UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIC DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Dpto. de Sistemas Informáticos y Computación

# Diseño y Despliegue de una Arquitectura Web Serverless para una Institución Religiosa

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones / Cloud and High-Performance Computing

AUTOR/A: Sabogal Cespedes, Giovanni

Tutor/a: Moltó Martínez, Germán

Director/a Experimental: Calatrava Arroyo, Amanda

CURSO ACADÉMICO: 2023/2024

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DSIIC

DEPARTAMENT DE SISTEMES INFORMÀTICS I COMPUTACIÓ

# Diseño y Despliegue de una Arquitectura Web Serverless para Institución religiosa

Trabajo Fin de Máster

Máster Universitario en Computación en la Nube y de Altas Prestaciones

**Autor**: Giovanni Sabogal Cespedes

**Tutor**: Germán Moltó Martínez

**Directora experimental**: Amanda Calatrava Arroyo

Curso 2023-2024

# Resumen

El objetivo de este trabajo final de máster es diseñar y desplegar una arquitectura web basada en el paradigma *serverless*, utilizando los servicios del proveedor *cloud* Amazon Web Services (AWS), para satisfacer las necesidades tecnológicas de la Iglesia Bautista Reformada. Esta arquitectura web proporciona una plataforma ágil, escalable y eficiente en cuanto a costes que permitirá a este tipo de congregaciones darse a conocer, ofrecer servicios en línea y comunicarse con su comunidad.

El diseño de la arquitectura se basa en el uso de servicios cómo AWS Lambda, Amazon Amplify, Amazon CloudFront y API Gateway. Cómo servicios principales, AWS Lambda permite ejecutar código sin necesidad de provisionar ni administrar servidores, lo que garantiza una escalabilidad automática y una alta disponibilidad de la arquitectura diseñada, mientras que Amplify se utiliza para el almacenamiento seguro y escalable de contenido estático en el Frontend, apoyado del servicio de S3 para el almacenamiento de imágenes y ficheros. También ha sido necesario emplear un servicio de base de datos, cómo es Amazon DynamoDB, para proporcionar un *Backend* a la arquitectura web. Finalmente, este trabajo evalúa la arquitectura diseñada mediante pruebas de carga sobre la misma, para las que se utilizan *frameworks* y librerías de código abierto diseñadas para este fin.

**Palabras clave:** Computación sin servidor, arquitectura web, AWS, Computación en la Nube.

# Abstract

The objective of this master's thesis is to design and deploy a web architecture based on the serverless paradigm, using the services of the cloud provider Amazon Web Services (AWS), to meet the technological needs of the Reformed Baptist Church. This web architecture provides an agile, scalable, and cost-efficient platform that will allow this type of congregation to make themselves known, offer online services and communicate with their community.

The architecture design is based on the use of services such as AWS Lambda, Amazon Amplify, Amazon CloudFront, and API Gateway. As the main services, AWS Lambda enables the execution of code without the need to provision or manage servers, ensuring automatic scalability and high availability of the designed architecture. Amplify is used for secure and scalable storage of static content in the Frontend, supported by the S3 service for image and file storage. It has been also necessary to employ a database service, as Amazon DynamoDB, to provide a Backend for the web architecture. Finally, this work evaluates the designed architecture through load testing, for which open-source frameworks and libraries designed for this purpose have been used.

**Keywords:** Serverless computing, web architecture, AWS, Cloud Computing.

# Agradecimientos

Me gustaría primeramente agradecer a Dios, por su paciencia, fidelidad, por su bondad y misericordia. Cada uno de los regalos inmerecidos día a día, por la luz de su precioso evangelio que llegó a mi vida en esta temporada, la belleza de sus grandes obras y sus promesas. Por sostenerme y ser mi fuerza.

Doy gracias a Dios también por mi familia, quienes aún a la distancia han cuidado, apoyado y animado a la distancia, una familia maravillosa de la cual me alegro de ser parte.

Tampoco quiero dejar pasar la oportunidad de agradecer a mis tutores Germán y Amanda, por ser esas maravillosas personas que me han estado guiando, instruyendo y apoyando para avanzar en este reto y también para crecer profesionalmente.

