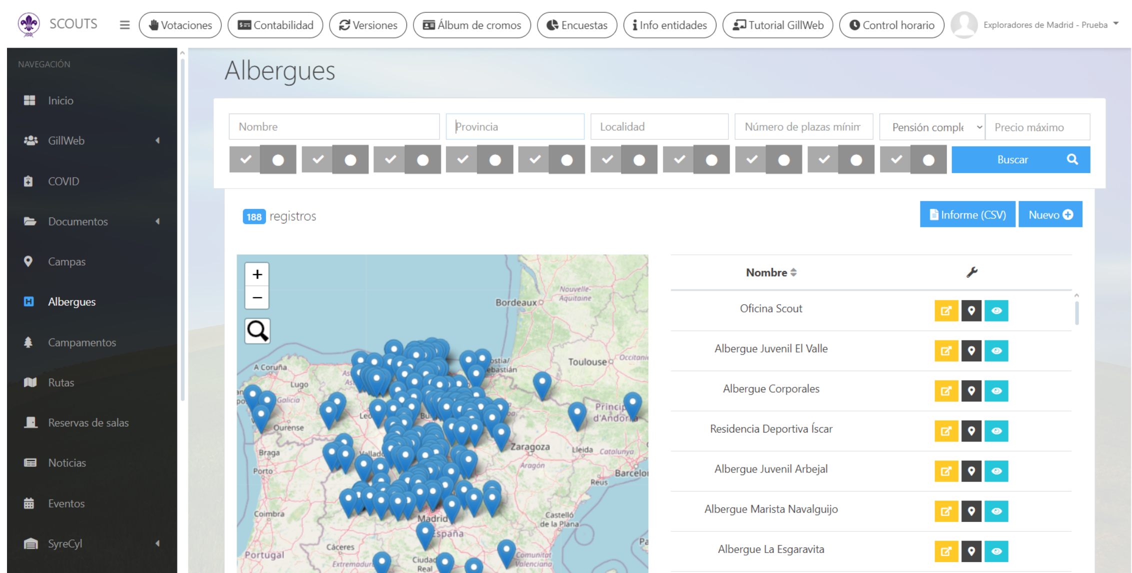

Figura 2.14: Vista albergues en Gillweb.

En el segundo módulo, en cambio, podremos encontrar un listado de rutas que, de igual manera, han sido creadas por la comunidad. Y que nos permitirá obtener y compartir información con el resto de usuarios de la plataforma.

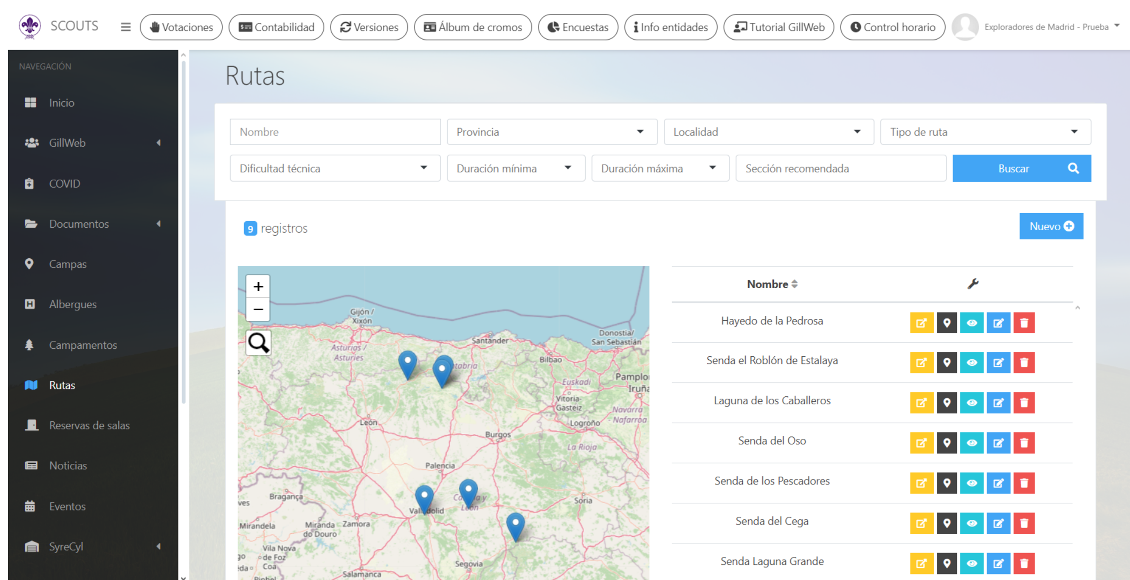

Figura 2.15: Vista de las rutas en Gillweb.

Arquitectura

Esta solución relega el alojamiento de la aplicación a un servicio de alojamiento en línea, que facilita el escalado vertical. Pese a esto, se trata de un sistema monolítico, por lo que el escalado horizontal resulta imposible.

2.4 Comparativa entre las aplicaciones

Tras analizar ambas aplicaciones, se puede crear una tabla comparativa de las funcionalidades más importantes de cada una.

Característica	**CiViCRM**	**Gillweb**
Informes	Si	Si
Gestión de usuarios	Si	Si
Gestión de membresías	Si	Si
Tesorería	No	Si
Albergues	No	Si
Rutas	No	Si

Tabla 2.1: Comparativa entre aplicaciones.

Como podemos ver, Gillweb presenta las mismas características que CiViCRM y añade nuevas características interesantes para los usuarios objetivos.

Respecto a la arquitectura, podemos ver que ambas aplicaciones poseen una carencia en común. Esta es que se encuentran alojadas como aplicaciones monolíticas en un mismo dispositivo. Esto complica el escalado del sistema ante un aumento de la demanda e implica una caída del servicio en su totalidad ante cualquier fallo imprevisto.

Ante los problemas de escalabilidad, como hemos visto anteriormente, se puede emplear el *cloud computing*, sin necesidad de poseer conocimientos sobre redes. Las diversas plataformas que hemos nombrado previamente presentan diferencias, pero ofrecen prácticamente los mismos servicios, manteniendo como funcionalidad principal la misma: distribuir recursos de tecnologías de la Información bajo demanda a través de Internet.

2.5 Propuesta

La propuesta de Safeguard es ofrecer una arquitectura de aplicación distribuida en servicios. Estos servicios se encontrarán virtualizados en una nube comercial, y solo la página web podrá ser accedida desde el cliente; el resto de componentes de la arquitectura se encontrarán ocultos tras la página web, siendo protegidos de esta manera.

La plataforma proveedora de servicios *cloud* elegida será AWS. AWS es la plataforma *cloud* líder en el mercado y cuenta con una amplia cantidad de difusores de contenido versados en la plataforma y una gran cantidad de profesionales, de los cuales puedo aprender en el entorno laboral en el que me encuentro.

Además, ofrece una amplia variedad de servicios virtualizados como alojamiento de páginas web estáticas, almacenamiento de datos, o despliegue de aplicaciones, entre otros. Por otra parte, garantizaremos la escalabilidad con la amplia cantidad de servidores disponibles.

Teniendo todo lo expuesto anteriormente en cuenta, podemos asegurar que la solución planteada garantiza cubrir las posibilidades de mejora que existen en el mercado, con base en las aplicaciones analizadas. Asimismo, la arquitectura del sistema propuesto garantiza la autoescalabilidad de la aplicación mediante la asignación automática de recursos, y la seguridad de los datos tratados, mediante el acceso regulado a los mismos únicamente por la web.

CAPÍTULO 3
Análisis de AWS

Tras elegir la opción de AWS como nuestro proveedor de *cloud*, vamos a revisar la información disponible sobre la plataforma y la oferta de servicios existentes.

3.1 Información general

Amazon nos define AWS de la siguiente manera:

> "Amazon Web Services (AWS) es la nube más adoptada y completa en el mundo, que ofrece más de 200 servicios integrales de centros de datos a nivel global. Millones de clientes, incluso las empresas emergentes que crecen más rápido, las compañías más grandes y los organismos gubernamentales líderes, están usando AWS para reducir los costos, aumentar su agilidad e innovar de forma más rápida."[4]

Además, también nos habla de diferentes aspectos de su plataforma:

> **Funcionalidad**
>
> "AWS cuenta con una cantidad de servicios y de características incluidas en ellos que supera la de cualquier otro proveedor de la nube, ofreciendo desde tecnologías de infraestructura como cómputo, almacenamiento y bases de datos hasta tecnologías emergentes como aprendizaje automático e inteligencia artificial, lagos de datos y análisis e internet de las cosas. Esto hace que llevar las aplicaciones existentes a la nube sea más rápido, fácil y rentable y permite crear casi cualquier cosa que se pueda imaginar.
>
> AWS también tiene la funcionalidad más completa dentro de esos servicios. Por ejemplo, AWS ofrece la más amplia variedad de bases de datos que están diseñadas especialmente para diferentes tipos de aplicaciones, de modo que usted puede elegir la herramienta adecuada para el trabajo a fin de obtener el mejor costo y rendimiento."[4]
>
> **Comunidad**
>
> "AWS tiene la comunidad más grande y dinámica, con millones de clientes activos y decenas de miles de socios en todo el mundo. Los clientes de prácticamente todos los sectores y de todos los tamaños, lo cual incluye las empresas emergentes, las compañías y las organizaciones del sector público, ejecutan todos los casos de uso imaginables en AWS. La red de socios de AWS (APN)

incluye miles de integradores de sistemas que se especializan en los servicios de AWS y decenas de miles de proveedores de software independientes (ISV) que adaptan su tecnología para que funcione en AWS."[4]

Seguridad

"AWS está diseñado para ser el entorno de informática en la nube más flexible y seguro disponible en la actualidad. Nuestra infraestructura principal se creó para cumplir con los requisitos de seguridad del ejército, los bancos internacionales y otras organizaciones que deben cumplir requisitos de confidencialidad estrictos. Cuenta con el respaldado de un amplio conjunto de herramientas de seguridad en la nube, con más de 300 servicios y funciones de seguridad, conformidad y gobernanza, así como compatibilidad con 143 normas de seguridad y certificaciones de conformidad."[4]

Experiencia e historia

"La experiencia, madurez, fiabilidad, seguridad y rendimiento de AWS son inigualables y están al servicio de sus aplicaciones más importantes. Durante más de 17 años, AWS ha brindado servicios en la nube a millones de clientes en todo el mundo que ejecutan una amplia variedad de casos de uso. AWS ofrece una mayor experiencia operativa, a gran escala, que cualquier otro proveedor de nube."[4]

3.2 Servicios

AWS ofrece una amplia variedad de servicios, como hemos visto en el apartado anterior. Estos servicios están clasificados en varias categorías, siendo las más interesantes para nuestro trabajo:

3.2.1. Informática

Dentro de la categoría de informática podemos encontrar servicios como EC2 o Lambda que nos permiten crear máquinas virtuales o instanciar aplicaciones a partir del código.

3.2.2. Almacenamiento

En la categoría de almacenamiento tenemos los servicios S3 y AWS Backup, entre otros, que nos permiten almacenar código de aplicaciones y realizar copias de seguridad de las mismas respectivamente.

3.2.3. Base de datos

Dentro de la oferta de servicios de bases de datos de AWS podemos encontrar una amplia variedad, siendo las más interesantes RDS y DynamoDB. RDS es la opción más popular para bases de datos relacionales, mientras que DynamoDB, en el polo opuesto, es la más popular para bases de datos no relacionales.

3.2.4. Redes y entrega de contenido

En el apartado de redes y entrega de contenido tenemos los servicios VPC, CloudFront, API Gateway y una amplia variedad de diferentes servicios que nos permiten gestionar el apartado de redes de nuestra aplicación.

3.2.5. Seguridad, identidad y conformidad

Para suplir las necesidades de seguridad tenemos una gran variedad de servicios, algunos que podrían resultar interesantes para la arquitectura son Cognito, WAF & Shield o IAM.

3.2.6. Móvil

Por último, de la categoría de servicios Móvil, podemos destacar AWS Amplify que nos permite desplegar aplicaciones enteras o páginas web estáticas.

CAPÍTULO 4
Análisis del problema

Desde el punto de vista de un arquitecto de *software*, la forma más eficiente de evaluar un problema es haciendo uso de los atributos de calidad. Los atributos de calidad son el resultado de todas las necesidades de las personas que usarán la arquitectura Safeguard, y que tienen que ser definidos claramente.

Para definir estos atributos haremos uso de un árbol de utilidad, donde los atributos de calidad serán las ramas, y los requisitos no funcionales serán las hojas. Para ello, definiremos primero quiénes serán las personas que usarán la arquitectura o *stakeholders* y las necesidades de los mismos en la tabla 4.1.

Rol	Necesidades
Desarrollador	**Mantenibilidad del código:** el código debería estar bien organizado y ser fácilmente mantenible. **Modularidad:** la arquitectura del sistema debería soportar el desarrollo modular para poder desarrollar cada servicio independientemente.
Administrador del sistema	**Escalabilidad:** el sistema debería tener capacidad de autoescalado para poder responder a flujos altos de usuarios en momentos puntuales. **Fiabilidad:** los componentes del sistema deberían ser fiables, minimizando la indisponibilidad del mismo.
Usuarios finales	**Capacidad de respuesta:** el sistema debería dar tiempos de respuesta rápidos. **Interoperabilidad:** el sistema debería ser accesible desde diversos dispositivos. **Seguridad:** el sistema debería garantizar la seguridad de los datos de los usuarios.

Tabla 4.1: *Stakeholders* de la arquitectura y sus necesidades.

Tras ver los diferentes *stakeholders* que harán uso de nuestra arquitectura, el siguiente paso es definir los atributos de calidad basándonos en las necesidades encontradas de los mismos. Para ello, vamos a hacer uso del estándar **ISO/IEC 25000**, conocido co-

mo **SQuaRE** (System and Software Quality Requirements and Evaluation, «Requisitos y Evaluación de la Calidad de Sistemas y Software»)[5].

Este estándar reemplaza a dos estándares previos: **ISO/IEC 9126** (*Software Product Quality*) e **ISO/IEC 14598** (*Software Product Evaluation*), uniendo así, en un mismo estándar renovado, ambas vertientes. Dentro de esta nueva unión, que es el estándar **ISO/IEC 25000**, podemos encontrar el **ISO/IEC 25010**. Este nos define un modelo de calidad compuesto por nueve características, las cuales a su vez están desglosadas, y que podemos ver a continuación en la figura 4.1.

CALIDAD DEL PRODUCTO SOFTWARE

ADECUACIÓN FUNCIONAL	EFICIENCIA DE DESEMPEÑO	COMPATIBILIDAD	CAPACIDAD DE INTERACCIÓN	FIABILIDAD	SEGURIDAD	MANTENIBILIDAD	FLEXIBILIDAD	PROTECCIÓN
COMPLETITUD FUNCIONAL	COMPORTAMIENTO TEMPORAL	COEXISTENCIA	RECONOCIBILIDAD DE ADECUACIÓN	AUSENCIA DE FALLOS	CONFIDENCIALIDAD	MODULARIDAD	ADAPTABILIDAD	RESTRICCIÓN OPERATIVA
CORRECCIÓN FUNCIONAL	UTILIZACIÓN DE RECURSOS	INTEROPERABILIDAD	APRENDIZABILIDAD	DISPONIBILIDAD	INTEGRIDAD	REUSABILIDAD	ESCALABILIDAD	IDENTIFICACIÓN DE RIESGOS
PERTINENCIA FUNCIONAL	CAPACIDAD		OPERABILIDAD	TOLERANCIA A FALLOS	NO-REPUDIO	ANALIZABILIDAD	INSTALABILIDAD	PROTECCIÓN ANTE FALLOS
			PROTECCIÓN FRENTE A ERRORES DE USUARIO	RECUPERABILIDAD	RESPONSABILIDAD	CAPACIDAD DE SER MODIFICADO	REEMPLAZABILIDAD	ADVERTENCIA DE PELIGRO
			INVOLUCRACIÓN DEL USUARIO		AUTENTICIDAD	CAPACIDAD DE SER PROBADO		INTEGRACIÓN SEGURA
			INCLUSIVIDAD		RESISTENCIA			
			ASISTENCIA AL USUARIO					
iso25000.com			AUTO-DESCRIPTIVIDAD					

Figura 4.1: Características de calidad definidas según el estándar **ISO/IEC 25010**.

4.1 Árbol de utilidad

Posteriormente a ver las diferentes características de calidad disponibles, el siguiente paso es definir la lista de prioridades para clasificar los diferentes requisitos. Para ello, haremos uso de la escala **MoSCoW** (*Must have, Should have, Could have, Won't have*), además, tratándose de un proyecto, sería adecuado añadir la complejidad esperada para satisfacer dichos requisitos. Partiendo de estas premisas, podemos definir el árbol de utilidad de la arquitectura en la tabla 4.2.

Id	***Stakeholder***	**Característica de calidad**	**Requisito**	**Prioridad**	**Complejidad**
R1	Desarrollador	Mantenibilidad	Capacidad de ser modificado	Debería tener	Media
R2	Desarrollador	Mantenibilidad	Modularidad	Tiene que tener	Fácil
R3	Administrador del sistema	Flexibilidad	Escalabilidad	Tiene que tener	Fácil
R4	Administrador del sistema	Fiabilidad	Disponibilidad	Podría tener	Media
R5	Usuarios finales	Eficiencia de desempeño	Comportamiento temporal	Podría tener	Media
R6	Usuarios finales	Compatibilidad	Interoperabilidad	Podría tener	Media
R7	Usuarios finales	Seguridad	Confidencialidad	Tiene que tener	Media

Tabla 4.2: Árbol de utilidad de la arquitectura Safeguard.

- **R1:** Este requisito se relaciona fuertemente con el objetivo 2.4, el cual nos dice que debe poder servir de base para desarrollar aplicaciones futuras, por lo que que el código pueda ser modificado es un punto fuerte de la arquitectura.

- **R2:** Este requisito se relaciona de igual forma con el objetivo 2.4, siendo imprescindible para la arquitectura, ya que al tratarse de una base para una aplicación multiplataforma, los componentes de la misma deben ser modulares para poder ser intercambiados.

- **R3:** El requisito R3, como su propio requisito indica, está relacionado con el objetivo 2.3 para garantizar la escalabilidad.

- **R4:** El requisito de la disponibilidad sirve para cumplir el objetivo 2.3 garantizando que cuando escale siga estando disponible, respondiendo así ante la demanda fluctuante.

- **R5:** El requisito R5 se relacionaría con el objetivo 3.0 ya que en el producto final de la prueba de concepto es recomendable tener tiempos de respuesta rápidos para los usuarios.

- **R6:** El requisito R6, al igual que el R5, se relaciona con el objetivo 3.0, ya que es un requisito de calidad recomendable a tener en una prueba de concepto.

- **R7:** El requisito R7 es de vital importancia, ya que responde al objetivo 2.2 de garantizar el acceso restringido y protegido a los datos.

4.2 Análisis de la seguridad

Tratándose de una arquitectura diseñada para el tratamiento de datos sensibles, un punto clave en la misma es la seguridad, como hemos visto anteriormente en la definición de objetivos, concretamente en el 2.2.