Por último, pero no menos importante, a la comunidad de hermanos en la Iglesia Bautista reformada de Valencia, por su fidelidad a la palabra y su buen testimonio, por su bondad y compañerismo al conocerlos, estoy enormemente agradecido con el Señor por su iglesia local aquí en Valencia.

# Tabla de contenido

# Índice de Ilustraciones

# Índice de tablas

# 1 Introducción

En este trabajo se investiga la problemática de la Iglesia Bautista Reformada, una institución religiosa ubicada en Valencia. Dado que esta institución es visitada tanto por sus pobladores cómo por extranjeros, requiere de mecanismos de comunicación eficiente con sus fieles, ya que muchos de ellos llegan sin saber con qué tipo de congregación se encontrarán. No están seguros si estarán de acuerdo con la fe que profesan o, para aquellos que desean conocer, abrir un canal sencillo que les permita acercarse y profundizar en la fe cristiana. Es por esta razón que surge la necesidad de proveer a los visitantes y participantes de la congregación una aplicación web que ofrezca toda la información necesaria. Dicha aplicación mostrará de forma clara su declaración de fe, historia y actividades. Además, proporcionará herramientas, incluyendo recomendaciones de otras iglesias en España con una declaración de fe similar, material de estudio, aprendizaje y reflexión. Entre estas herramientas se encuentra la publicación del contenido de las últimas prédicas dadas en el culto dominical.

Dado que este tipo de instituciones religiosas son (según el caso) sin ánimo de lucro, es necesario encontrar una solución económicamente eficiente para ofrecer este canal de comunicación. Es por lo que este trabajo de fin de máster diseña y propone una arquitectura w*eb* basada en el paradigma *Serverless* con el objetivo de proporcionar una solución caracterizada por la escalabilidad automática, la alta disponibilidad y basada en el modelo de pago por uso, que podría ser beneficioso para las congregaciones religiosas. Considerando el contexto actual en el que se desarrolla, caracterizado por la creciente adopción del modelo de computación de *Cloud Computing* debido a ventajas cómo escalabilidad y la baja o nula inversión inicial para acceder a recursos computacionales. el diseño e implementación de la arquitectura *Web Serverless* para satisfacer las necesidades teológicas de la Iglesia Bautista Reformada se basa en estos dos paradigmas de computación. Este trabajo permitirá a esta congregación darse a conocer, mostrando sus principios con claridad, así como el contenido de las últimas predicaciones, libros de estudio recomendados, la recomendación de otras iglesias con una declaración de fe similar, las futuras actividades programadas.

En este capítulo se analiza con más detalle el contexto del trabajo, la motivación y los objetivos de este. El capítulo finaliza repasando la estructura de este documento.

## 1.1 Contexto

A lo largo de la historia, el desarrollo de *software* ha pasado por diversos paradigmas en cuanto a diseño e implementación. La irrupción de los grandes proveedores de infraestructura en la nube, bajo el paradigma del *Cloud Computing,* ha transformado el desarrollo de *software*, permitiendo el despliegue de aplicaciones y servicios sobre los recursos que proporcionan estos proveedores. Esta migración al *Cloud* podía ser ventajosa para algunos usuarios, cómo empresas, fundaciones y particulares, gracias a aspectos cómo el modelo de pago por uso o la alta disponibilidad de estos proveedores, suponiendo una ventajosa alternativa al anterior modelo *on-premises* [1]. La migración es accesible para cualquier usuario con acceso a internet y, gracias a su catálogo de servicios, ofrece un gran potencial para muchos proyectos, ya que el proveedor ofrece a los usuarios recursos y servicios necesarios para desplegar la infraestructura de una aplicación en la nube. Cuando en la arquitectura se incluye a la máquina o máquinas que actúan como servidor, esta se conoce cómo *serverful computing*. Al utilizar esta arquitectura, el

proveedor de infraestructura se encarga del mantenimiento de la infraestructura física, mientras que el equipo de desarrollo es responsable del diseño y despliegue de la arquitectura virtual alojada en la nube del proveedor. Esto incluye la consideración de la complejidad en la selección, configuración e implementación de la amplia variedad de servicios necesarios para gestionar la arquitectura. En contraposición a esta arquitectura, se encuentra el *serverless computing* [2].