Queremos que el acceso a los datos esté cifrado, y solo puedan acceder personas pertenecientes a la organización, y que, además, la transmisión de los mismos se produzca en una subred aparte, solamente accesible a través de un *proxy*.

Con AWS podemos garantizar la seguridad de la arquitectura mediante el uso de *tokens* de autenticación y la restricción del acceso a los servicios.

4.3 Análisis del marco legal y ético

En el apartado del marco legal, encontramos un tema a tener en cuenta por todo profesional que trate con datos de usuarios. La ley de protección de datos (o Ley Orgánica de Protección de Datos Personales y garantía de los derechos digitales) [6].

Teniendo en cuenta que se trata de una arquitectura basada en microservicios alojados en la nube, todo el tratamiento de datos se hará en la red. Esto implica que, aparte de asegurar el acceso a los mismos, solo deben poder ser accedidos por las personas pertinentes dependiendo de las reglas de la organización y de acuerdo a la ley anteriormente mencionada.

Para garantizar el cumplimiento con el marco legal y ético en el tratamiento de datos, haremos uso de AWS para garantizar que el acceso a los datos solo pueda ser accedido por personas pertenecientes a la asociación. Además, Cognito nos ofrece opciones para definir roles de usuarios o crear cuentas de usuario únicamente manualmente.

4.4 Análisis de la escalabilidad

Respecto a la escalabilidad, es un tema complejo debido a los usuarios objetivos. Para esta arquitectura no se espera una afluencia constante de usuarios; en cambio, se esperan picos puntuales de afluencia. Por ello, es necesario el uso de una arquitectura que sea capaz de autoescalar cada componente individualmente y que responda ante la demanda.

4.5 Identificación y análisis de soluciones posibles

Con base en los análisis realizados previamente, podemos destacar que existen diversos requisitos arquitectónicos, legales, de seguridad y de escalabilidad. A partir de estos requisitos se han desarrollado diferentes propuestas arquitectónicas a nivel de servicios.

4.5.1. Máquinas virtuales

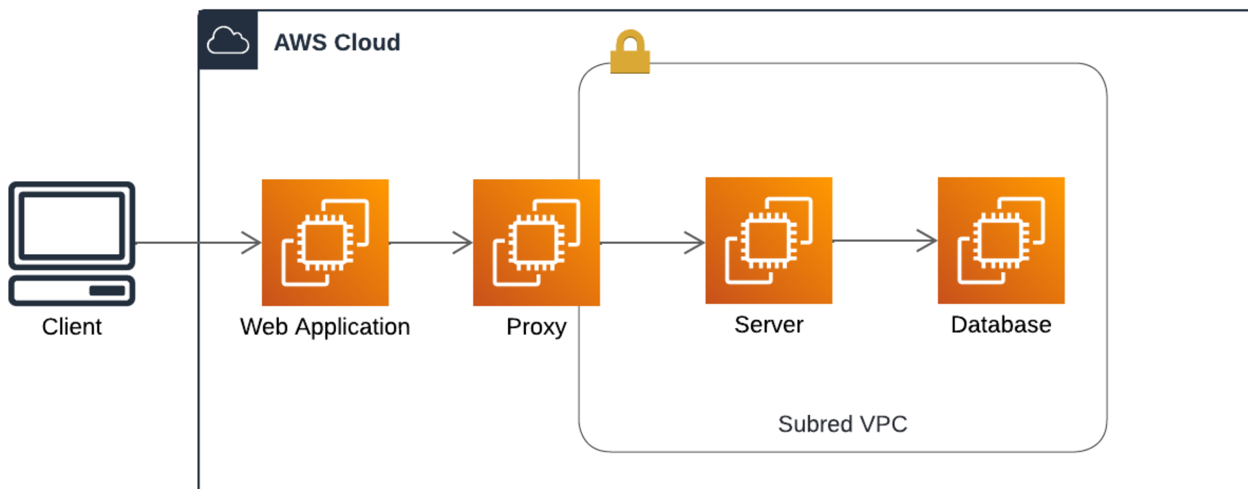

Figura 4.2: Diagrama de arquitectura de la aplicación con instancias EC2 (Elaboración propia).

La propuesta basada en máquinas virtuales consiste en usar diferentes instancias del servicio **Amazon EC2** (*Elastic Compute Cloud*) para alojar la aplicación web, el *proxy*, el servidor y la base de datos.

La base de datos y el servidor se encontrarían en un *Virtual Private Cloud* (VPC), accesible únicamente a través del *proxy*. Un VPC funciona de igual manera que un *cloud* normal, pero con la ventaja de tener los servicios desplegados en la misma, aislados del resto de servicios del *cloud*. Asimismo, el VPC únicamente habilitaría las conexiones autentificadas previamente en la aplicación web mediante la definición de reglas de seguridad.

Los pros de esta solución residen en el casi nulo conocimiento, respecto a AWS, requerido para iniciar las instancias y poner en marcha la arquitectura y, por otra parte, la posibilidad de autoescalado innata en los servicios de AWS.

Los contras son el requerimiento de conocimiento en redes para configurar correctamente la VPC y las conexiones con el fin de cumplir el requisito de seguridad. Esto es debido a que no se hace uso de uno de los principales beneficios de usar *cloud computing*,

la posibilidad de abstraer todo el proceso de despliegue de la aplicación desde una interfaz intuitiva que maneja la configuración por nosotros. Por otra parte, también sería vulnerable a una amplia diversidad de ataques informáticos debido a la falta de un *Web Application Firewal* (WAF). Además, el uso del servicio **EC2** está limitado a 12 meses de uso gratuito.

4.5.2. Servicios AWS Free tier

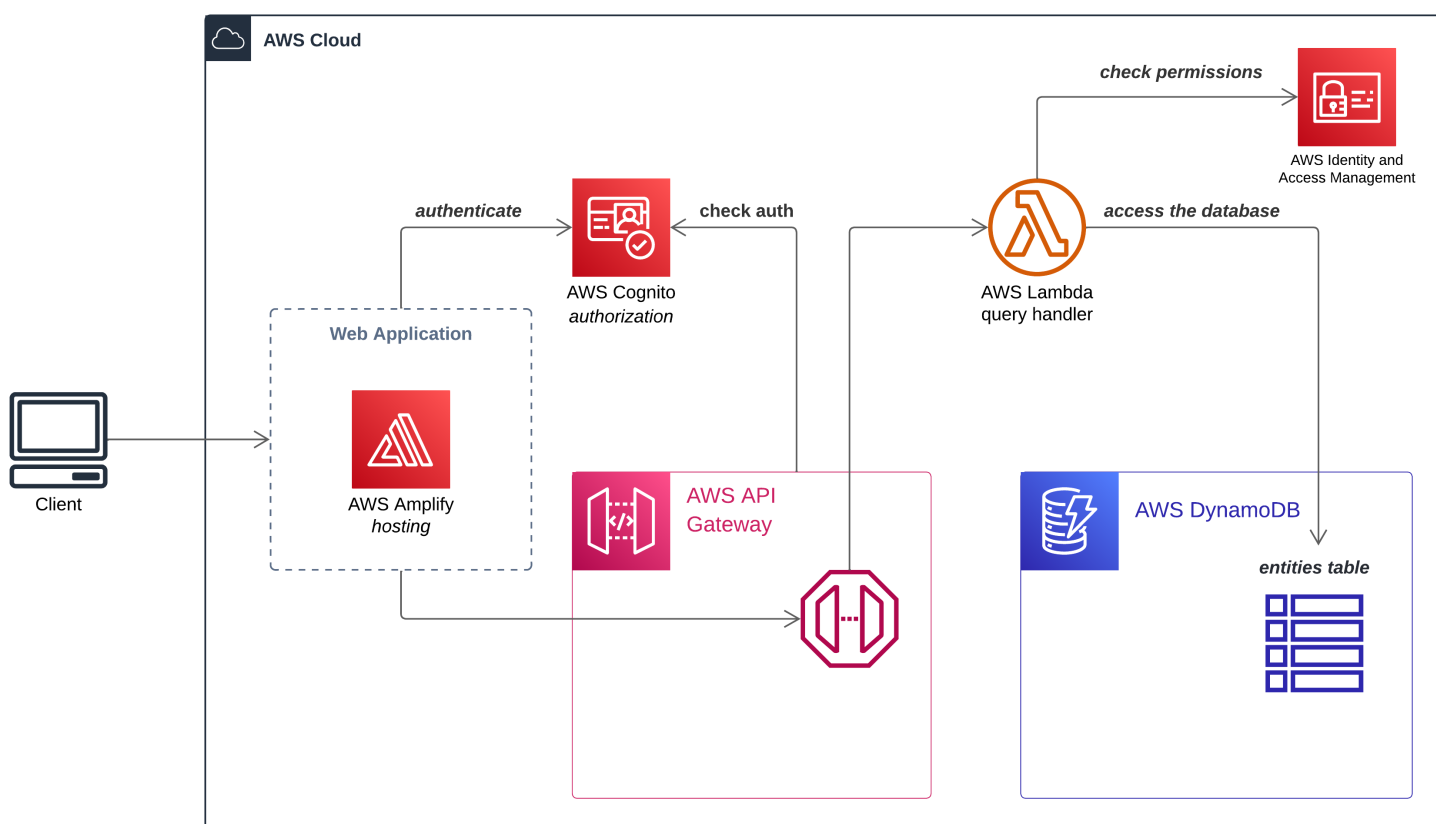

Figura 4.3: Diagrama de arquitectura de la aplicación con servicios del free tier de AWS (Elaboración propia).

La propuesta desarrollada usando el *Free tier* de AWS haría uso de una amplia variedad de servicios. Cada uno presentaría una funcionalidad descrita a continuación:

Amplify: Se encargaría de compilar y alojar la aplicación web.

Cognito: Se encargaría de la gestión, autenticación y comprobación de usuarios.

API Gateway: Se encargaría de validar las conexiones con el servicio Cognito y redirigir las conexiones a la función Lambda.

Lambda: Se encargaría de almacenar y ejecutar el código del servidor ante una solicitud recibida por el servicio de API Gateway.

Identity and Access Management (IAM): Se encargaría de gestionar los roles de permisos asignados a cada servicio de AWS para autorizar comportamientos (p. ej. leer o modificar una base de datos).

DynamoDB: Se encargaría de almacenar tablas de datos no relacionales para el uso por parte del resto de servicios de AWS.

Como hemos visto, cada servicio tendría un rol único y definido en la arquitectura. Amplify alojaría la página web que enviaría solicitudes a la **API Gateway**; esta llamaría a la función **Lambda** para ejecutar el backend y hacer las operaciones pertinentes en la tabla de **DynamoDB**. Todo esto además, mientras **Cognito** protege el acceso al backend e **IAM** regula el acceso a la base de datos.

Los pros de esta solución serían el **autoescalado** de cada servicio independientemente, el uso de la interfaz de AWS para desplegar los servicios, y la posibilidad de configurar las **medidas de seguridad** desde la interfaz, omitiendo así requerimientos en conocimiento de redes. Además, en su mayoría los servicios empleados son **gratuitos**, con un límite mensual que deja holgura ante la afluencia normal de los usuarios objetivos.

Los contras de esta solución radicarían en el uso obligatorio de una **base de datos no relacional** (disminuyendo así la cantidad de opciones tecnológicas cuando se use la arquitectura), la **vulnerabilidad a ataques informáticos** a la web por la falta de un WAF, y el uso gratuito **limitado a 12 meses** de Amplify y API Gateway.

4.5.3. Servicios AWS Pago

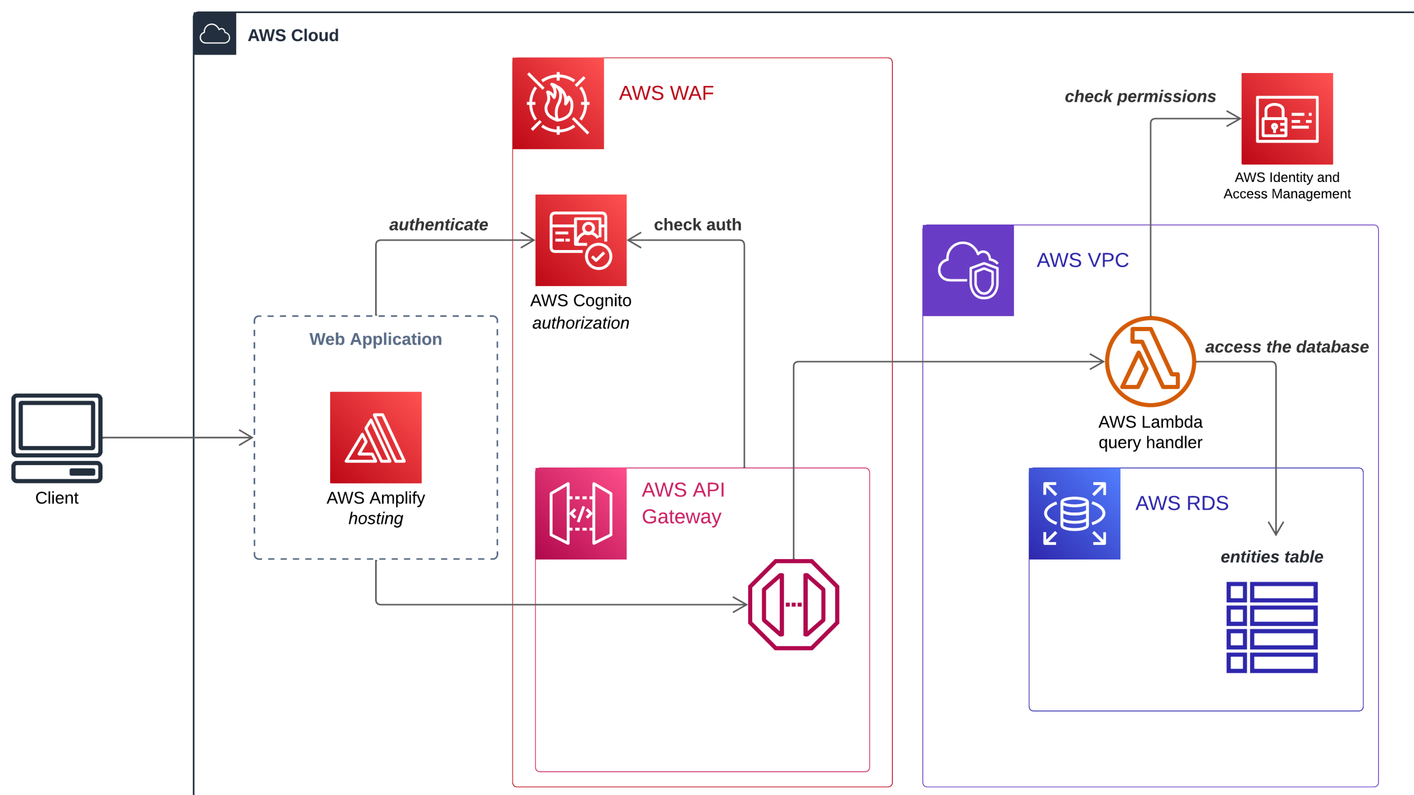

Figura 4.4: Diagrama de arquitectura de la aplicación con servicios de pago de AWS (Elaboración propia).

Esta propuesta está basada en la anterior; la diferencia radicaría en la adición de un **WAF** y el uso del servicio **Amazon RDS** junto a una **VPC**. Estos tres nuevos componentes funcionarían de la siguiente manera:

WAF: Se encargaría de interceptar y analizar cada solicitud HTTP recibida por los servicios protegidos y compararlo con una serie de reglas definidas previamente para autorizar la solicitud.

RDS: Se encargaría de almacenar tablas de datos relacionales para el uso por parte del resto de servicios de AWS.

VPC: Se encargaría de crear una subred para proteger el acceso al servicio RDS.

Los pros de esta solución serían los mismos que la anterior propuesta, con la adición de la **protección** de la aplicación web con un **WAF** y la posibilidad de utilizar el tipo de base de datos que más se ajuste a las necesidades del usuario.

En cambio, los contras serían que esta opción implicaría un **coste mensual** por la protección del WAF y el uso de la VPC, ya que el acceso a la base de datos RDS estaría expuesto y protegido únicamente con par de usuario/clave.

4.5.4. Comparación entre las tres soluciones

A modo de resumen, podemos definir una tabla que compare las características principales de las diferentes soluciones propuestas.

Característica	**Maquinas virtuales**	**AWS Free Tier**	**AWS Pago**
Modularización de componentes	Si	Si	Si
Conocimiento de AWS requerido	No	Si	Si
Conocimiento de redes requerido	Si	No	No
Autoescalado	Si	Si	Si
Abstracción del despliegue de servicios	No	Si	Si
Tiene WAF	No	No	Si
Gratuito	Si	Si	No
Libertad elección base de datos	Si	No	Si
Necesita VPC	Si	No	Si

Tabla 4.3: Comparativa entre arquitecturas.

4.6 Solución propuesta

La solución elegida, tras el análisis anterior, será la de los servicios AWS con el *Free tier*.

La elección de la arquitectura se ha fundamentado principalmente en que tanto la solución AWS Free Tier como la de AWS Pago cumplían con los requisitos definidos con anterioridad en la tabla 4.2.

La diferencia radica en que el componente monetario tiene más peso en este caso que la protección que pueda dar un WAF, ya que tal y como se presentó al principio del documento, los usuarios objetivos "son organizaciones pequeñas y medianas, que no necesitan o no pueden invertir grandes cantidades de recursos" y un WAF al final contribuye a la disponibilidad del sistema en vez de a la confidencialidad de los datos.

Este tipo de organizaciones generalmente no poseen la capacidad de invertir mensualmente en mantener una página web, ya que el dinero ingresado por parte de los socios está destinado a la realización de actividades para los mismos.

Con la solución elegida, el pago debería empezar a realizarse al año de iniciar el sistema distribuido, con la posibilidad de crear una cuenta nueva y reiniciar el proceso de despliegue de los servicios. El problema de este proceso sería la necesidad de mudar la información de la base de datos de una cuenta a la nueva y la necesidad de configurar de nuevo algunas variables del entorno y servicios de AWS.

4.7 Plan de Trabajo

El plan de trabajo inicial para desarrollar el proyecto está planteado en etapas mediante un **diagrama de Gantt** y dividido en tareas mediante un tablero **Kanban**. Todo esto está alojado en un proyecto de **Github** para facilitar la integración con el código.

El **diagrama de Gantt** está conformado por un total de 11 etapas, las cuales están planteadas en días.

Nº	Nombre de la etapa	Estimación (días)	Fecha de finalización
1	Plan de trabajo	3	11/04/2024
2	Investigación y diseño	13	24/04/2024
3	Preparación del proceso de *software*	4	28/04/2024
4	Configuración del entorno de desarrollo	10	08/05/2024
5	Desarrollo del *Backend* y la BD	9	17/05/2024
6	Desarrollo del *Proxy*	5	22/05/2024
7	Desarrollo de la Aplicación Web	6	28/05/2024
8	Pruebas y depuración	5	02/06/2024
9	Creación de la prueba de concepto	6	08/06/2024
10	Despliegue y configuración	7	15/06/2024
11	Redacción de la memoria	13	28/06/2024
	Total	81	

Tabla 4.4: Plan de trabajo inicial en días.

Cada una de las diferentes etapas presenta tareas a realizar, las cuales definen el contenido de la misma.

1. **Plan de trabajo:** en esta etapa se crearán y definirán los puntos de la memoria del trabajo. Asimismo, se diseñará el plan de trabajo a seguir para la elaboración del mismo.

2. **Investigación y diseño:** el objetivo de esta etapa es investigar el tema elegido y diseñar una solución que supla las carencias encontradas en el tema.

3. **Preparación del proceso del software:** para completar esta etapa será necesario crear un proyecto de GitHub y los repositorios correspondientes, además de crear *milestones* para cada etapa y tareas dentro de cada una.

4. **Configuración del entorno de desarrollo:** en esta etapa buscamos crear una cuenta de AWS y configurar todo lo necesario para poder empezar a trabajar con los servicios de la plataforma. Además, también se configurarán los diferentes IDEs y se conectarán los repositorios de GitHub.

5. **Desarrollo del Backend y la BD:** el objetivo de esta etapa es definir el modelo de la base de datos y desarrollar el *backend* de la arquitectura para poder interactuar con la base de datos.

6. **Desarrollo del Proxy:** para esta etapa buscamos crear el *proxy*, que haga de intermediario entre el *backend* y el *frontend*, y que se encargue de analizar las peticiones que le lleguen.

7. **Desarrollo de la aplicación web:** esta etapa será el culmen del sistema, siendo la misma en la que se desarrollará el *frontend* de la aplicación y que conectará con el *proxy* enlazando todas las piezas del sistema.

8. **Pruebas y depuración:** tras desarrollar la arquitectura, será necesario realizar pruebas y depurar el código en busca de errores.

9. **Creación de la prueba de concepto:** una vez que la arquitectura sea estable y haya pasado las pruebas pertinentes, se desarrollará una prueba de concepto basándonos en la tesorería de un grupo scout.

10. **Despliegue y configuración:** en penúltima instancia, esta etapa asegurará que la arquitectura sea desplegada en la nube y se configure correctamente.

11. **Redacción de la memoria:** por último, en esta etapa se dará forma y estructura a toda la información que se ha ido escribiendo en el documento de la memoria creado en la primera etapa y se añadirá información relevante.

Tras la estimación inicial, una estimación más concreta en horas tuvo lugar. Cada día de trabajo correspondería a una jornada de cuatro horas, dando lugar a un cómputo total de 324 horas de trabajo. Esta estimación estaría pendiente de ser modificada más adelante según fuera avanzando el proyecto y se fueran concretando las tareas a realizar.

Nº	Nombre de la etapa	Estimación (horas)	Fecha de finalización
1	Plan de trabajo	12	11/04/2024
2	Investigación y diseño	52	24/04/2024
3	Preparación del proceso de *software*	16	28/04/2024
4	Configuración del entorno de desarrollo	40	08/05/2024
5	Desarrollo del *Backend* y la BD	36	17/05/2024
6	Desarrollo del *Proxy*	20	22/05/2024
7	Desarrollo de la Aplicación Web	24	28/05/2024
8	Pruebas y depuración	20	02/06/2024
9	Creación de la prueba de concepto	24	08/06/2024
10	Despliegue y configuración	28	15/06/2024
11	Redacción de la memoria	52	28/06/2024
	Total	324	

Tabla 4.5: Plan de trabajo inicial en horas.

A medida que se iba avanzando con la etapa de Investigación y diseño, fue surgiendo un problema. Debido a que la complejidad de AWS fue mayor a la esperada, se tomó la decisión de dividir la etapa de "Investigación y diseño" en dos subetapas. Además, debido a la necesidad de investigar las tecnologías compatibles con AWS y los servicios disponibles, tuve que establecer un proyecto base con *frontend* y *backend* antes de lo esperado y reorganizar algunas tareas. Esto dio lugar a una actualización y reorganización de la planificación:

Nº	Nombre de la etapa	Ejecución (horas)	Fecha de finalización
1	Plan de trabajo	12	11/04/2024
2	Investigación	40	26/04/2024
3	Preparación del proyecto	20	21/04/2024
4	Configuración de la arquitectura y AWS	60	10/05/2024
5	Diseño	16	10/05/2024
6	Desarrollo del *Backend* y la BD	28	17/05/2024
7	Desarrollo de la Aplicación Web	24	22/05/2024
8	Pruebas y depuración	20	22/05/2024
9	Creación de la prueba de concepto	24	29/05/2024
10	Redacción de la memoria	52	28/06/2024
	Total	296	

Tabla 4.6: Plan de trabajo actualizado en horas.

Finalmente, el plan de trabajo ejecutado fue el siguiente:

Nº	Nombre de la etapa	Ejecución (horas)	Fecha de finalización
1	Plan de trabajo	8	10/04/2024
2	Investigación	43	26/04/2024
3	Preparación del proyecto	22	21/04/2024
4	Configuración de la arquitectura y AWS	80	15/05/2024
5	Diseño	10	15/05/2024
6	Desarrollo del *Backend* y la BD	20	20/05/2024
7	Desarrollo de la Aplicación Web	22	25/05/2024
8	Pruebas y depuración	15	26/05/2024
9	Creación de la prueba de concepto	24	01/06/2024
10	Redacción de la memoria	60	20/06/2024
	Total	304	

Tabla 4.7: Plan de trabajo ejecutado en horas.

4.8 Presupuesto

Tal y como hemos visto en el apartado anterior, el trabajo a realizar correspondería a un total de 300 horas aproximadamente, y teniendo en cuenta que el sueldo medio de un arquitecto de *software* en España ronda los 53.000 € por año, o lo que es lo mismo, 27,18€ por hora, podemos asegurar que tendría un coste total de 8154€ u 8200€ aproximado.

Respecto a material *hardware*, será necesario un dispositivo donde configurar la arquitectura del sistema en la plataforma *cloud*, y poder trabajar. En cambio, para el material *software*, únicamente necesitaremos un IDE de nuestra elección para desarrollar la arquitectura.

CAPÍTULO 5
Diseño de la solución

En este capítulo vamos a realizar el diseño de la arquitectura de la aplicación partiendo de la propuesta elegida anteriormente, y veremos las tecnologías a utilizar para desarrollar la misma. En concreto, para el diseño de la arquitectura, definiremos el medio de comunicación entre los servicios, veremos un ejemplo de interacción con la arquitectura resultante y cómo actúan los diferentes servicios de la arquitectura final.

5.1 Arquitectura del Sistema

Tal y como hemos visto en el capítulo anterior, la arquitectura que usaremos será la de la figura 4.3. *Grosso modo*, podemos destacar 3 bloques principales. El *frontend* o aplicación web, que está alojado en el servicio de Amplify, el *backend* o servidor, que está alojado en Lambda, y la base de datos, que está alojada en DynamoDB. La problemática pendiente a tratar sería la comunicación entre los diferentes servicios.

5.2 Diseño Detallado

5.2.1. Comunicación entre servicios

La comunicación entre los servicios es, sin duda, la parte esencial de la arquitectura, tratándose de un sistema distribuido. Para efectuar la misma, se empleará una arquitectura **REST**.

La arquitectura REST, en palabras del equipo de Codecademy - una plataforma de aprendizaje de programación con más de 45 millones de usuarios - es definida de la siguiente forma:

> Los sistemas conformes con REST, normalmente llamados sistemas RESTful, se caracterizan por ser sistemas sin estado y separar las responsabilidades cliente y servidor.
>
> En el estilo arquitectónico REST, la implementación del cliente y del servidor puede ser hecha independientemente sin que ninguno conozca sobre el otro. Esto significa que el codigo en el lado del cliente puede ser cambiado en cualquier momento sin afectar el funcionamiento del servidor y viceversa.
>
> Mientras cada lado conozca el formato de los mensajes a mandar al otro lado, pueden mantenerse modulares y separados... Adicionalmente, la separación

> permite a cada componente tener la habilidad de evolucionar independientemente.
>
> En la arquitectura REST, los clientes envían peticiones para extraer o modificar recursos, y los servidores envían respuestas a estas peticiones [7].

Partiendo de la definición proporcionada por Codecademy podemos ver que el uso de una arquitectura **REST** es apropiado para sistemas distribuidos por definición. Solo quedaría definir el tipo de petición y respuestas, y el formato de los mensajes.

Para las peticiones se usa generalmente un verbo **HTTP**, una **cabecera**, una **dirección a un recurso** y un **mensaje opcional**, que será lo usado para seguir el estándar.

Con respecto a los mensajes, usaremos un **Objeto de Transferencia de Datos (DTO)**, el cual es un patrón de diseño arquitectónico de *software*, que facilita la transferencia de datos entre subsistemas, y desacopla el modelo de dominio de la capa de presentación, permitiendo mantener la modularidad e independencia conseguidas con la arquitectura **REST**.

Por último, para las respuestas seguiremos un patrón de respuestas **HTTP**, conformado por un **código de estado**, una **cabecera "*content-type*"** que contiene el tipo de datos en el cuerpo de la respuesta, y el propio **cuerpo de la respuesta** con los datos pertinentes.

Teniendo esto definido, podemos ver cómo resultaría la arquitectura propuesta integrando la arquitectura **REST** para comunicar los servicios y ver un ejemplo de una interacción con la misma.

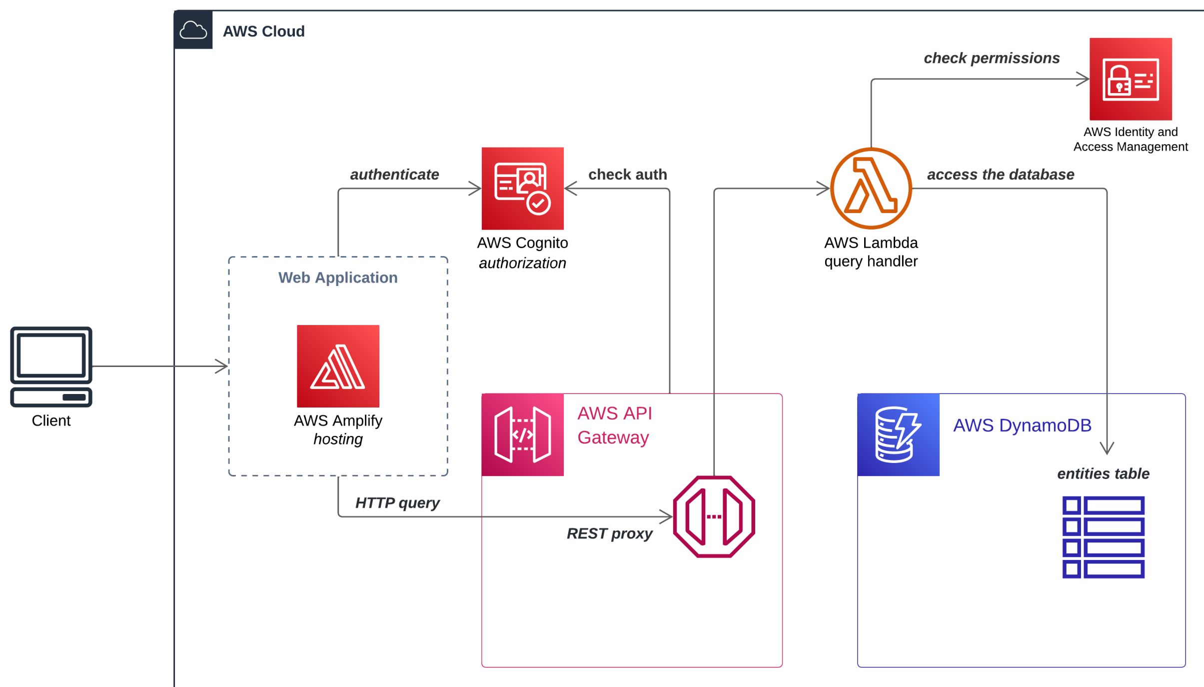

Figura 5.1: Arquitectura Safeguard con componentes **REST** integrados.

1 Al acceder el cliente a la aplicación web, se autentificará con **Amazon Cognito**.

2 La aplicación web enviará una ***query*** **HTTP**, haciendo uso de la arquitectura **REST**, a la cual se añadirá una cabecera de autenticación con el *token* obtenido previamente de **Amazon Cognito**.

3 La **query HTTP** mandada previamente llegará al ***proxy*** **REST** construido con **API Gateway** que validará la cabecera de autenticación con el servicio de **Amazon Cognito**.

4 Tras comprobar que la *query* está autenticada, la redirigirá al *backend*, alojado en el servicio **Lambda**, que se encargará de enviar la petición pertinente a la tabla en **DynamoDB**.

5 Al intentar enviar la petición, el servicio **Lambda** comprobará que la función que se está ejecutando tiene permisos para realizar la operación en **Amazon Identity and Access Management**.

6 Una vez que hayan sido verificados los permisos de la función, la petición será enviada con éxito a la tabla de **DynamoDB**.

5.2.2. Servicios en profundidad

Habiendo visto cómo se comunican los servicios, hay que ver en profundidad cómo actúan los mismos, empezando por **Amplify** y terminando por **DynamoDB**.

Amplify, como hemos visto anteriormente, nos permite alojar, compilar, y asignarle un dominio a una aplicación web.

Cognito, por su parte, nos permite crear grupos de usuarios y gestionarlos, definir los atributos de los usuarios y los requisitos de las contraseñas, implementar verificación multifactor y una amplia variedad de opciones relacionadas con la creación y autenticación de cuentas de usuarios.

API Gateway, en cambio, nos da una mayor libertad. Presenta una variedad de tipos de **API** a implementar, pero en nuestro caso hemos hecho caso de una **API REST** con una integración de *proxy*, así conseguimos crear un *endpoint* único que enmascara el acceso a nuestro servidor, pero que redirige las peticiones a la función **Lambda** sin problemas. Además, para mayor seguridad, nos permite asociar un autorizador que contendrá el **ID** del grupo de usuarios de **Cognito** para verificar la cabecera de autenticación que le llegue en las solicitudes.

Respecto a **Lambda**, almacena el código de un programa y ejecuta una función del mismo cuando recibe una solicitud. Esta función puede desencadenar otras y actuar como un programa normal, pero en nuestro caso actuará como un servidor de una arquitectura **REST** y redirigirá las solicitudes a los *endpoints* correspondientes.

Identity and Access Management o IAM, es un servicio de AWS con integraciones en muchos otros servicios de la misma plataforma. Su principal función es asignar perfiles de permisos al resto de servicios y asegurar que no se interactúe con un servicio para el cual no se tienen permisos.

Finalmente, **DynamoDB** proporciona un entorno donde alojar tablas de datos e interactuar con ellas. Algo similar a **MySQL Workbench**, pero haciendo uso de tablas de datos no relacionales. Además, estas tablas pueden ser accedidas por otros servicios para interactuar con los datos.

5.2.3. Despliegue de los servicios

Por último, una vez visto cómo se comunican los servicios y qué hace cada uno, resta ver cómo desplegarlos y hacerlos funcionar. El despliegue de los servicios se realizará de tres formas:

El código de la aplicación web estará alojado en **GitHub** y mediante integración continua será desplegado y actualizado en **Amplify**.

El *backend*, en cambio, hará uso de otro servicio aparte llamado *AWS Serverless Application Model* (SAM), junto a una plantilla definida en el directorio del proyecto. En esta plantilla se recogerán los servicios y configuraciones de los mismos para ser desplegados y poner en marcha el *backend* (API Gateway, Lambda, IAM).

Los servicios restantes serán **Cognito** y **DynamoDB**, que requerirán una creación manual en la interfaz de servicios de AWS para ponerlos en marcha.

5.3 Tecnología Utilizada

5.3.1. Lenguajes de desarrollo y Frameworks

Por una parte, el desarrollo de *frontend* comúnmente usa lenguajes informáticos como HTML, CSS, JavaScript, TypeScript, etc. Entre las diversas opciones disponibles, la elección vendrá definida por el *framework* usado. Actualmente, los *frameworks* más populares para desarrollo web son React, Angular, Vue o Django.

Para elegir entre una de las tantas opciones, se ha hecho uso de la estimación del coste de aprendizaje de los mismos. Y como resultado, la elección ha sido **Angular** debido al aprendizaje paralelo, realizado en las prácticas de empresa, sobre el *framework*. Esta elección implica que, para el *frontend*, los lenguajes a usar serán HTML, CSS y TypeScript.

Por otra parte, el desarrollo del *backend*, generalmente usa lenguajes de programación como Ruby, Java, Python o C#. En este caso, hay un factor condicionante en las opciones, y es el uso del servicio **AWS Lambda**. Entre las opciones disponibles para usar este servicio se encuentran Java, Go, PowerShell, Node.js, C#, Python y el código Ruby.

Cada una de estas opciones posee amplia documentación para ser usada por el servicio **AWS Lambda**, y realmente no existe un factor condicionante para elegir sobre un lenguaje u otro más allá del conocimiento previo en el mismo, ya que existen *frameworks* para todos los lenguajes disponibles que nos permitirían implementar la arquitectura **REST**.

Teniendo esto en cuenta, la elección para desarrollar el *backend* ha sido **Java** junto con el *framework* **Spring Boot**, debido a que, al igual que con el *framework* de *frontend*, poseía una fuente de conocimiento amplia y gratuita en la empresa de prácticas.

5.3.2. Entorno de desarrollo integrado

En el mundo de los entornos de desarrollo integrados o **IDEs** podemos encontrar una amplia variedad de opciones. La elección entre ellos, generalmente, se rige por el lenguaje de programación utilizado, el sistema operativo, las diferentes funciones de automatización disponibles y la capacidad de personalización de los mismos. Dentro de las diferentes opciones disponibles, las más populares podrían ser VSCode, IntelliJ IDEA o Eclipse.

Para el desarrollo del proyecto, se ha elegido **VSCode** para el *frontend* por su amplia cantidad de extensiones disponibles y facilidad de uso.

Y para el *backend*, la elección estaba entre IntelliJ IDEA y Eclipse. Pese a que ambos presentan opciones de automatización que resultarían muy útiles para trabajar con **Java**, es el uso de **IntelliJ** el que está más extendido en la industria y es más cómodo de usar que Eclipse.

5.3.3. Control de versiones

Para el control de versiones se ha hecho uso de **Git**, y se han alojado los proyectos en **GitHub**. La estructura seguida ha consistido en dos repositorios, uno para el *frontend* y otro para el *backend*.

Como hemos visto en secciones anteriores, para hacer el despliegue del *frontend*, se ha hecho uso de un repositorio en **GitHub**. Y para desplegar el *backend* se ha hecho uso del servicio **AWS SAM**, por lo que ambas partes de la aplicación requerían de un modo de despliegue distinto, lo que justifica su separación en dos repositorios diferentes.