El concepto de *serverless computing* se refiere a un modelo de programación y arquitectura en el que se utilizan servicios gestionados por proveedores Cloud responsables de asignar de forma dinámica los recursos de cómputo y/o almacenamiento. En este sentido, FaaS (*Functions as a Service*) es un modelo de servicio donde los fragmentos de código se ejecutan en un entorno *Cloud* y se abstrae al usuario de la infraestructura subyacente, tanto física como virtual. En este modelo, el usuario no tiene control sobre los recursos computacionales donde se ejecuta el código, salvo por la cantidad de memoria dedicada a la ejecución de la función. Aunque continúa la existencia de servidores, su gestión ya no depende del usuario. Los grandes proveedores de servicios *Cloud*, cómo Amazon Web Services [3], Google Cloud [4], IBM Cloud [5] o Microsoft Azure [6], ofrecen herramientas que forman parte del ecosistema *serverless* y con las cuales se pueden desarrollar aplicaciones de estas características. En este modelo, los proveedores de servicios *Cloud* son los encargados de la gestión y aprovisionamiento de los recursos necesarios para el correcto funcionamiento del código. Esto puede ser ventajoso para el usuario, ya que evita preocupaciones de gestión de infraestructura, abstrayéndola casi totalmente.

Este trabajo se enmarca en la constante evolución de los paradigmas del desarrollo de software y la creciente influencia del Cloud Computing, que ha revolucionado la forma en que las organizaciones acceden y utilizan recursos tecnológicos. La migración hacia la nube ha aumentado el acceso a infraestructuras escalables y flexibles, permitiendo a una amplia gama de usuarios aprovechar un extenso catálogo de servicios para la implementación de proyectos innovadores. En este contexto, el diseño e implementación de una arquitectura web *serverless* para la Iglesia Bautista Reformada representa un paso significativo hacia la integración de la tecnología en la difusión de principios y contenidos relevantes en la era digital, demostrando cómo la innovación tecnológica puede potenciar la comunicación y el alcance de instituciones con propósitos específicos.

## 1.2 Motivación

Este trabajo se enfoca en el diseño e implementación de una arquitectura web *serverless* para satisfacer las necesidades tecnológicas de la Iglesia Bautista Reformada. Esta arquitectura web proporcionará una plataforma ágil, escalable y eficiente en cuanto a costes que permitirá a este tipo de congregaciones darse a conocer, mostrar con claridad sus principios, ofrecer servicios en línea y comunicarse con su comunidad.

La selección del tema se basó en el concepto de FaaS y sus casos de uso, vistos en el temario del máster. Se decidió diseñar una solución basada en esta tecnología *serverless*, ya que en los últimos años se ha vuelto una opción rentable económicamente, para ciertos tipos de aplicaciones que requieren patrones de uso variables, y también es eficiente en cuanto a tiempo de desarrollo y configuración de la infraestructura. En este trabajo se exploran los servicios de AWS que permiten ejecutar código, administrar datos e integrar aplicaciones sin la necesidad de administrar servidores.

Así, este trabajo aprovecha las ventajas de AWS Lambda, el servicio de AWS para la ejecución de funciones cómo servicio basadas en eventos, y que permite la creación de

aplicaciones con un bajo coste asociado y sin necesidad de tener una infraestructura pre-aprovisionada. La tecnología *serverless* incluye escalabilidad automática, alta disponibilidad integrada y un modelo de facturación de pago por uso en milisegundos para evitar costes excesivos de aprovisionamiento [7]. Resulta interesante el modo de ejecución de una función Lambda, donde se ejecuta un fragmento de código en cada invocación, sin necesidad de administración o configuración de la infraestructura subyacente y con tiempos de ejecución medidos en milisegundos. Al pagar solo por el tiempo en el que la función está en ejecución, el coste de los servicios de cómputo involucrados en la ejecución está ajustado al milisegundo, dando un mayor control en los gastos. Esta característica es de especial interés para congregaciones sin ánimo de lucro, cómo la Iglesia Bautista Reformada.