Además, gracias a tener el sistema modularizado y que cada parte esté separada, podemos implementar integración continua en cada una de las partes. El *frontend* será actualizado y desplegado continuamente con cada *push* que se realice al repositorio, gracias a su integración con **Amplify**. El *backend*, por su parte, gracias al servicio **SAM** podemos guardar los cambios con un *push* en su respectivo repositorio y hacer inmediatamente un *deploy* de todo el *backend* con **SAM**.

CAPÍTULO 6
Desarrollo e implementación de la arquitectura

En este capítulo se hará una revisión del proceso de desarrollo y se describirán los diferentes componentes del sistema y cómo es la implementación resultante del sistema, centrándonos en la implementación de cada capa.

6.1 Desarrollo de la arquitectura

El proceso de desarrollo de la solución ha sido un proceso largo y tedioso, y aunque muy enriquecedor, caracterizado principalmente por los problemas y la complejidad de la tecnología.

Inicialmente, el planteamiento del trabajo, como hemos podido ver en secciones anteriores, contaba con una arquitectura y un plan diferentes.

A medida que el proceso de investigación sobre las diferentes plataformas *cloud*, y posteriormente, sobre AWS y sus servicios, avanzaba, nuevos contratiempos aparecían. Esto resultó en una reformulación del plan de trabajo, y en consecuencia, de una nueva estimación de costos.

Se tuvo que crear un proyecto inicial de pruebas (para poder ir investigando los diferentes servicios disponibles y cómo ir encajando cada uno de ellos en la arquitectura) que iba siendo reiniciado constantemente debido a la nueva información que se obtenía.

Durante el desarrollo del *frontend*, hubo algunos problemas relacionados con el aprendizaje de **Angular** y de **Angular Material**, debido a los tutoriales desfasados en internet y a la documentación confusa del propio *framework*. Además, la integración con **Amplify** estuvo dando problemas durante varios días, debido a la compilación esperada por el servicio y la compilación realizada por la última versión de **Angular**.

Respecto al *backend*, surgieron bastantes más problemas que con el *frontend*. Inicialmente, el servicio de **API Gateway** se encargaba de gestionar las redirecciones de las peticiones en función del *path* que tuviesen. El problema con esto es que no permitía tener un *backend* unificado y tenía que estar disperso en varias funciones **Lambda** diferentes. Además, cada nueva adición a los *endpoints*, requería de una configuración manual de varios parámetros en la interfaz que ralentizaba el proceso.

Para solucionar esto, se estuvo buscando soluciones en diversas fuentes, y se encontró, en el **GitHub** de AWS, un repositorio[1] con una guía para integrar **Spring Boot** y **Lambda**, gracias a una dependencia desarrollada por ellos.

Una vez el *backend* estaba funcional, restaba crear una prueba para acceder a la base de datos a través del mismo. Aunque durante las pruebas preliminares usando el software **Postman**, las peticiones HTTP devolvían un estatus 200 OK, al intentar acceder desde el *frontend* resultaba imposible. Tras varios días y mucha investigación, se encontró que actualmente existían dos mecanismos **CORS** en uso actualmente en la aplicación. Uno correspondía al servicio **API Gateway** de AWS y el otro correspondía al propio *framework* de **Spring Boot**. Después de habilitar las cabeceras y configurar ambos, finalmente fue posible acceder al servidor y realizar peticiones a los diferentes *endpoints*.

Finalmente, una vez todo era estable, se recreó la implementación del *backend* haciendo uso del servicio **SAM** para poder desplegarlo de forma continua y agilizar el proceso.

6.2 Implementación de la aplicación web

La aplicación web es el punto de interacción del usuario con el sistema, en esta se tendrán en cuenta las interacciones del usuario y la reacción a las mismas. La implementación de la aplicación web está realizada usando el *framework* Angular. De esta manera, está conformada de componentes; cada uno de estos a su vez, está formado por una plantilla HTML o vista, una clase TypeScript para el comportamiento y un selector de estilos.

6.2.1. Componentes

En Angular, la interfaz de usuario se compone de componentes. Existen algunos predeterminados, como podría ser el caso de los proporcionados por Angular Material, o personalizados, como en el caso de una página entera.

La relación entre los componentes se puede clasificar en uno de los siguientes cuatro tipos (pueden existir diversas relaciones entre dos componentes):

1. **Relación padre-hijo:** un componente sirve como padre y otro como su hijo. Esto permite transferir información mediante *inputs* y la comunicación de eventos mediante *outputs* y emisores de eventos.

2. **Relación hijo-padre:** un componente sirve como hijo y otro como su padre; la dinámica es la misma que la anterior relación pero a la inversa.

3. **Relación de hermanos:** dos componentes que comparten el mismo padre; estos son incapaces de compartir información entre sí directamente, pero pueden compartirla utilizando al padre como intermediario o mediante un servicio.

4. **Sin relación:** los componentes que no tienen relación entre sí de ningún tipo, pueden comunicarse mediante servicios que actúen como intermediarios.

6.2.2. Vistas

Una vista es un conjunto de elementos de interfaz de usuario que se agrupan y trabajan juntos con un fin. En concreto, con el *framework* Angular, las vistas describen la disposición de los componentes en la interfaz del usuario.

[1] https://github.com/aws/serverless-java-container

6.2.3. Comportamiento

Las clases de comportamiento nos permiten describir el comportamiento de la aplicación web ante una interacción por parte del usuario o manejar la información recibida por el servidor. En Angular, son la parte central de cada componente ya que contienen la referencia a la vista y a los estilos, y se encargan de gestionar el comportamiento de cada componente.

6.2.4. Estilos

Los estilos se encargan de dotar de propiedades a los diferentes elementos de la vista, permitiendo ajustar colores, márgenes, posiciones, etc. Con Angular, cada componente tiene su propio archivo de estilos relacionado.

6.2.5. Navegación

La navegación en la página web se realiza mediante el *RouterModule* del *framework* Angular. En este módulo podemos definir las diferentes rutas accesibles de la aplicación web, a qué componente pertenece la ruta y opcionalmente qué *guard* activan.

6.2.6. Guards

Los *guards* son componentes que se pueden generar mediante Angular que contienen un método en el cual podemos definir reglas. Si las reglas se cumplen y devuelve un valor verdadero habilitará el tráfico, en caso contrario, se podrá definir acciones a realizar.

6.2.7. Interceptor HTTP

Los interceptores, al igual que los *guards*, son componentes que se pueden generar con Angular y que podemos enlazar a un cliente HTTP. De esta manera, nos permite definir acciones a realizar cuando se efectúe una solicitud desde el cliente. En el caso de nuestra arquitectura, se usarán para añadir el token obtenido desde AWS Cognito a la cabecera de la solicitud que se enviará desde la aplicación web.

6.2.8. Despliegue

Para el despliegue de la aplicación web, se ha hecho uso de integración continua entre el repositorio de GitHub y el servicio AWS Amplify. Esto nos permite que, con subir los cambios al repositorio, se notifique al servicio de Amplify y comience la nueva implementación.

6.3 Implementación del servidor

El servidor es el encargado de procesar las solicitudes recibidas desde la aplicación web, extraer información de la base de datos, y procesar la información que recibe y extrae para cumplir las solicitudes. Para desarrollar el servidor se ha hecho uso del *framework* Spring Boot y de las dependencias de AWS "aws-serverless-java-container-springboot3"[2]

[2] https://mvnrepository.com/artifact/com.amazonaws.serverless/aws-serverless-java-container-springboot3

y "aws-java-sdk-dynamodb".[3] Estas dos últimas dependencias nos permiten integrar Spring Boot con el servicio AWS Lambda y acceder a la base de datos DynamoDB respectivamente.

6.3.1. Lambda Handler

El Lambda Handler es una clase java que contiene el método que será invocado al recibir una petición el servicio AWS Lambda implementado. En concreto, con la dependencia para integrar Spring Boot, el resultado debería ser algo similar a lo siguiente:

```
1 public class StreamLambdaHandler implements RequestStreamHandler {
2     private static SpringBootLambdaContainerHandler<AwsProxyRequest,
          AwsProxyResponse> handler;
3     static {
4         try {
5             handler = SpringBootLambdaContainerHandler.getAwsProxyHandler(
                  SafeguardbackendApplication.class);
6         } catch (ContainerInitializationException e) {
7             // if we fail here. We re-throw the exception to force another
                  cold start
8             e.printStackTrace();
9             throw new RuntimeException("Could not initialize Spring Boot
                  application", e);
10        }
11    }
12
13    @Override
14    public void handleRequest(InputStream inputStream, OutputStream
          outputStream, Context context)
15            throws IOException {
16        handler.proxyStream(inputStream, outputStream, context);
17    }
18 }
```

Listing 6.1: StreamLambdaHandler class

6.3.2. Controladores

Los controladores son clases de java marcadas con la etiqueta ´´**@RestController**" de Spring Boot en las cuales definiremos los *endpoints* del servidor a los que se podrán hacer solicitudes desde la aplicación web. Asimismo, en cada *endpoint* se definirán los criterios de aceptación de los parámetros recibidos (p.ej. que el ID no sea nulo).

6.3.3. Servicios

Los servicios, al igual que los controladores, son clases de java marcadas con la etiqueta "**@Service**" que se encargan de gestionar las solicitudes recibidas y efectuar las operaciones necesarias en la base de datos con la ayuda del **DynamoDBMapper**, de la dependencia de AWS DynamoDB, que nos permite realizar operaciones **CRUD** (*Create, Read, Update, Delete*).

[3] https://mvnrepository.com/artifact/com.amazonaws/aws-java-sdk-dynamodb

6.3.4. Modelos

Los modelos o entidades son objetos con identidades únicas, y que a su vez son representaciones de las tablas que tenemos en la base de datos y de objetos del mundo real.

6.3.5. Despliegue

Para el despliegue del servidor se hará uso del servicio AWS SAM y de una plantilla donde se indican los diferentes servicios AWS a desplegar. La plantilla a usar puede variar dependiendo de las necesidades de la aplicación, pero una base desde la cual partir es la siguiente:

```
1 AWSTemplateFormatVersion: '2010-09-09'
2 Transform: AWS::Serverless-2016-10-31
3 Description: Safeguard Backend API
4
5 Globals:
6   Api:
7     EndpointConfiguration: REGIONAL
8     Cors:
9       AllowMethods: "'DELETE, GET, HEAD, OPTIONS, PATCH, POST, PUT'"
10      AllowHeaders: "'Content-Type,X-Amz-Date,Authorization,X-Api-Key,X-Amz
          -Security-Token'"
11      AllowOrigin: "'*'"
12
13 Resources:
14   SafeguardBackendApi:
15     Type: AWS::Serverless::Api
16     Properties:
17       StageName: Prod
18       Auth:
19         DefaultAuthorizer: MyCognitoAuth
20         Authorizers:
21           MyCognitoAuth:
22             Type: COGNITO_USER_POOLS
23             UserPoolArn: "YOUR_COGNITO_USER_POOL_ARN"
24   SafeguardBackend:
25     Type: AWS::Serverless::Function
26     Properties:
27       Handler: com.safeguard.safeguardbackend.StreamLambdaHandler::
          handleRequest
28       Runtime: java21
29       CodeUri: .
30       MemorySize: 2048
31       Policies:
32         - AWSLambdaBasicExecutionRole
33         - AmazonDynamoDBFullAccess
34       Timeout: 60
35       Events:
36         RestApiEvent:
37           Type: Api
38           Properties:
39             Path: /{proxy+}
40             Method: ANY
41             RestApiId: !Ref SafeguardBackendApi
42             Auth:
43               Authorizer: MyCognitoAuth
```

Listing 6.2: Template.yml

6.4 Cognito y DynamoDB

Los servicios de AWS, Cognito y DynamoDB requieren una inicialización manual. Esto es debido a que la plantilla anterior sirve principalmente para regenerar el servidor en caso de alguna actualización del mismo y permitir así integración continua, en cambio, los servicios tratados en esta sección no requieren regenerarse una vez se hayan puesto en marcha por primera vez.

6.4.1. Cognito

Para iniciar el servicio de Cognito, necesitamos acceder a la página del servicio en AWS (figura 6.1) y hacer clic en "Crear grupo de usuarios".

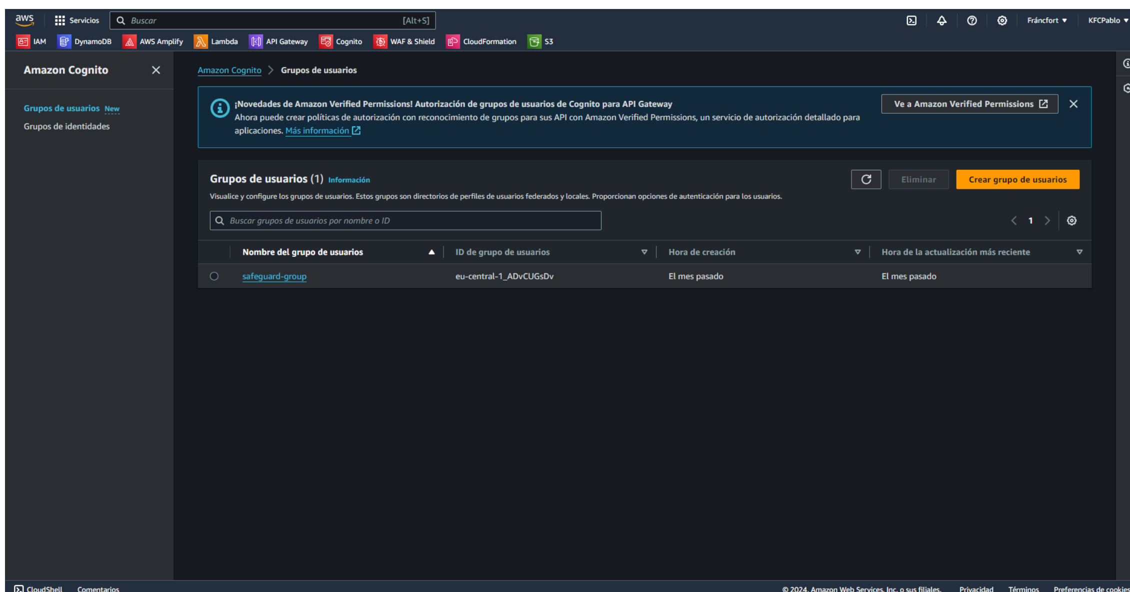

Figura 6.1: Vista principal del servicio AWS.

Una vez hayamos hecho clic, se nos abrirá un diálogo donde podremos seleccionar las opciones que nos interesen de entre la amplia variedad disponible. Para finalizar la creación, elegiremos el nombre del grupo de usuarios. 6.2

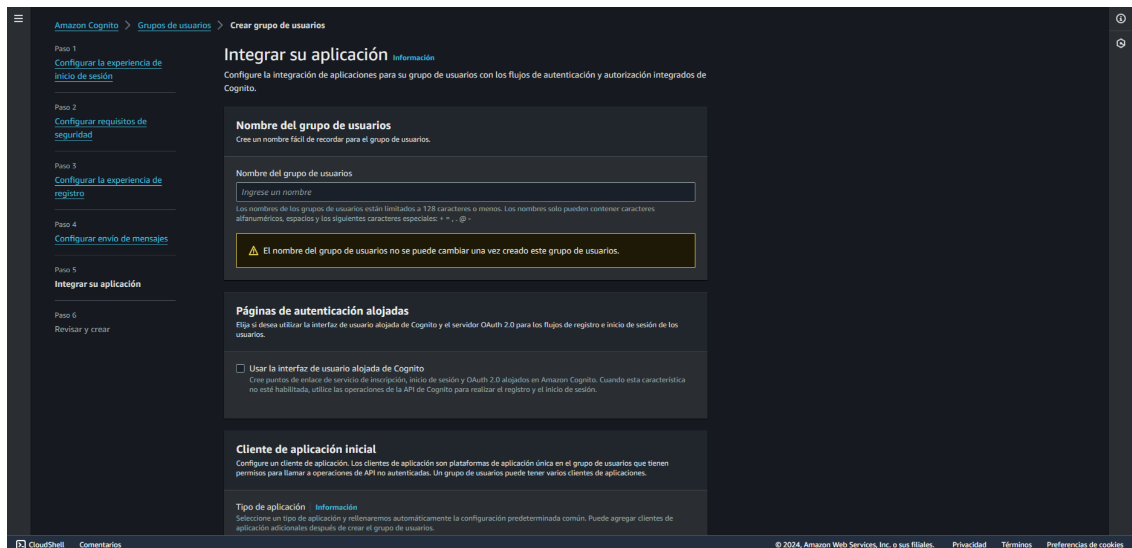

Figura 6.2: Diálogo de creación de un grupo de usuarios.

Una vez tengamos el grupo creado, accederemos al mismo haciendo clic sobre su nombre en la vista principal del servicio, en la figura 6.3, y podremos ver el ID del grupo de usuarios.

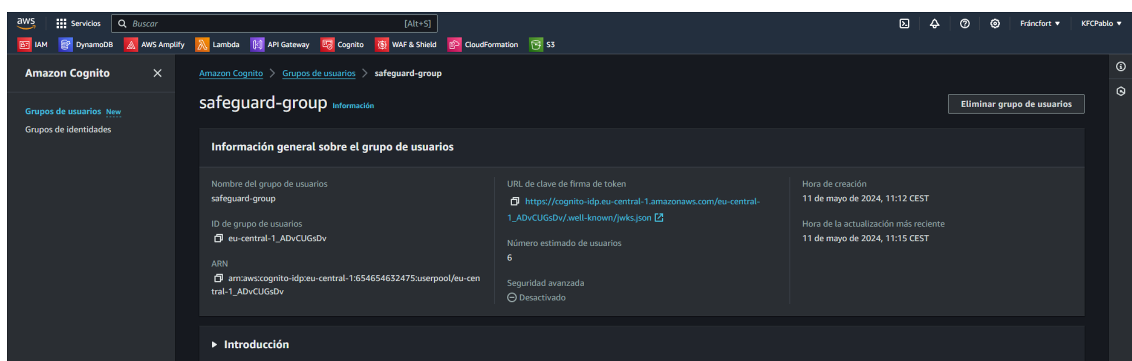

Figura 6.3: Vista del grupo de usuarios Safeguard.

Obtenido el ID del grupo de usuarios, faltaría el ID del cliente de Cognito que se nos ha generado automáticamente durante el diálogo de creación del grupo de usuarios. Para acceder al mismo, haremos clic en el apartado "Integración de aplicaciones" y bajaremos hasta abajo del todo. Una vez aquí, como podemos ver en la figura 6.4, podremos ver el ID de nuestro cliente.