## 1.3 Objetivos

El trabajo de fin de máster presentado en esta memoria tiene como objetivo principal:

- Desarrollar una arquitectura web *serverless* basada en los servicios de AWS que permita la entrega de información, herramientas de estudio y anuncios de la Iglesia Bautista Reformada a sus usuarios.

  El objetivo principal se divide en los siguientes objetivos específicos:

  - Análisis y selección de los servicios a utilizar dentro del proveedor AWS, que requiere:
    - Identificar y seleccionar los servicios que cumplan con los requisitos de la arquitectura *Web Serverless* para la Iglesia Bautista Reformada.
    - Analizar y evaluar los costes asociados a los servicios seleccionados, asegurándose de que sean adecuados para la arquitectura y no sean muy elevados.
  - Diseño de la arquitectura web *serverless,* que requiere:
    - Identificar qué servicios corresponden al *Frontend* y cuáles servicios al *Backend*.
    - Diseñar la integración entre los servicios de AWS seleccionados, asegurando una comunicación fluida y eficiente.
  - Implementación de la arquitectura web *serverless* en la infraestructura de AWS.
    - Desarrollar la aplicación que se desplegará en el *Frontend*.
    - Desarrollar la conexión con la base de datos.
    - Configuración y despliegue de los servicios de AWS necesarios para la arquitectura.

Para ello, en este trabajo en primer lugar se analizan y seleccionan los servicios a utilizar, seguido de la metodología aplicada para el desarrollo del proyecto. La arquitectura del sistema se describe antes de detallar las fases de desarrollo y despliegue. Posteriormente, se expone el proceso de validación de los resultados obtenidos. Finalmente, se concluye con un capítulo de conclusiones y trabajos futuros, seguido de la bibliografía y los anexos pertinentes que complementan la investigación.

# 2 Tecnologías Relacionadas.

En este capítulo se describen las tecnologías, plataformas y librerías empleadas en el desarrollo de la arquitectura web *serverless* presentada en este trabajo. El capítulo se estructura en diversas secciones, a través de las que se presentan y analizan las herramientas esenciales del desarrollo del trabajo. La primera sección corresponde a la explicación de los lenguajes de programación Java y *TypeScript*, este último, utilizado como base para el desarrollo del *Framework* Angular, descrito después. La siguiente sección agrupa la descripción de diversos servicios de AWS involucrados en este trabajo, como son los servicios utilizados para el *Frontend* (CloudFront, Amplify, S3), los servicios utilizados para el *Backend* (API Gateway, Lambda, DynamoDB) y el servicio de monitorización CloudWatch que nos permite visualizar servicios previamente mencionados, en específico para verificar el comportamiento de AWS *Lambda* y DynamoDB. Por último, se describe la herramienta JMeter que permite realizar pruebas de carga para analizar el rendimiento de las aplicaciones para verificar el comportamiento de AWS Lambda y el comportamiento de la capacidad aprovisionada de Amazon DynamoDB.

## 2.1 Java

Java es un lenguaje de programación de alto nivel y orientado a objetos. Permite la creación de programas utilizando objetos que interactúan entre sí para realizar tareas específicas. Java utiliza el paradigma de programación orientada a objetos, lo que facilita la reutilización de código y el desarrollo de aplicaciones escalables y modulares [8]. Es administrado por Oracle Corporation [9], una empresa líder en tecnología que se encarga del desarrollo, mantenimiento y actualización del lenguaje Java, así como de proporcionar soporte técnico a los desarrolladores [10].

En cuanto a su licencia, Java ha tenido cambios a lo largo del tiempo. Inicialmente, fue lanzado bajo la licencia de código abierto GNU General Public License (GPL) [11] en su versión 2. Sin embargo, a partir de Java 11, Oracle ha adoptado una licencia comercial llamada Oracle Technology Network License Agreement (OTN-LA) [12]. Esto implica que, aunque el código fuente de Java sigue siendo accesible, el uso comercial de Java puede requerir una licencia.

El lenguaje de programación Java es ampliamente utilizado en el desarrollo de aplicaciones debido a su portabilidad y versatilidad. A continuación, se describe brevemente cómo se desarrolla, compila, interpreta y ejecuta el código en Java:

- Desarrollo: Para desarrollar en Java, se utiliza un Entorno de Desarrollo Integrado (IDE) como Eclipse [13], IntelliJ [14] o Visual Studio Code [15]. Estos IDE proporcionan herramientas y funcionalidades que facilitan la escritura y organización del código.
- Compilación: El código fuente escrito en Java debe ser compilado antes de poder ser ejecutado. El compilador de Java convierte el código fuente en *bytecode*, que es un lenguaje de bajo nivel que puede ser interpretado por la Máquina Virtual de Java (JVM).
- Interpretación: Una vez compilado, el *bytecode* se ejecuta en la JVM. La JVM interpreta el *bytecode* y lo ejecuta línea por línea. Durante la interpretación, la JVM realiza diversas tareas como la gestión de memoria, la resolución de referencias y la ejecución de instrucciones.

- Ejecución: La JVM carga el *bytecode* en memoria y lo ejecuta, produciendo los resultados deseados de la aplicación. Durante la ejecución, la JVM gestiona la asignación de recursos, la ejecución de hilos y la interacción con el sistema operativo.

Por lo tanto, el desarrollo en Java implica escribir y organizar el código en un IDE, luego el código se compila en *bytecode* utilizando el compilador de Java. El *bytecode* es interpretado y ejecutado por la JVM, que se encarga de gestionar la ejecución y proveer los resultados esperados.

Para este proyecto, Java se utiliza como lenguaje de programación para desarrollar un servicio capaz de responder a las solicitudes mediante HTTP, de información almacenada en el servicio de almacenamiento NoSQL de Amazon DynamoDB, respondiendo con un archivo JSON (JavaScript Object Notation)[16].

## 2.2 TypeScript

TypeScript es un lenguaje de programación de código abierto desarrollado y mantenido por Microsoft. Es un lenguaje de programación fuertemente tipado que se basa en JavaScript, lo que significa que cualquier programa JavaScript válido también es un programa TypeScript válido. Sin embargo, TypeScript agrega características y mejoras adicionales al lenguaje JavaScript que ayudan a mejorar la productividad y la calidad del código [17]. Beneficios aportados por las características de TypeScript:

- Tipado estático: TypeScript introduce el concepto de tipos estáticos en JavaScript. Esto significa que se pueden declarar y utilizar tipos para las variables, parámetros de función y valores de retorno de función. El tipado estático permite detectar y prevenir errores comunes durante el desarrollo, mejorando la calidad del código y facilitando su mantenimiento.
- Soporte para características de ECMAScript: TypeScript está diseñado para ser compatible con las últimas características de ECMAScript (el estándar en el que se basa JavaScript). Esto significa que se pueden utilizar características como clases, módulos, funciones flecha, desestructuración y más, incluso si el entorno de ejecución no las admite directamente.
- Autocompletado y herramientas de desarrollo: TypeScript proporciona una experiencia de desarrollo mejorada mediante el uso de autocompletado y herramientas de análisis estático. Estas herramientas ayudan a los desarrolladores a escribir código más rápido y detectar posibles errores antes de que ocurran.
- Refactorización y mantenibilidad: TypeScript facilita la refactorización del código al proporcionar información y sugerencias útiles durante el proceso. Esto ayuda a los desarrolladores a realizar cambios en el código de manera segura y eficiente, sin introducir errores.
- Integración con herramientas y *frameworks*: TypeScript es ampliamente utilizado en el desarrollo de aplicaciones web y móviles. Se integra bien con herramientas populares como Visual Studio Code y diversos *frameworks* como Angular [18], React [19] y Node.js [20].

TypeScript se utiliza en este proyecto para desarrollar todas las funcionalidades del *Frontend* y un servicio capaz de generar las solicitudes mediante HTTP al *Backend*, de la información almacenada en el servicio de Amazon DynamoDB, y la respuesta es recibida en un archivo JSON, que posteriormente es tratada para convertirla en un listado de objetos acorde al servicio.