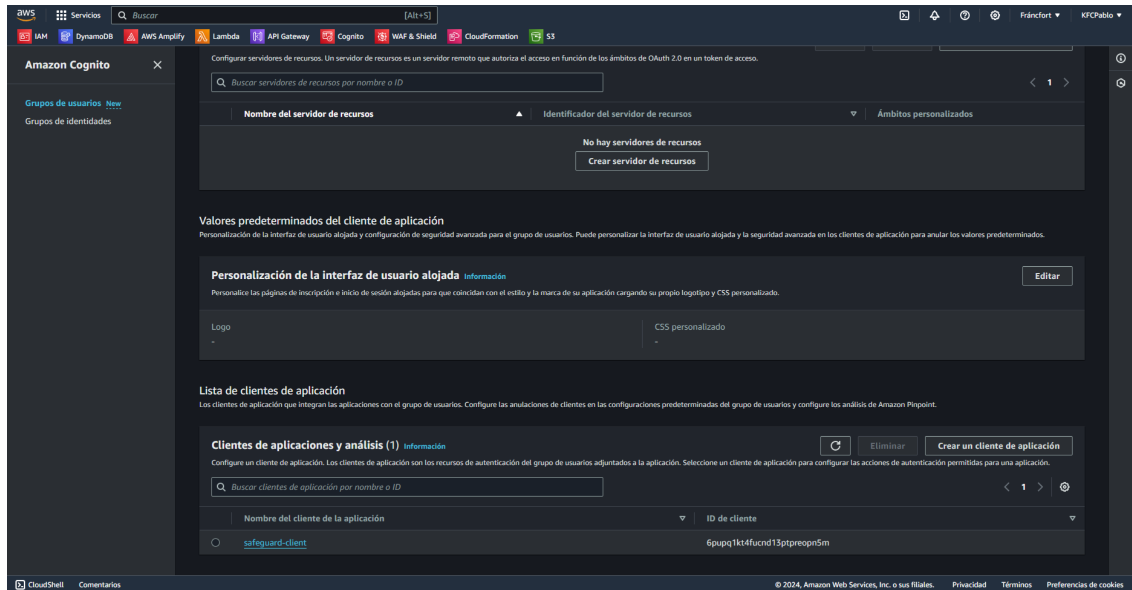

Figura 6.4: Vista del cliente del grupo de usuarios Safeguard.

6.4.2. DynamoDB

Para poner en marcha el servicio de DynamoDB, será necesario ir a la página principal del servicio, la cual será similar a la figura 6.5, y hacer clic en "Crear tabla".

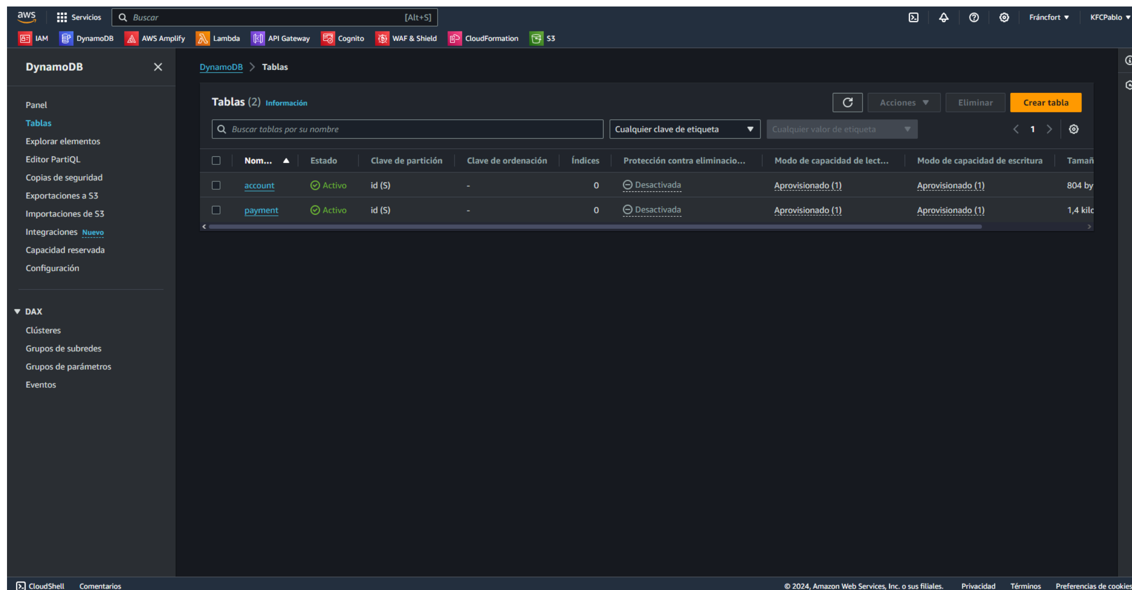

Figura 6.5: Vista principal del servicio DynamoDB.

Una vez hagamos clic, se abrirá un dialogo (figura 6.6) para definir las propiedades de la tabla de la base de datos.

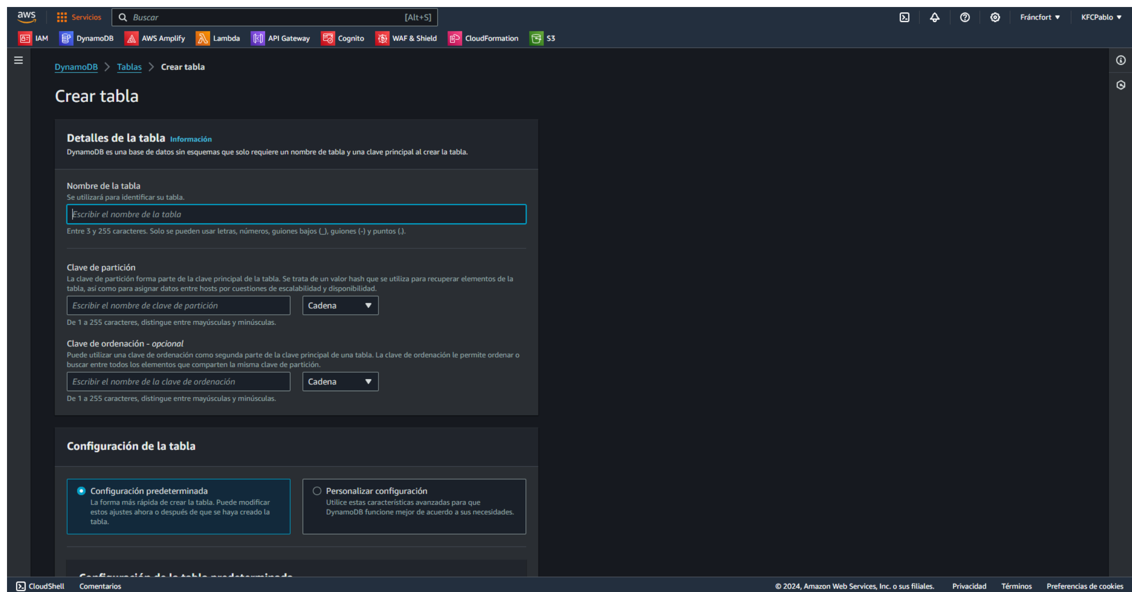

Figura 6.6: Vista del dialogo de crear tablas.

En este diálogo tendremos que definir el nombre de la tabla y la clave de partición obligatoriamente. Una vez hayamos terminado, haremos clic en "Crear tabla" y podremos ver las tablas creadas de igual manera que en la figura 6.5.

CAPÍTULO 7
Pruebas

Las pruebas para la arquitectura se han basado principalmente en pruebas de seguridad, pruebas de integración y pruebas de carga.

7.0.1. Pruebas de seguridad

Para las pruebas de seguridad, también hemos hecho uso del software **Postman**, y además usaremos **Nmap** y haremos una inyección SQL en el login.

Postman

Las pruebas con Postman han consistido en enviar una petición a la API mientras que la API tenía el autentificador asociado de **Cognito**.

En caso de no adjuntar una cabecera "*Authorization*" la respuesta será un estado 401 *Unauthorized*:

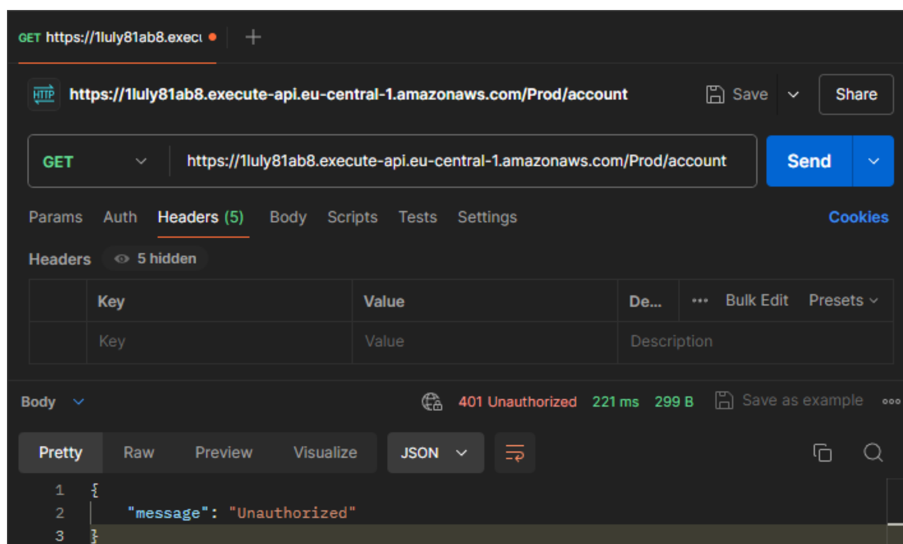

Figura 7.1: Prueba de seguridad: *Authorization* sin cabecera.

En caso de adjuntar la cabecera "*Authorization*" pero no adjuntar el *token*, la respuesta será la misma que en el caso anterior:

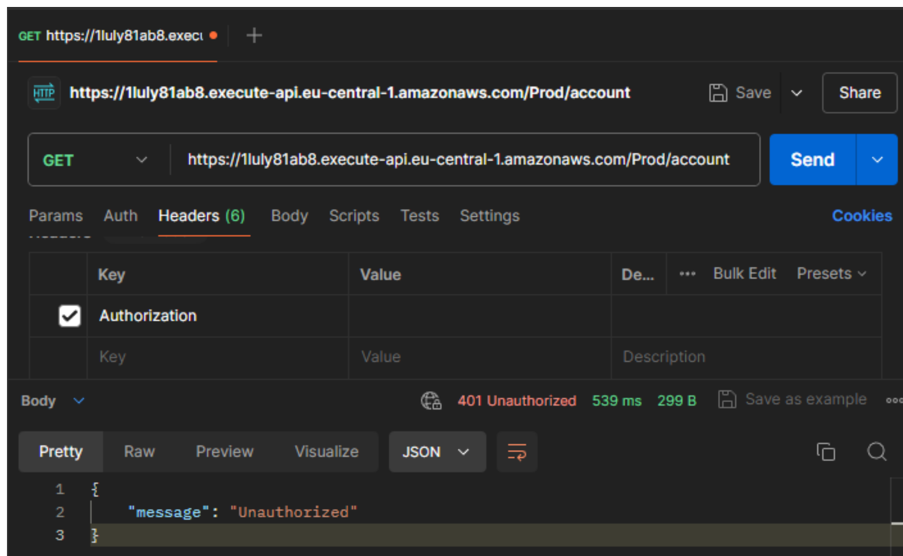

Figura 7.2: Prueba de seguridad: *Authorization* con cabecera y sin *token*.

Por último, en caso de adjuntar la cabecera “*Authorization*” y adjuntar el *token*, la respuesta esperada contendrá la respuesta del *endpoint* llamado:

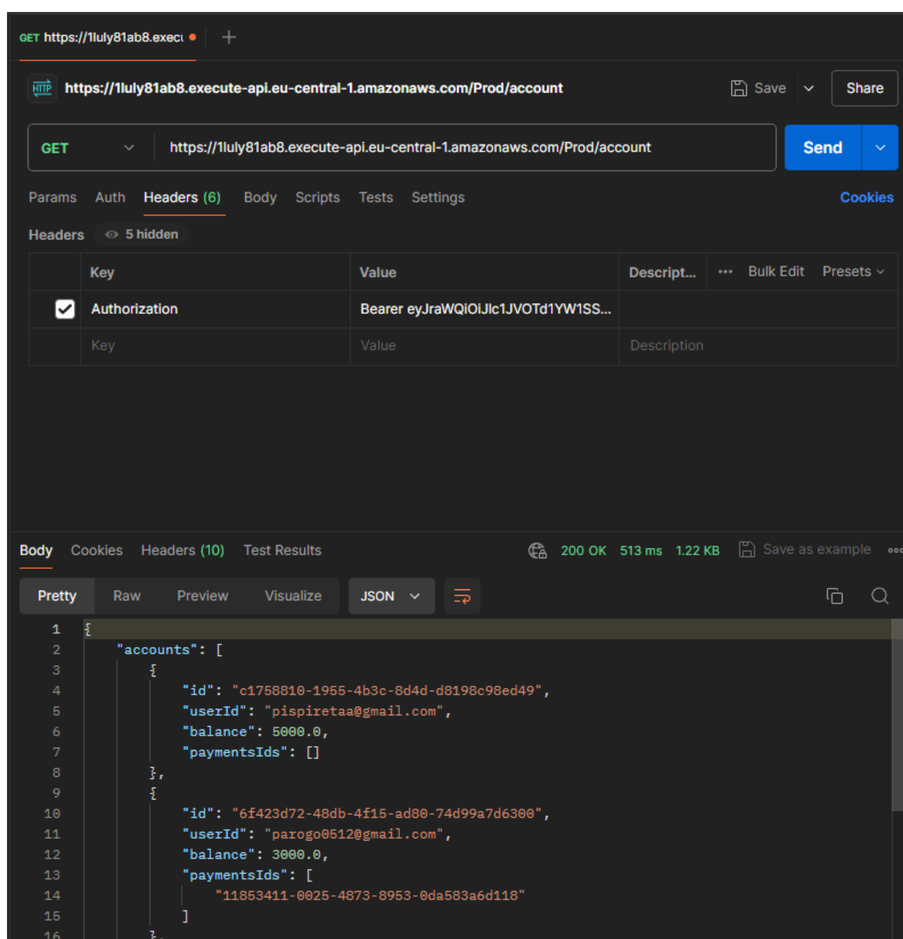

Figura 7.3: Prueba de seguridad: *Authorization* con cabecera y con *token*.

Nmap

Con Nmap analizaremos la aplicación web, en concreto haremos un análisis en busca de vulnerabilidades y lo compararemos con otra página web que sí que tenga vulnerabilidades.

Tras hacer los análisis con Nmap, podemos obtener los *logs* adjuntados en el apéndice A.

En el *log* A.1, el del Nmap a Safeguard, podemos ver que únicamente hay dos puertos abiertos, el puerto 80/tcp y el 443/tcp (línea 17 y 18). Por otra parte, se ha intentado detectar el sistema operativo sobre el que funciona la aplicación web, pero tampoco ha podido (líneas 23 y 24). Además, ha conseguido detectar otras tres IPs para acceder a la aplicación web, estas parecen ser réplicas de la misma IP en varios servidores de AWS (línea 36). Si seguimos viendo, se escanean los puertos 80/tcp y 443/tcp, en el 80 se descubre que es un puerto para el servicio HTTP y que pertenece al servicio Amazon CloudFront (línea 40) y del 443 se descubre también que pertenece a Amazon CloudFront, pero en este caso tiene un cifrado SSL (línea 45). Por último, nos da una dificultad de predecir la secuencia TCP de 254, lo que resulta en una dificultad elevada para ser susceptible de ataques informáticos.

En el *log* A.2 perteneciente al Nmap a Gillweb, podemos ver algo similar a lo comentado en el anterior párrafo, la única diferencia radica en que en este caso todo se encuentra detrás de un *proxy* nginx en vez de Amazon CloudFront y además posee un *firewall* web, lo que le da una dificultad de 262 (línea 68), superior a la de Safeguard por el *firewall* incluido.

En cambio, en el *log* A.3 que es de scoutslasalle214.com, sí que podemos ver algo más interesante y que nos sirve para comparar con una página con menor seguridad. Se han descubierto un total de 12 puertos (líneas 17-28), una cantidad más elevada que la descubierta en las anteriores aplicaciones web. El resto de hallazgos son similares a las anteriores aplicaciones web, por lo que la dificultad en este caso es de 252, inferior a la de Safeguard por la amplia cantidad de puertos abiertos.

Inyección SQL

Para comprobar la seguridad en el login de la aplicación web, haremos una inyección SQL en el formulario de inicio de sesión para intentar sortear la seguridad de la página y obtener un token de autentificación sin tener cuenta.

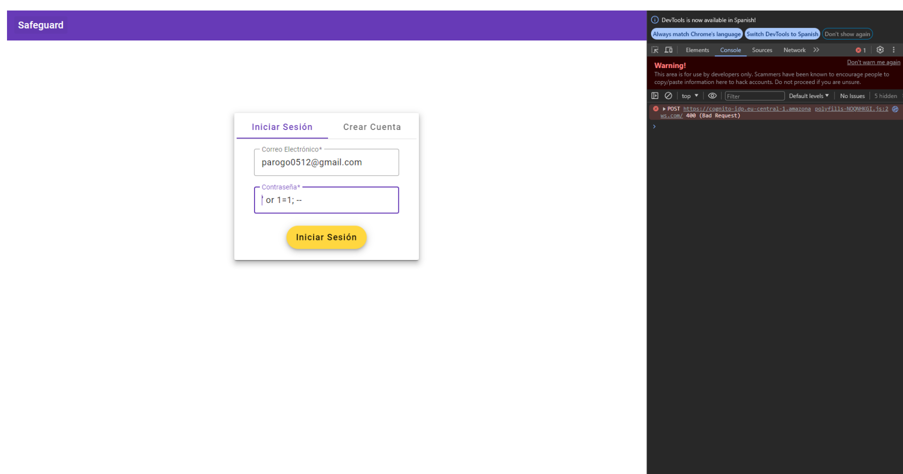

Figura 7.4: Inyección SQL en el login de Safeguard.

En este caso, podemos ver cómo al intentar hacer una inyección SQL, el propio servicio de AWS Cognito devuelve un estado 400 y está protegido ante inyecciones SQL.

7.0.2. Pruebas de integración

Las pruebas de integración nos permiten comprobar si el funcionamiento de varios componentes conectados es el esperado. Para llevar a cabo estas pruebas se ha hecho uso de la aplicación **Postman**. Esta es una aplicación *software* que nos permite, entre otras cosas, efectuar pruebas en APIs y sus diferentes *endpoints*. Esto nos ha permitido comprobar que la API, que actuaba como *proxy* y había sido desplegada con el servicio **API Gateway**, recibía y devolvía las peticiones correctamente.

Además, una vez el funcionamiento de la API estaba comprobado, hemos podido hacer uso de **Postman** para comprobar que el servidor funcionaba correctamente, estableciendo un *endpoint* de prueba e intentando acceder al mismo mediante una *query*.