## 2.3 Framework Angular

Angular es un *framework* de desarrollo de aplicaciones web de código abierto desarrollado y mantenido por Google. Se utiliza para crear aplicaciones web de una sola página (*Single Page Application* - SPA) y aplicaciones móviles híbridas. Angular se basa en el lenguaje TypeScript y utiliza la arquitectura de componentes para facilitar el desarrollo y la reutilización del código [21]. Angular ofrece:

- Arquitectura de componentes: Angular utiliza una arquitectura de componentes, lo que significa que las aplicaciones se dividen en componentes reutilizables. Cada componente representa una parte de la interfaz de usuario y contiene su propia lógica y plantillas HTML. Esto facilita la construcción y el mantenimiento de aplicaciones complejas.
- Enlace de datos bidireccional: Angular ofrece enlace de datos bidireccional, lo que significa que los cambios realizados en los datos del modelo se reflejan automáticamente en la interfaz de usuario y viceversa. Esto simplifica la manipulación y actualización de datos en la aplicación.
- Inyección de dependencias: Angular utiliza un sistema de inyección de dependencias para facilitar la gestión de las dependencias entre los componentes de la aplicación. Esto permite una mayor modularidad y reutilización del código.
- Enrutamiento: Angular proporciona un enrutador incorporado que permite la navegación entre diferentes vistas y componentes dentro de la aplicación. Esto facilita la creación de aplicaciones de varias páginas y mejora la experiencia del usuario.
- Pruebas automáticas: Angular se diseñó teniendo en cuenta la facilidad de las pruebas. Proporciona herramientas y bibliotecas integradas para escribir y ejecutar pruebas unitarias y de integración, lo que facilita la detección de errores y la mejora de la calidad del código.
- Amplia comunidad y soporte: Angular cuenta con una amplia comunidad de desarrolladores y una documentación detallada que proporciona soporte y recursos para el desarrollo de aplicaciones. Además, Google brinda soporte activo y continúa mejorando y actualizando el *framework*.

Para este proyecto, el *framework* Angular es utilizado para el desarrollo de la arquitectura *Frontend* basada en componentes utilizando el lenguaje de TypeScript, generando así una SPA ofreciendo una mejor experiencia de usuario y conectando a través de HTTP con el Backend.

## 2.4 Amazon Web Services

Amazon Web Services (AWS) [22] es una plataforma de servicios en la nube con más de 200 servicios integrales de centros de datos a nivel global. Muchos clientes, incluyendo empresas emergentes, empresas grandes y organismos gubernamentales, utilizan AWS para disminuir costes, incrementar su agilidad e innovar con mayor rapidez [23]. Las características de esta plataforma son:

- Funcionalidad: AWS es una plataforma de servicios en la nube que proporciona una amplia gama de soluciones de infraestructura, incluyendo cómputo, almacenamiento y bases de datos. Además, ofrece tecnologías avanzadas como el aprendizaje automático, la inteligencia artificial y el Internet de las cosas. AWS también ofrece una amplia variedad dentro de sus servicios, como diferentes tipos de bases de datos y tecnologías de almacenamiento, cada una diseñada para ajustar el coste y el rendimiento según las necesidades del cliente.

- Seguridad: AWS es una plataforma diseñada para proporcionar un entorno flexible y seguro para las operaciones en la nube. Con un conjunto de herramientas de seguridad que abarca 230 servicios y características de gobernanza y conformidad, AWS ofrece un alto nivel de seguridad para los datos de los clientes. Además, AWS es compatible con 90 estándares de seguridad y certificaciones de conformidad. Los 117 servicios de AWS que almacenan datos de los clientes ofrecen la función de cifrar los datos, lo que proporciona un mayor nivel de protección para la información sensible del cliente.
- Innovación: AWS ofrece soluciones avanzadas como AWS *Lambda* [24], una plataforma sin servidor que permite a los desarrolladores ejecutar código sin la necesidad de aprovisionar ni administrar servidores, así como Amazon SageMaker[25], un servicio de aprendizaje automático totalmente administrado que democratiza el uso del aprendizaje automático y permite a los desarrolladores y científicos aprovechar esta tecnología sin requerir experiencia previa en el campo.
- Red global de regiones: AWS posee una infraestructura subyacente que permite ofrecer regiones con múltiples zonas de disponibilidad interconectadas por redes de baja latencia, alto rendimiento y redundantes, con 102 zonas disponibles en 32 regiones geográficas alrededor del mundo [26].[26]
- Amplia experiencia y trayectoria: Presentándose al público en el año 2006, actualmente cuenta con más de una década de experiencia [27].

En este trabajo, se ha seleccionado al proveedor Cloud AWS, considerando su amplia experiencia y trayectoria, su funcionalidad, escalabilidad, seguridad, innovación y distribución global de regiones. AWS proporcionará la plataforma tecnológica sólida y confiable que se necesita para llevar a cabo el diseño de la arquitectura de manera satisfactoria.

En el marco de la arquitectura web presentada en esta memoria, se han utilizado los siguientes servicios proporcionados por AWS: Amazon Amplify [28], API Gateway [29], Amazon CloudWatch [30], AWS Lambda, Amazon DynamoDB [31], Amazon CloudFront [32] y Amazon S3 [33]. Cada uno de ellos se describirá de manera detallada en las subsecciones siguientes.

### 2.4.1 Amazon S3

Amazon *Simple Storage Service* (S3) [34] es un servicio de almacenamiento de objetos diseñado para permitir a los clientes almacenar y recuperar cualquier cantidad de datos. El servicio ofrece características de administración que permiten a los clientes organizar los datos y configurar los controles de acceso a los objetos. Además, Amazon S3 garantiza una durabilidad del 99,99999999%.

Los datos se almacenan como objetos dentro de *buckets*, que son los contenedores de los objetos. Se pueden tener uno o más *buckets*, y se pueden controlar quién tiene acceso al *bucket*, quién puede crear, eliminar o enumerar los objetos dentro de él, ver los registros de acceso al *bucket*, y seleccionar la región geográfica para almacenar el *bucket* y su contenido.

En esta memoria, se utiliza el servicio de S3 para el almacenamiento de ficheros estáticos en específico las imágenes mostradas en la arquitectura web reduciendo el tamaño de la aplicación web SPA alojada en el servicio de AWS Amplify y su entrega de contenido al cliente del navegador. Se realiza la integración a través de Amazon CloudFront, para la reducción de costes generado por la entrega de contenido de S3 hacia internet.

### 2.4.2 Amazon CloudFront

Amazon CloudFront es un servicio de distribución de contenido (CDN) ofrecido por AWS. Proporciona una red global de servidores en la nube diseñada para acelerar la entrega de contenido web estático y dinámico a los usuarios finales [35].

CloudFront ayuda a mejorar la experiencia del usuario al reducir los tiempos de carga de los sitios web, aplicaciones y otros recursos digitales al almacenar en caché el contenido en ubicaciones estratégicas cercanas a los usuarios finales. Esto permite entregar contenido de manera eficiente y rápida, reduciendo la latencia y mejorando el rendimiento [36]. Las características que ofrece el servicio Amazon CloudFront son:

- Distribución global: CloudFront cuenta con una red de servidores distribuidos en múltiples ubicaciones alrededor del mundo, lo que permite entregar contenido a los usuarios finales desde el servidor más cercano a ellos. Esto reduce la distancia que la información debe recorrer, mejorando la velocidad de carga y la experiencia del usuario.
- Almacenamiento en caché: CloudFront almacena en caché el contenido en sus servidores distribuidos, lo que significa que una vez que un usuario solicita un recurso, como una imagen o un archivo, este se almacena temporalmente en el servidor más cercano al usuario. Si otro usuario solicita el mismo recurso, CloudFront lo entrega desde la caché, evitando la necesidad de acceder al servidor de origen, lo que acelera la entrega del contenido.
- Seguridad y protección: CloudFront ofrece características de seguridad para proteger el contenido distribuido. Esto incluye la capacidad de configurar políticas de acceso y autenticación, así como la capacidad de usar certificados SSL/TLS para cifrar las comunicaciones entre los usuarios finales y CloudFront.
- Integración con otros servicios de AWS: CloudFront se integra estrechamente con otros servicios de AWS, como Amazon S3, Amazon EC2 y Amazon Lambda. Esto permite una fácil configuración y administración de la distribución de contenido a través de la consola de AWS.
- Análisis y monitoreo: CloudFront proporciona herramientas de análisis y monitoreo para ayudar a los administradores a comprender el rendimiento de la distribución de contenido. Esto incluye métricas, registros de acceso y la capacidad de entregar registros a servicios como Amazon S3 y Amazon Athena para un análisis más profundo.