Por último, también se ha comprobado el acceso a la base de datos mediante el *endpoint* y la API. Para ello, se ha hecho uso de la API y el *endpoint* anteriormente creados, y se ha añadido el *mapper* necesario para acceder a los datos de la tabla de **DynamoDB**. El resultado de esta última comprobación, que nos sirve a su vez para comprobar los otros dos casos, es el siguiente:

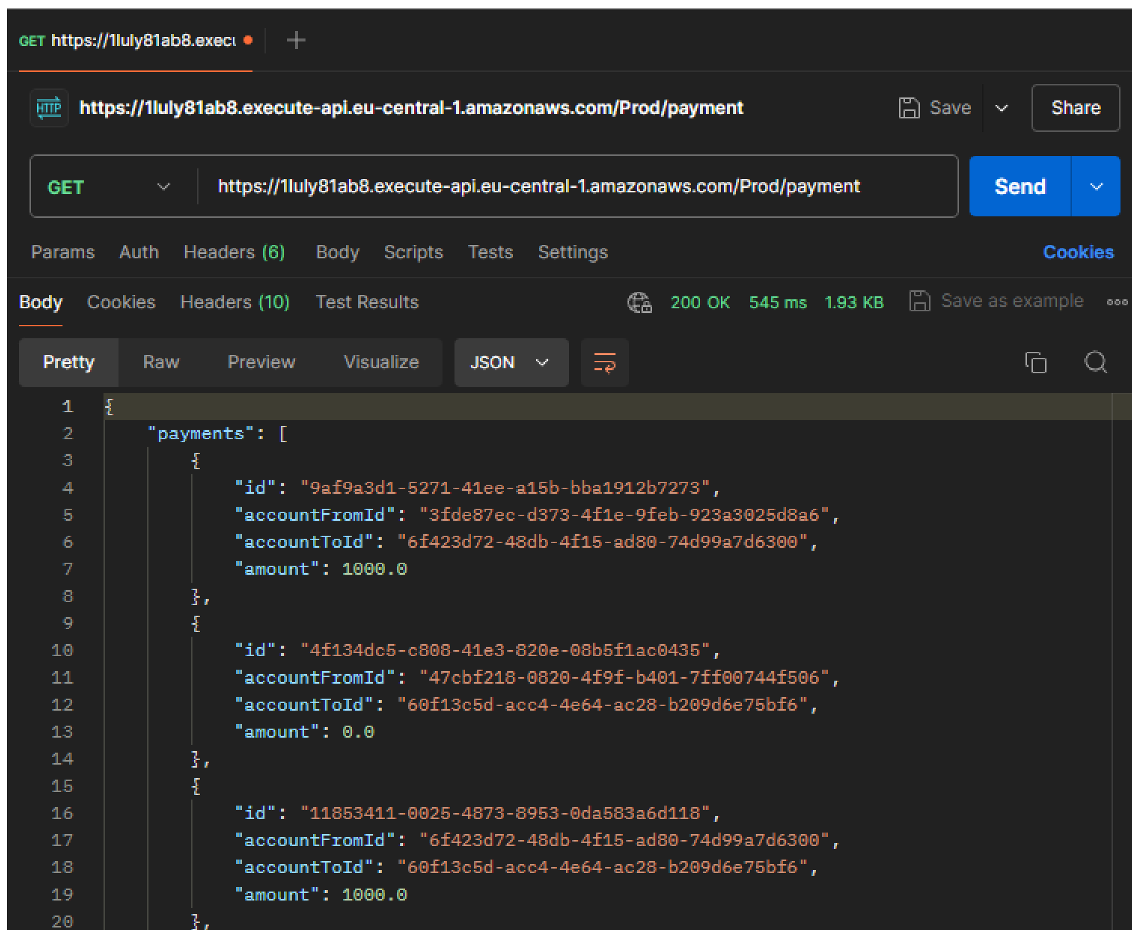

Figura 7.5: Prueba de integración: Acceso a **DynamoDB** a través del *endpoint* y la API.

En la figura 7.5 podemos ver que al hacer una *query* a la API con el *path* del *endpoint* “*payment*” nos devuelve el listado de pagos almacenados en **DynamoDB**.

7.0.3. Pruebas de carga

Para realizar pruebas de carga sobre la arquitectura, se ha revisado un documento [8] proporcionado en la documentación de AWS que describe diversas herramientas disponibles para tal cometido. Dentro del mismo se encuentra Apache JMeter.

La necesidad de realizar pruebas de carga con Apache JMeter es debida al inevitable caso de que ocurra un pico de usuarios en la aplicación. Con estas pruebas de carga podremos comprobar el comportamiento de la arquitectura ante estos supuestos y buscar el límite actual con el plan gratuito.

Para simular este pico de usuarios vamos a generar peticiones en un lapso de 60 segundos y generar una gráfica con los tiempos de respuesta. Para ello, se creará un *Thread Group*, que es un grupo de hilos (usuarios) cuya configuración para las peticiones variará según los valores mostrados en la tabla 7.1.

Cantidad de hilos	Periodo de arranque (s)
5000	60
4000	60
3000	60
2000	60
1000	60
500	60
250	60

Tabla 7.1: Valores de pruebas de carga.

Tras realizar las pruebas de carga con los diferentes valores previos, hemos obtenido las siguientes gráficas presentes en las figuras 7.6, 7.7, 7.8, 7.9, 7.10, 7.11 y 7.12.

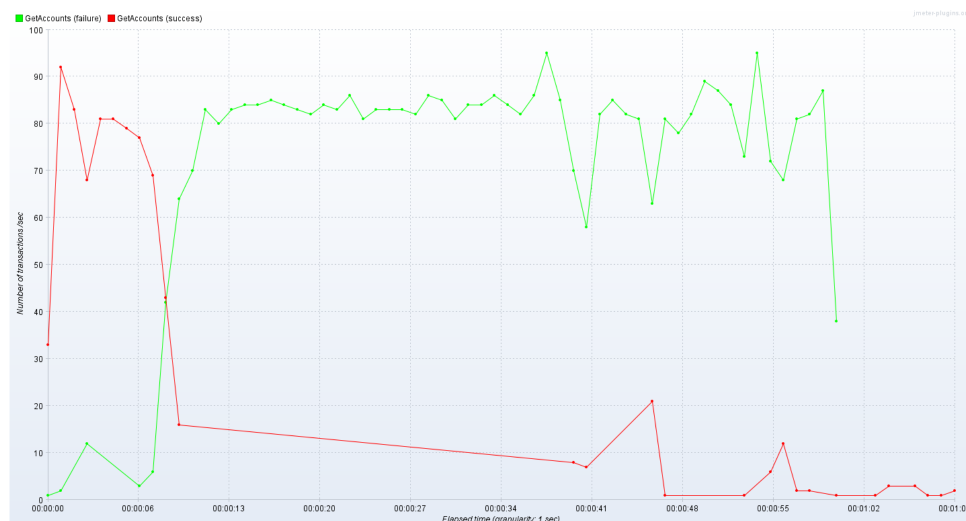

Figura 7.6: Prueba de carga con 5000 hilos en 60 segundos.

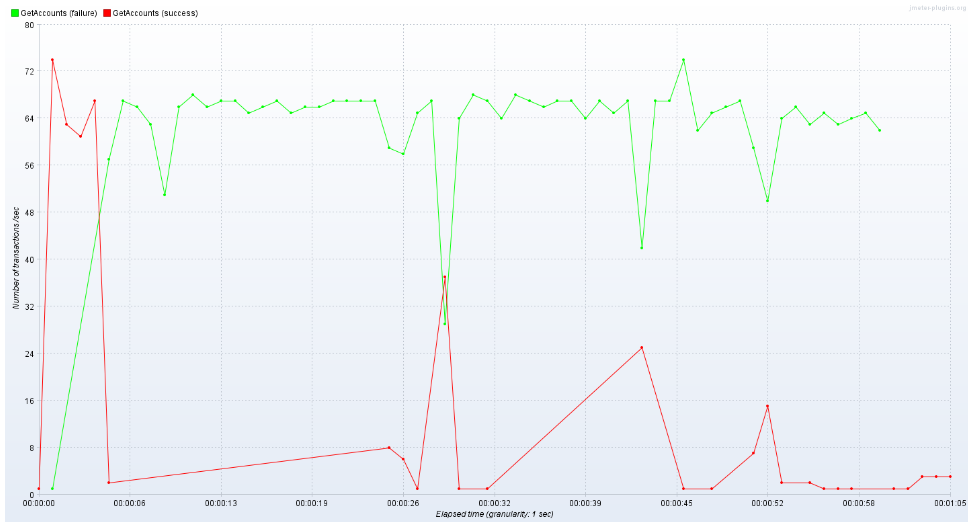

Figura 7.7: Prueba de carga con 4000 hilos en 60 segundos.

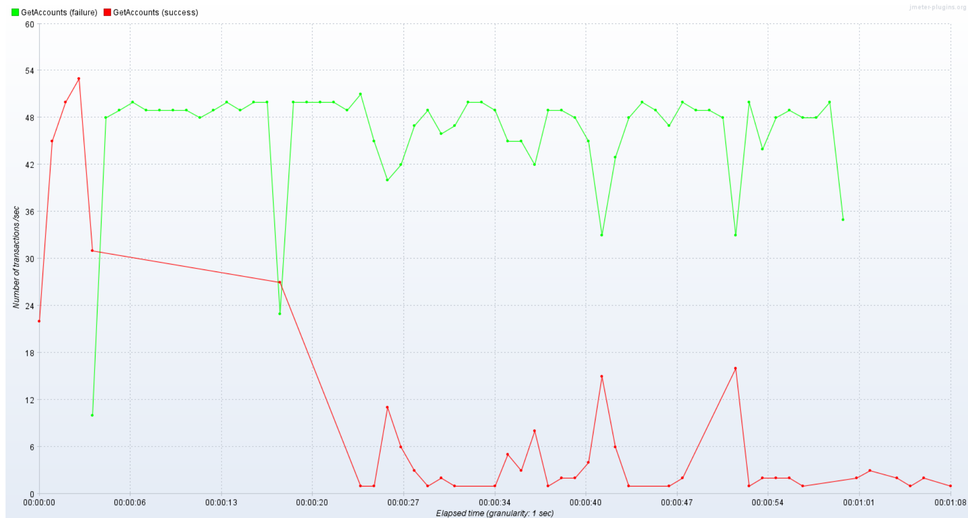

Figura 7.8: Prueba de carga con 3000 hilos en 60 segundos.

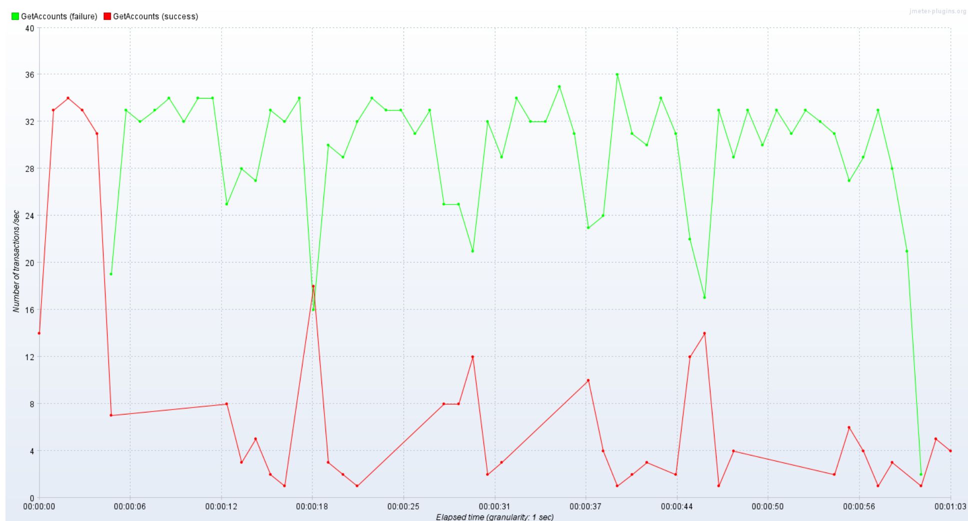

Figura 7.9: Prueba de carga con 2000 hilos en 60 segundos.

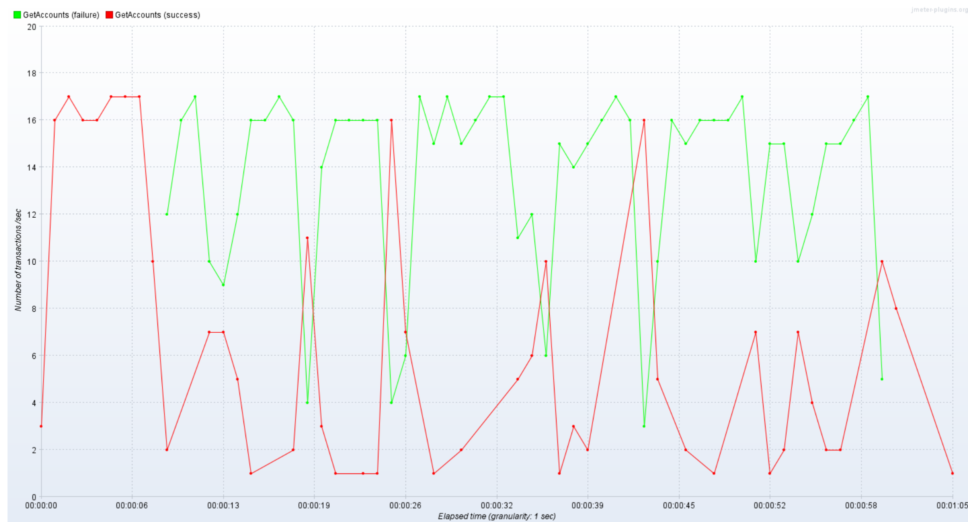

Figura 7.10: Prueba de carga con 1000 hilos en 60 segundos.

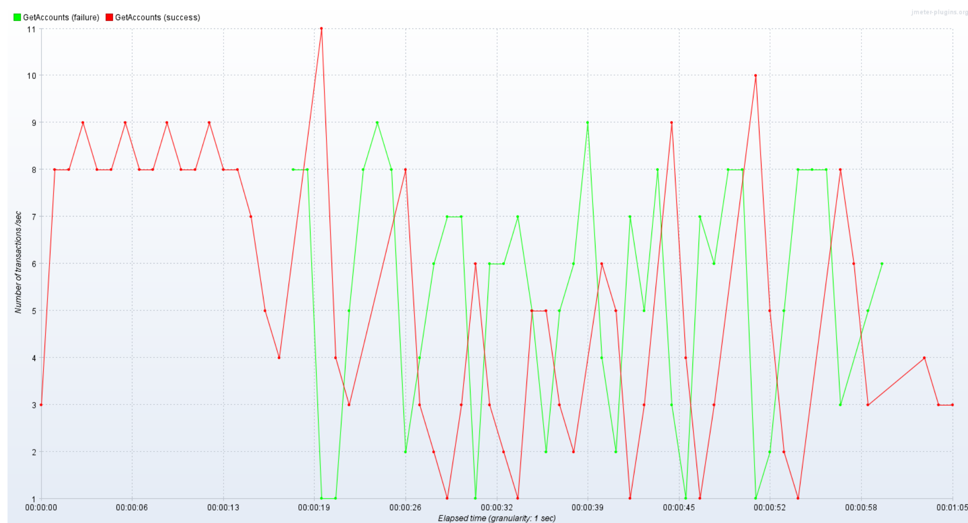

Figura 7.11: Prueba de carga con 500 hilos en 60 segundos.

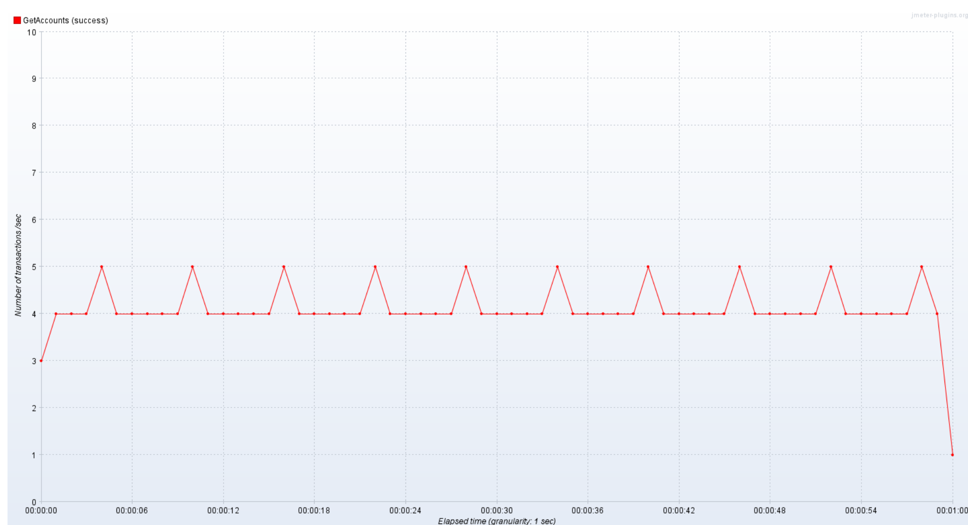

Figura 7.12: Prueba de carga con 250 hilos en 60 segundos.

Tras ver las diferentes tablas, podemos ver que desde los 5000 hilos hasta los 2000, las gráficas obtenidas son bastante similares; en ellas podemos ver cómo una gran cantidad de las solicitudes enviadas al servidor fracasan. Esto es debido a que el servidor no puede procesar tantas solicitudes; el servidor autoescala hasta el máximo del plan gratuito.

Por otra parte, al llegar a los 1000 y 500 hilos, podemos ver que la gráfica empieza a cambiar y la cantidad de hilos que tienen éxito y fallan es aproximadamente la mitad.

Por último, con 250 hilos podemos ver que todas las solicitudes tienen éxito, por lo que el plan gratuito de AWS nos permite escalar el servidor hasta procesar alrededor de 250 solicitudes en un lapso de 60 segundos.

CAPÍTULO 8

Prueba de concepto

Para efectuar la prueba de concepto de la arquitectura Safeguard se ha seguido el supuesto de uso por parte de un grupo scout para tratamiento de datos sensibles. En concreto, se tratarán dos casos de uso en el apartado de la tesorería del grupo.

8.1 Consulta de información

El objetivo de este caso de uso es consultar información de una cuenta haciendo uso de la arquitectura desarrollada y enviando la petición a través de los diferentes componentes de la misma.

La consulta de información tendrá lugar en la vista 8.1 en la cual se podrá ver el estado de la cuenta y los pagos efectuados.

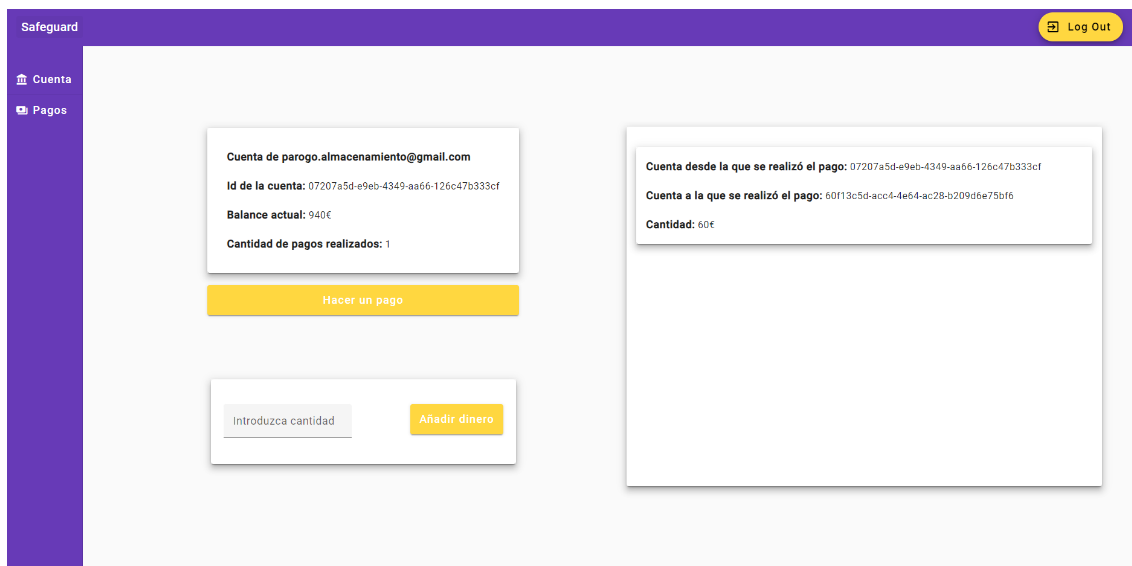

Figura 8.1: Prueba de concepto: Vista de la cuenta.

Para obtener esta información, al iniciar la vista se hace uso del servicio "AccountService" para hacer una solicitud HTTP desde la aplicación web al servidor. En concreto, el método empleado para realizarla es el visto en el *listing* 8.1.

```
1 // @RequestMapping(path = "/account/accountByUserId/{userId}")
2 public retrieveAccountByUserId(userId: string): Observable<
      RetrieveAccountResponse> {
3     return this.httpClient.get<RetrieveAccountResponse>(
4     `${this.APIURL}/account/accountByUserId/${userId}`
5     );
6 }
```

Listing 8.1: retrieveAccountByUserId method in account.service.ts

Tras efectuar ese método, la aplicación web alojada en AWS Amplify enviará la solicitud que será interceptada por un HTTPInterceptor 8.2 del *framework* Angular que añadirá la cabecera de autenticación y enviará la solicitud al destino original.

```
1 export const apiRestInterceptor: HttpInterceptorFn = (req, next) => {
2     const authService = inject(AuthService);
3 
4     let authReq;
5     const authToken = authService.getJwtIdToken();
6 
7     if (authToken != null) {
8         authReq = req.clone({
9             setHeaders: {
10                Authorization: 'Bearer ' + authToken,
11            },
12        });
13    }
14
15    return next(authReq);
16 };
```

Listing 8.2: apiRestInterceptor method in api-rest.interceptor.ts

La solicitud llegará al servicio API Gateway que comprobará la cabecera de autenticación con el autorizador de AWS Cognito. Tras verificarlo, redirigirá la petición al servicio AWS Lambda que llamará al *handler* visto previamente en el *listing* 6.1. Este *handler* iniciará el *framework* Spring Boot que se encargará de gestionar la solicitud que le ha llegado y redirigirla al *endpoint* descrito en el *listing* 8.3 y a su 8.4.

```
1 @RestController
2 @EnableWebMvc
3 public interface AccountController {
4     ...
5     @RequestMapping(path = "/account/accountByUserId/{userId}", method =
          RequestMethod.GET)
6     ResponseEntity<RetrieveAccountResponse> retrieveAccountByUserId(
          @PathVariable("userId") String userId);
7     ...
8 }
```

Listing 8.3: retrieveAccountByUserId endpoint in AccountController.java

```java
@Service
public class AccountControllerImpl implements AccountController {

    @Autowired
    AccountService accountService;
    ...
    @Override
    public ResponseEntity<RetrieveAccountResponse> retrieveAccountByUserId(
        String userId) {
        if (userId == null) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "No
                account userID");
        }

        var list = accountService.retrieveAccounts();
        var filteredAccount = list.stream().filter(account -> {
            if(account.getUserId().equals(userId)) {
                return true;
            }
            return false;
        }).toList();

        var response = new RetrieveAccountResponse(filteredAccount.get(0));

        return ResponseEntity.ok(response);
    }
    ...
}
```

Listing 8.4: retrieveAccountByUserId endpoint implementation in AccountControllerImpl.java

La implementación del *endpoint* hará uso del servicio "AccountService" para extraer las cuentas de la base de datos. Para ello, se llamará al método "retrieveAccounts" descrito en el *listing* 8.5, que hará uso del DynamoDBMapper para consultar la base de datos DynamoDB de la cuenta en la que está desplegado el servidor.

```java
// AccountService.java
public interface AccountService {
    ...
    @Transactional(readOnly = true)
    List<Account> retrieveAccounts();
    ...
}

// AccountServiceImpl.java
@Service
public class AccountServiceImpl implements AccountService {

    @Autowired
    DynamoDBMapper dynamoDBMapper;

    ...
    @Override
    public List<Account> retrieveAccounts() {
        return dynamoDBMapper.scan(Account.class, new
            DynamoDBScanExpression());
    }
    ...
}
```

Listing 8.5: retrieveAccounts method in AccountService

Tras enviar la petición a la base de datos con el DynamoDBMapper, el servicio AWS Lambda comprueba que tengamos los permisos necesarios para leer en el servicio AWS IAM y tras verificarlo nos permite realizar con éxito la petición a DynamoDB. El resultado obtenido será devuelto al servicio Lambda y a continuación a través de la cadena de llamadas, empezando por el DynamoDBMapper. Al llegar al controlador, Spring Boot generará la respuesta del servidor que será enviada al cliente, el cual mostrará los datos vistos en la vista 8.1.

8.2 Modificación de información

El objetivo de este caso de uso es realizar un pago modificando la información almacenada en la base de datos haciendo uso de los diferentes componentes de la arquitectura desplegada.

Este caso de uso se desarrollará en la vista 8.2, donde se podrán efectuar pagos seleccionando la cuenta a la que realizarlo.

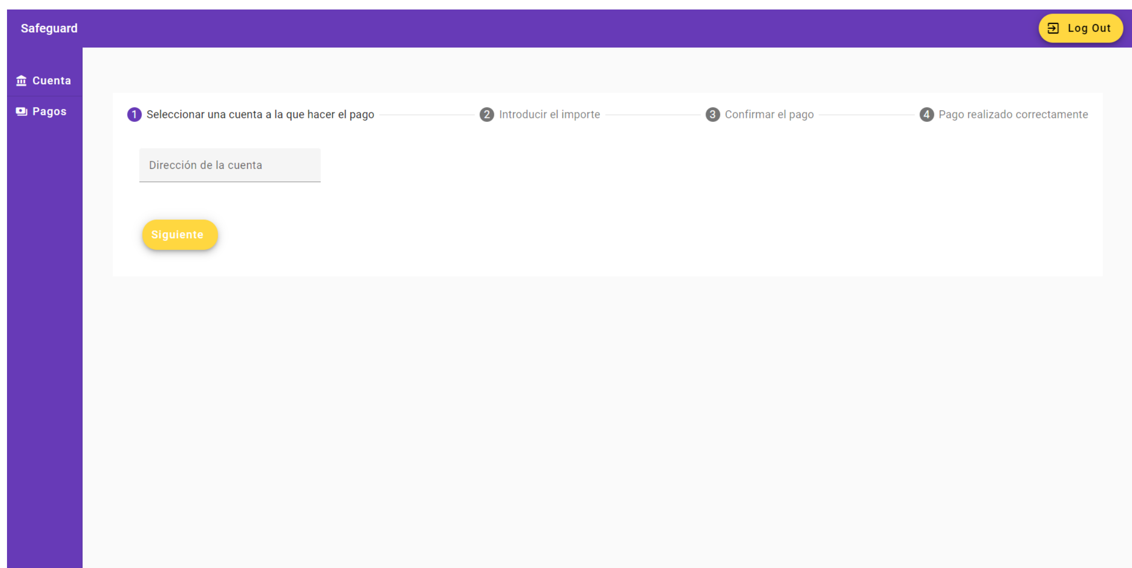

Figura 8.2: Prueba de concepto: Vista de los pagos.

Primero se efectuará una petición al servidor para obtener la lista de cuentas presentes en el sistema, tras esto se seleccionará la cuenta y se introducirá la cantidad del pago a efectuar. Al igual que en el anterior caso de uso, la aplicación web enviará un mensaje a la API del servicio API Gateway que comprobará el *token* de autenticación y redirigirá la petición a la función Lambda del servicio AWS Lambda. Esta función comprobará tener los permisos para escribir en DynamoDB con el servicio AWS IAM y tras verificarlo escribirá en la base de datos y completará la solicitud devolviendo un estado 200 OK.

CAPÍTULO 9
Conclusiones

Este TFG ha supuesto el diseño y desarrollo de una arquitectura *software*, fruto de la recopilación de la información dispersa en la red, que sirve como base para otros proyectos.

Al principio se hizo un análisis de los diferentes tipos de sistemas existentes y sus variantes, tras esto se analizaron las funcionalidades de las dos aplicaciones presentes en el mercado objetivo y posteriormente su arquitectura.

Los problemas encontrados a lo largo del desarrollo del trabajo han sido fruto de la inexperiencia con las tecnologías usadas, y han sido errores necesarios de cometer para ir adquiriendo experiencia y aprender más sobre las mismas.

Hablando de aprendizaje, el desarrollo de este trabajo ha supuesto aprender sobre una amplia cantidad de tecnologías y conocimientos. Entre ellos, TypeScript, Angular, CSS, Spring Boot, AWS, APIs o arquitecturas REST. La amplia mayoría de ellos han sido dominados hasta el nivel de poder trabajar con ellos de forma cómoda.

La excepción ha sido AWS, ya que presenta una abrumadora variedad de opciones que aprender y tener en cuenta, y pese a que podamos desarrollar una arquitectura sin problemas, sigue habiendo mucho que aprender y descubrir.

Concluyendo, respecto a los objetivos, podemos asegurar que se han cumplido satisfactoriamente con el trabajo realizado.

9.1 Relación del trabajo desarrollado con los estudios cursados

En el proceso de desarrollo de este TFG, se han puesto en práctica los conocimientos adquiridos en las asignaturas de la Universidad Politécnica de Valencia y en la Universidad Politécnica de Poznan. Entre todas las asignaturas cursadas, las que han contribuido principalmente han sido las siguientes:

- UPV (Universidad Politécnica de Valencia)
  - **PSW (Proceso del Software):** En esta asignatura se impartieron los conocimientos necesarios para saber realizar desarrollo ágil de productos software y gracias al proyecto realizado también se aprendió a tratar con bases de datos no relacionales.
  - **IPC (Interfaces Persona Computador):** Esta asignatura mostró las bases necesarias para desarrollar interfaces siguiendo buenas prácticas y criterios de usabilidad.

- PUT (Poznan University of Technology)
  - **SAV (Software Architecture and Verification):** Esta asignatura ha sido en gran parte la base para realizar este TFG, en ella se enseñó sobre arquitecturas *software*, cómo diseñarlas y cómo analizarlas.
  - **Pre-diploma seminar:** Enseñó las bases para realizar una investigación científica para preparar el estado del arte de un trabajo científico-técnico.
  - **Scientific & Technical writing:** Impartió los conocimientos necesarios para redactar documentos científicos y técnicos siguiendo buenas prácticas.

Por otra parte, las competencias transversales han tenido una gran importancia, y las relacionadas con el desarrollo del proyecto han sido las siguientes:

- **Análisis y resolución de problemas:** En el desarrollo del proyecto han surgido una amplia variedad de problemas, pero también se han ido resolviendo de manera satisfactoria.
- **Diseño y proyecto:** El diseño ha sido la tarea fundamental de este proyecto, se ha diseñado una arquitectura siguiendo una metodología de trabajo para proyectos.
- **Responsabilidad ética, medioambiental y profesional:** Siendo un trabajo en el que se tratan temas de educandos y en el cual influyen leyes de protección de datos, existe un requisito de responsabilidad ética y profesional. Además, relacionando el trabajo con los ODS tratamos la responsabilidad medioambiental.
- **Aprendizaje permanente:** El aprendizaje sobre las diversas tecnologías nuevas necesarias para desarrollar este proyecto ha sido necesario debido al desconocimiento de las mismas, pero el conocimiento anterior ha servido de base para aprenderlas.
- **Planificación y gestión del tiempo:** Se ha realizado una planificación del trabajo, y una gestión del tiempo continua durante todo el proceso de desarrollo.

9.2 Trabajos futuros

Los objetivos planteados para el trabajo se han alcanzado de forma exitosa, pero siempre hay margen de mejora en cualquier trabajo. De cara al uso de forma extendida de la arquitectura como base de aplicaciones, se podrían implementar las siguientes mejoras:

- **Implementar OAuth:** La arquitectura cuenta con un login básico por propósitos de seguridad. Una medida para aumentar la seguridad de la aplicación sería usar cuentas verificadas de otras plataformas para iniciar sesión.
- **Multi-factor Authentication (MFA):** Con el fin de incrementar la seguridad de la aplicación, otra medida que podría ser útil sería implementar un MFA que asegurara la identidad de la persona.
- **Implementar un WAF gratuito:** Tal y como hemos visto anteriormente, un WAF sería una adición muy útil a la arquitectura, pero el proporcionado por AWS resultaría en gastos mensuales, por ello añadir una alternativa gratuita o desarrollarla podría ser una adición a tener en cuenta en el futuro.

CAPÍTULO 10
Glosario

Cloud Computing: Tecnología que permite acceder a recursos informáticos a través de Internet.

Proxy: Servidor intermediario que actúa como intermediario para las solicitudes de los clientes.

Arquitectura: Estructura fundamental de un sistema.

Arquitectura Software: Es el diseño de alto nivel de un sistema de software que abarca la estructura de sus componentes, sus relaciones, y los principios que guían su organización. [9]

Arquitectura Cloud: La arquitectura de nube es un elemento clave de la compilación en la nube. Se refiere al diseño y conecta todos los componentes y tecnologías necesarios para la computación en la nube. [10]

Escalabilidad horizontal: La escalabilidad horizontal se enfoca en añadir más máquinas para distribuir la carga e incrementar el almacenamiento o potencia de procesamiento global. [11]

Escalabilidad vertical: La escalabilidad vertical se enfoca en añadir más recursos o más potencia de procesamiento a una única máquina. [11]

Bibliografía

[1] Amazon Web Services. ¿Cuál es la diferencia entre la arquitectura monolítica y la de microservicios? [en línea]. Consultado en: https://aws.amazon.com/es/compare/the-difference-between-monolithic-and-microservices-architecture/

[2] Atlassian. ¿Qué es un sistema distribuido? [en línea]. Consultado en: https://www.atlassian.com/es/microservices/microservices-architecture/distributed-architecture

[3] Scott P. Pluralsight. AWS vs Azure vs GCP: comparación de los 3 grandes proveedores de cloud [AWS vs Azure vs GCP: The big 3 cloud providers compared] [en línea]. Consultado en: https://www.pluralsight.com/resources/blog/cloud/aws-vs-azure-vs-gcp-the-big-3-cloud-providers-compared

[4] Amazon Web Services. ¿Qué es AWS? [en línea]. Consultado en: https://aws.amazon.com/es/what-is-aws/

[5] ISO 25000. NORMAS ISO 25000 [en línea]. Consultado en: https://www.iso25000.com/index.php/normas-iso-25000

[6] Boletín Oficial del Estado (BOE). BOE-A-2018-16673 Ley Orgánica 3/2018, de 5 de diciembre, de Protección de Datos Personales y garantía de los derechos digitales [en línea]. Consultado en: https://www.boe.es/buscar/act.php?id=BOE-A-2018-16673

[7] Codecademy. ¿Qué es REST? [What is REST?] [en línea]. Consultado en: https://www.codecademy.com/article/what-is-rest

[8] AWS Aplicaciones de pruebas de carga. AWS Guía prescriptiva. [en línea]. Consultado en: https://docs.aws.amazon.com/es_es/prescriptive-guidance/latest/load-testing/load-testing.pdf

[9] Jose Francisco Ojeda Montoya. Arquitectura de Software: Fundamentos, Características, Usos y Ejemplos [en línea]. Consultado en: https://www.linkedin.com/pulse/arquitectura-de-software-fundamentos-caracterÃ■sticas-y-ojeda-montoya/

[10] Google Cloud platform. ¿Qué es la arquitectura de nube? [en línea]. Consultado en: https://cloud.google.com/learn/what-is-cloud-architecture?hl=es-419

[11] MongoDB. Una guía al escalado horizontal vs vertical. [A Guide to Horizontal vs Vertical Scaling]. Consultado en: https://www.mongodb.com/resources/basics/horizontal-vs-vertical-scaling

[12] Naciones Unidas. La Agenda para el Desarrollo Sostenible. [en línea]. Consultado en: https://www.un.org/sustainabledevelopment/es/development-agenda/

=1

APÉNDICE A
Logs Nmap

```
1  Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-25 20:49 BST
2  NSE: Loaded 151 scripts for scanning.
3  NSE: Script Pre-scanning.
4  Initiating NSE at 20:49
5  Completed NSE at 20:49, 0.00s elapsed
6  Initiating NSE at 20:49
7  Completed NSE at 20:49, 0.00s elapsed
8  Initiating NSE at 20:49
9  Completed NSE at 20:49, 0.00s elapsed
10 Initiating Ping Scan at 20:49
11 Scanning main.d1d8uibhlji936.amplifyapp.com (13.224.132.45) [4 ports]
12 Completed Ping Scan at 20:49, 0.01s elapsed (1 total hosts)
13 Initiating Parallel DNS resolution of 1 host. at 20:49
14 Completed Parallel DNS resolution of 1 host. at 20:49, 0.01s elapsed
15 Initiating SYN Stealth Scan at 20:49
16 Scanning main.d1d8uibhlji936.amplifyapp.com (13.224.132.45) [1001 ports
       ]
17 Discovered open port 80/tcp on 13.224.132.45
18 Discovered open port 443/tcp on 13.224.132.45
19 Completed SYN Stealth Scan at 20:49, 4.56s elapsed (1001 total ports)
20 Initiating Service scan at 20:49
21 Scanning 2 services on main.d1d8uibhlji936.amplifyapp.com
       (13.224.132.45)
22 Completed Service scan at 20:50, 12.11s elapsed (2 services on 1 host)
23 Initiating OS detection (try #1) against main.d1d8uibhlji936.amplifyapp
       .com (13.224.132.45)
24 Retrying OS detection (try #2) against main.d1d8uibhlji936.amplifyapp.
       com (13.224.132.45)
25 Initiating Traceroute at 20:50
26 Completed Traceroute at 20:50, 0.01s elapsed
27 NSE: Script scanning 13.224.132.45.
28 Initiating NSE at 20:50
29 Completed NSE at 20:50, 2.05s elapsed
30 Initiating NSE at 20:50
31 Completed NSE at 20:50, 0.30s elapsed
32 Initiating NSE at 20:50
33 Completed NSE at 20:50, 0.00s elapsed
34 Nmap scan report for main.d1d8uibhlji936.amplifyapp.com (13.224.132.45)
35 Host is up (0.0045s latency).
36 Other addresses for main.d1d8uibhlji936.amplifyapp.com (not scanned):
       13.224.132.73 13.224.132.90 13.224.132.96
37 rDNS record for 13.224.132.45: server-13-224-132-45.lhr3.r.cloudfront.
       net
38 Not shown: 999 filtered ports
39 PORT    STATE SERVICE  VERSION
40 80/tcp  open  http     Amazon CloudFront httpd
41 | http-methods:
```

```
42  |_  Supported Methods: GET HEAD POST OPTIONS
43  |_http-server-header: CloudFront
44  |_http-title: Did not follow redirect to https://main.d1d8uibhlji936.
       amplifyapp.com/
45  443/tcp open  ssl/http Amazon CloudFront httpd
46  |_http-favicon: Unknown favicon MD5: 05BCFE9A02B93E1C5A5DA14BFDA8C41F
47  | http-methods:
48  |_  Supported Methods: GET HEAD OPTIONS
49  | http-server-header:
50  |   AmazonS3
51  |_  CloudFront
52  |_http-title: safeguard-frontend
53  | ssl-cert: Subject: commonName=*.d1d8uibhlji936.amplifyapp.com
54  | Subject Alternative Name: DNS:*.d1d8uibhlji936.amplifyapp.com, DNS:
       d1d8uibhlji936.amplifyapp.com
55  | Issuer: commonName=Amazon RSA 2048 M02/organizationName=Amazon/
       countryName=US
56  | Public Key type: rsa
57  | Public Key bits: 2048
58  | Signature Algorithm: sha256WithRSAEncryption
59  | Not valid before: 2024-04-23T00:00:00
60  | Not valid after:  2025-05-22T23:59:59
61  | MD5:   5f3e d269 c68b 2621 097a 0329 161b 806a
62  |_SHA-1: 9490 524b 7598 0f6f 99c7 212b d32b ea7b 1c0f 7445
63  Warning: OSScan results may be unreliable because we could not find at
       least 1 open and 1 closed port
64  Device type: WAP
65  Running (JUST GUESSING): Linksys embedded (92%)
66  OS CPE: cpe:/h:linksys:befw11s4
67  Aggressive OS guesses: Linksys BEFW11S4 WAP (92%)
68  No exact OS matches for host (test conditions non-ideal).
69  Uptime guess: 0.000 days (since Tue Jun 25 20:50:06 2024)
70  Network Distance: 1 hop
71  TCP Sequence Prediction: Difficulty=254 (Good luck!)
72  IP ID Sequence Generation: All zeros
73
74  TRACEROUTE (using port 80/tcp)
75  HOP RTT     ADDRESS
76  1   2.07 ms server-13-224-132-45.lhr3.r.cloudfront.net (13.224.132.45)
77
78  NSE: Script Post-scanning.
79  Initiating NSE at 20:50
80  Completed NSE at 20:50, 0.00s elapsed
81  Initiating NSE at 20:50
82  Completed NSE at 20:50, 0.00s elapsed
83  Initiating NSE at 20:50
84  Completed NSE at 20:50, 0.00s elapsed
85  Read data files from: /usr/bin/../share/nmap
86  OS and Service detection performed. Please report any incorrect results
        at https://nmap.org/submit/ .
87  Nmap done: 1 IP address (1 host up) scanned in 23.73 seconds
88             Raw packets sent: 2072 (95.752KB) | Rcvd: 39 (1.824KB)
```

Listing A.1: Nmap log de Safeguard

```
1  Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-25 20:46 BST
2  NSE: Loaded 151 scripts for scanning.
3  NSE: Script Pre-scanning.
4  Initiating NSE at 20:46
5  Completed NSE at 20:46, 0.00s elapsed
6  Initiating NSE at 20:46
7  Completed NSE at 20:46, 0.00s elapsed
8  Initiating NSE at 20:46
9  Completed NSE at 20:46, 0.00s elapsed
10 Initiating Ping Scan at 20:46
11 Scanning gillweb.es (217.160.0.113) [4 ports]
12 Completed Ping Scan at 20:46, 0.03s elapsed (1 total hosts)
13 Initiating Parallel DNS resolution of 1 host. at 20:46
14 Completed Parallel DNS resolution of 1 host. at 20:46, 0.07s elapsed
15 Initiating SYN Stealth Scan at 20:46
16 Scanning gillweb.es (217.160.0.113) [1001 ports]
17 Discovered open port 443/tcp on 217.160.0.113
18 Discovered open port 80/tcp on 217.160.0.113
19 Completed SYN Stealth Scan at 20:46, 5.65s elapsed (1001 total ports)
20 Initiating Service scan at 20:46
21 Scanning 2 services on gillweb.es (217.160.0.113)
22 Completed Service scan at 20:46, 12.35s elapsed (2 services on 1 host)
23 Initiating OS detection (try #1) against gillweb.es (217.160.0.113)
24 Retrying OS detection (try #2) against gillweb.es (217.160.0.113)
25 Initiating Traceroute at 20:46
26 Completed Traceroute at 20:46, 0.01s elapsed
27 NSE: Script scanning 217.160.0.113.
28 Initiating NSE at 20:46
29 Completed NSE at 20:46, 5.10s elapsed
30 Initiating NSE at 20:46
31 Completed NSE at 20:46, 1.16s elapsed
32 Initiating NSE at 20:46
33 Completed NSE at 20:46, 0.00s elapsed
34 Nmap scan report for gillweb.es (217.160.0.113)
35 Host is up (0.011s latency).
36 Other addresses for gillweb.es (not scanned): 2001:8d8:100f:f000::228
37 rDNS record for 217.160.0.113: 217-160-0-113.elastic-ssl.ui-r.com
38 Not shown: 999 filtered ports
39 PORT    STATE SERVICE  VERSION
40 80/tcp  open  http     nginx
41 | http-methods:
42 |_  Supported Methods: GET HEAD POST OPTIONS
43 |_http-server-header: Apache
44 |_http-title: Did not follow redirect to https://gillweb.es/
45 443/tcp open  ssl/http nginx
46 | http-methods:
47 |_  Supported Methods: GET POST OPTIONS HEAD
48 |_http-server-header: Apache
49 |_http-title: GillWeb Scouts
50 | ssl-cert: Subject: commonName=*.gillweb.es
51 | Subject Alternative Name: DNS:*.gillweb.es, DNS:gillweb.es
52 | Issuer: commonName=Encryption Everywhere DV TLS CA - G2/
     organizationName=DigiCert Inc/countryName=US
53 | Public Key type: rsa
54 | Public Key bits: 2048
55 | Signature Algorithm: sha256WithRSAEncryption
56 | Not valid before: 2024-06-21T00:00:00
57 | Not valid after:  2025-07-04T23:59:59
58 | MD5:   4701 d537 bf06 c4ce 54f3 66fc 1458 5b3b
59 |_SHA-1: fd93 c23b 37a4 61fe b796 db64 f014 5961 8e17 3fe6
60 Warning: OSScan results may be unreliable because we could not find at
     least 1 open and 1 closed port
61 Device type: WAP|router|general purpose|load balancer|storage-misc|
     firewall
```

```
62  Running (JUST GUESSING): Linksys embedded (91%), Synology embedded
       (88%), Linux 2.6.X (88%), F5 Networks TMOS 11.1.X (88%), Ubiquiti
       embedded (88%), Netgear embedded (87%), Palo Alto embedded (86%)
63  OS CPE: cpe:/h:linksys:befw11s4 cpe:/h:synology:rt1900ac cpe:/o:linux:
       linux_kernel:2.6.32 cpe:/o:f5:tmos:11.1 cpe:/o:linux:linux_kernel
       :2.6 cpe:/h:netgear:readynas_3200 cpe:/h:paloalto:pa-500
64  Aggressive OS guesses: Linksys BEFW11S4 WAP (91%), Synology RT1900ac
       router (88%), Linux 2.6.32 (88%), F5 3600 LTM load balancer (88%),
       Ubiquiti WAP (Linux 2.6.32) (88%), Netgear ReadyNAS 3200 NAS device
       (Linux 2.6) (87%), Palo Alto PA-500 firewall (86%)
65  No exact OS matches for host (test conditions non-ideal).
66  Uptime guess: 24.188 days (since Sat Jun  1 16:16:31 2024)
67  Network Distance: 1 hop
68  TCP Sequence Prediction: Difficulty=262 (Good luck!)
69  IP ID Sequence Generation: All zeros
70
71  TRACEROUTE (using port 80/tcp)
72  HOP RTT     ADDRESS
73  1   2.39 ms 217-160-0-113.elastic-ssl.ui-r.com (217.160.0.113)
74
75  NSE: Script Post-scanning.
76  Initiating NSE at 20:46
77  Completed NSE at 20:46, 0.00s elapsed
78  Initiating NSE at 20:46
79  Completed NSE at 20:46, 0.00s elapsed
80  Initiating NSE at 20:46
81  Completed NSE at 20:46, 0.00s elapsed
82  Read data files from: /usr/bin/../share/nmap
83  OS and Service detection performed. Please report any incorrect results
        at https://nmap.org/submit/ .
84  Nmap done: 1 IP address (1 host up) scanned in 29.19 seconds
85             Raw packets sent: 2074 (95.832KB) | Rcvd: 41 (1.904KB)
```

Listing A.2: Nmap log de Gillweb

```
1   Starting Nmap 7.80 ( https://nmap.org ) at 2024-06-25 20:46 BST
2   NSE: Loaded 151 scripts for scanning.
3   NSE: Script Pre-scanning.
4   Initiating NSE at 20:46
5   Completed NSE at 20:46, 0.00s elapsed
6   Initiating NSE at 20:46
7   Completed NSE at 20:46, 0.00s elapsed
8   Initiating NSE at 20:46
9   Completed NSE at 20:46, 0.00s elapsed
10  Initiating Ping Scan at 20:46
11  Scanning www.scoutslasalle214.com (151.80.184.149) [4 ports]
12  Completed Ping Scan at 20:46, 0.04s elapsed (1 total hosts)
13  Initiating Parallel DNS resolution of 1 host. at 20:46
14  Completed Parallel DNS resolution of 1 host. at 20:46, 0.00s elapsed
15  Initiating SYN Stealth Scan at 20:46
16  Scanning www.scoutslasalle214.com (151.80.184.149) [1001 ports]
17  Discovered open port 443/tcp on 151.80.184.149
18  Discovered open port 25/tcp on 151.80.184.149
19  Discovered open port 110/tcp on 151.80.184.149
20  Discovered open port 21/tcp on 151.80.184.149
21  Discovered open port 993/tcp on 151.80.184.149
22  Discovered open port 53/tcp on 151.80.184.149
23  Discovered open port 143/tcp on 151.80.184.149
24  Discovered open port 80/tcp on 151.80.184.149
25  Discovered open port 587/tcp on 151.80.184.149
26  Discovered open port 995/tcp on 151.80.184.149
27  Discovered open port 26/tcp on 151.80.184.149
28  Discovered open port 465/tcp on 151.80.184.149
29  Completed SYN Stealth Scan at 20:46, 4.47s elapsed (1001 total ports)
30  Initiating Service scan at 20:46
31  Scanning 12 services on www.scoutslasalle214.com (151.80.184.149)
32  Completed Service scan at 20:49, 164.36s elapsed (12 services on 1 host
        )
33  Initiating OS detection (try #1) against www.scoutslasalle214.com
        (151.80.184.149)
34  Retrying OS detection (try #2) against www.scoutslasalle214.com
        (151.80.184.149)
35  Initiating Traceroute at 20:49
36  Completed Traceroute at 20:49, 0.01s elapsed
37  NSE: Script scanning 151.80.184.149.
38  Initiating NSE at 20:49
39  Completed NSE at 20:49, 27.86s elapsed
40  Initiating NSE at 20:49
41  Completed NSE at 20:53, 187.52s elapsed
42  Initiating NSE at 20:53
43  Completed NSE at 20:53, 0.00s elapsed
44  Nmap scan report for www.scoutslasalle214.com (151.80.184.149)
45  Host is up (0.0083s latency).
46  rDNS record for 151.80.184.149: ns1.dominioanonimo.com
47  Not shown: 989 filtered ports
48  PORT    STATE SERVICE  VERSION
49  21/tcp  open  ftp      Pure-FTPd
50  25/tcp  open  smtp?
51  |_smtp-commands: Couldn't establish connection on port 25
52  26/tcp  open  smtp     Exim smtpd 4.96.2
53  |_smtp-commands: Couldn't establish connection on port 26
54  53/tcp  open  domain?
55  | dns-nsid:
56  |   NSID: servidor.dominioanonimo.com (7365727669646
        f722e646f6d696e696f616e6f6e696d6f2e636f6d)
57  |_  id.server: servidor.dominioanonimo.com
58  80/tcp  open  http     nginx
59  |_http-favicon: Unknown favicon MD5: E56DB5E02094AAB63CC76213AB59F668
60  | http-methods:
```

```
61  |_  Supported Methods: GET HEAD POST OPTIONS
62  |_http-server-header: imunify360-webshield/1.21
63  |_http-title: One moment, please...
64  |_http-trane-info: Problem with XML parsing of /evox/about
65  110/tcp open  pop3     Dovecot pop3d
66  143/tcp open  imap     Dovecot imapd
67  |_imap-capabilities: CAPABILITY
68  443/tcp open  ssl/http nginx
69  |_http-favicon: Unknown favicon MD5: FCDDF88A7F45256B4B2753974E781DB5
70  | http-methods:
71  |_  Supported Methods: GET HEAD POST OPTIONS
72  |_http-server-header: imunify360-webshield/1.21
73  |_http-title: 400 The plain HTTP request was sent to HTTPS port
74  |_http-trane-info: Problem with XML parsing of /evox/about
75  | ssl-cert: Subject: commonName=scoutslasalle214.com
76  | Subject Alternative Name: DNS:*.scoutslasalle214.com, DNS:
      scoutslasalle214.com
77  | Issuer: commonName=R11/organizationName=Let's Encrypt/countryName=US
78  | Public Key type: rsa
79  | Public Key bits: 2048
80  | Signature Algorithm: sha256WithRSAEncryption
81  | Not valid before: 2024-06-17T13:01:09
82  | Not valid after:  2024-09-15T13:01:08
83  | MD5:   faf0 1aeb 72ef 6cbf 6ce9 76af a12c e79f
84  |_SHA-1: ce1c f212 6e3d 1983 53c8 c2df ffdf d8db 517f 2c19
85  |_ssl-date: TLS randomness does not represent time
86  | tls-alpn:
87  |   h2
88  |_  http/1.1
89  465/tcp open  ssl/smtp Exim smtpd 4.96.2
90  |_smtp-commands: Couldn't establish connection on port 465
91  587/tcp open  smtp     Exim smtpd 4.96.2
92  | smtp-commands: servidor.dominioanonimo.com Hello www.scoutslasalle214
      .com [146.191.32.49], SIZE 52428800, 8BITMIME, PIPELINING,
      PIPECONNECT, AUTH PLAIN LOGIN, STARTTLS, HELP,
93  |_ Commands supported: AUTH STARTTLS HELO EHLO MAIL RCPT DATA BDAT NOOP
       QUIT RSET HELP
94  993/tcp open  imaps?
95  995/tcp open  pop3s?
96  Warning: OSScan results may be unreliable because we could not find at
      least 1 open and 1 closed port
97  Device type: WAP|router|general purpose|storage-misc|load balancer|
      firewall
98  Running (JUST GUESSING): Linksys embedded (91%), Synology embedded
      (88%), Linux 2.6.X (88%), Ubiquiti embedded (88%), Netgear embedded
      (87%), F5 Networks TMOS 11.1.X (87%), Palo Alto embedded (86%)
99  OS CPE: cpe:/h:linksys:befw11s4 cpe:/h:synology:rt1900ac cpe:/o:linux:
      linux_kernel:2.6.32 cpe:/o:linux:linux_kernel:2.6 cpe:/h:netgear:
      readynas_3200 cpe:/o:f5:tmos:11.1 cpe:/h:paloalto:pa-500
100 Aggressive OS guesses: Linksys BEFW11S4 WAP (91%), Synology RT1900ac
      router (88%), Linux 2.6.32 (88%), Ubiquiti WAP (Linux 2.6.32) (88%),
       Netgear ReadyNAS 3200 NAS device (Linux 2.6) (87%), F5 3600 LTM
      load balancer (87%), Palo Alto PA-500 firewall (86%)
101 No exact OS matches for host (test conditions non-ideal).
102 Uptime guess: 7.898 days (since Mon Jun 17 23:19:53 2024)
103 Network Distance: 1 hop
104 TCP Sequence Prediction: Difficulty=252 (Good luck!)
105 IP ID Sequence Generation: All zeros
106 Service Info: Host: servidor.dominioanonimo.com
107
108 TRACEROUTE (using port 80/tcp)
109 HOP RTT     ADDRESS
110 1   5.03 ms ns1.dominioanonimo.com (151.80.184.149)
111
```

```
112    NSE: Script Post-scanning.
113    Initiating NSE at 20:53
114    Completed NSE at 20:53, 0.00s elapsed
115    Initiating NSE at 20:53
116    Completed NSE at 20:53, 0.00s elapsed
117    Initiating NSE at 20:53
118    Completed NSE at 20:53, 0.00s elapsed
119    Read data files from: /usr/bin/../share/nmap
120    OS and Service detection performed. Please report any incorrect results
            at https://nmap.org/submit/ .
121    Nmap done: 1 IP address (1 host up) scanned in 388.87 seconds
122            Raw packets sent: 2062 (95.312KB) | Rcvd: 49 (2.264KB)
```

Listing A.3: Nmap log de scoutslasalle214

APÉNDICE B
ODS

Según la página de las Naciones Unidas, "los Objetivos de Desarrollo Sostenible (ODS) constituyen un llamamiento universal a la acción para poner fin a la pobreza, proteger el planeta y mejorar las vidas y las perspectivas de las personas en todo el mundo. En 2015, todos los Estados Miembros de las Naciones Unidas aprobaron 17 Objetivos como parte de la Agenda 2030 para el Desarrollo Sostenible, en la cual se establece un plan para alcanzar los Objetivos en 15 años."[12]

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**		X		
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Tabla B.1: Relación con los ODS.

Tal y como se puede ver en la tabla B.1, el proyecto desarrollado tiene relación con algunos de los ODS. En concreto, guarda relación con el ODS 9 (Industria, innovación e infraestructuras) y con el ODS 12 (Producción y consumo responsables).

- **ODS 9. Industria, innovación e infraestructuras:** El proyecto busca modernizar e innovar en las soluciones existentes actualmente en el mercado, dando lugar a una nueva infraestructura sobre la que basar el desarrollo de nuevas aplicaciones.

- **ODS 12. Producción y consumo responsables:** Al desarrollar una arquitectura autoescalable, se hará uso únicamente de los recursos necesarios y no se malgastarán los que sobren.