En este proyecto, se utiliza Amazon CloudFront considerando sus ventajas, principalmente de almacenamiento en cache, la integración con otros servicios, la distribución global y la reducción de costes. Esto se logra al aprovechar el contenido almacenado en cache y evitar peticiones innecesarias al servicio AWS Amplify, además de minimizar los costes asociados con S3 al utilizar CloudFront como intermediario en la entrega de contenido estático.

### 2.4.3 Amazon Amplify

Amazon Amplify es un conjunto de herramientas y servicios desarrollados por AWS que facilitan la creación y el despliegue de aplicaciones web y móviles. Amplify proporciona una forma sencilla de crear aplicaciones escalables y seguras, integrando diferentes servicios de AWS [37]. Sus características son las siguientes:

- Desarrollo simplificado: Amplify permite a los desarrolladores crear aplicaciones rápidamente mediante la utilización de bibliotecas de cliente, generadores de código y

modelos predefinidos. Esto permite acelerar el proceso de desarrollo y reducir la complejidad.
- Integración con servicios de AWS: Amplify se integra con una amplia gama de servicios de AWS, como AWS AppSync, AWS Lambda, Amazon S3 y Amazon DynamoDB, entre otros. Esto permite a los desarrolladores aprovechar la potencia y la escalabilidad de estos servicios en sus aplicaciones.
- Autenticación y autorización: Amplify proporciona opciones de autenticación y autorización listas para usar, lo que permite a los desarrolladores agregar fácilmente características de inicio de sesión, registro de usuarios y gestión de permisos en sus aplicaciones.
- Hosting y despliegue: Amplify ofrece servicios de alojamiento y despliegue que permiten a los desarrolladores implementar sus aplicaciones de manera rápida y sencilla. Esto incluye la opción de implementar aplicaciones web estáticas, así como funciones de *Backend* sin servidor.
- Escalabilidad y rendimiento: Amplify facilita la escalabilidad de las aplicaciones al permitir el ajuste automático de la capacidad y el aprovisionamiento de recursos adicionales según sea necesario. Esto garantiza que las aplicaciones puedan manejar cargas de trabajo variables y mantener un rendimiento óptimo.

En este trabajo se emplea el servicio de Amplify como alojamiento de la aplicación web orientada al Frontend. Este servicio permite el rápido despliegue de aplicaciones, la continua integración y entrega/implementación (CI/CD), el despliegue es posible gracias a su servicio de hosting escalable y simplificado para el desarrollo, lo que permite la integración con otros servicios de AWS. Además, se mejora el rendimiento y se controla los costes de la aplicación mediante la red de entrega de contenido (CDN) de Amazon CloudFront.

### 2.4.4 Amazon API Gateway

Amazon API Gateway es un servicio completamente administrado por AWS que permite a los desarrolladores crear, publicar, mantener, monitorear y proteger fácilmente APIs (Interfaces de Programación de Aplicaciones) para sus aplicaciones y servicios en la nube.

API Gateway actúa como un punto de entrada para las aplicaciones y servicios, permitiendo a los desarrolladores exponer sus APIs de manera segura y controlada. Permite a los desarrolladores crear APIs RESTful (*Representational State Transfer*) o WebSocket utilizando una variedad de servicios *Backend*, como AWS Lambda, Amazon EC2, y otros servicios web. Las características ofrecidas del servicio Amazon API Gateway son:

- Creación y configuración de APIs: API Gateway proporciona una interfaz fácil de usar para crear y configurar APIs. Los desarrolladores pueden definir endpoints, métodos HTTP, parámetros, modelos de datos y otros aspectos de la API utilizando la consola de administración de AWS o mediante la API de línea de comandos.
- Escalabilidad y alta disponibilidad: API Gateway es altamente escalable y se puede configurar para manejar cargas de tráfico variables. Puede escalar automáticamente para manejar picos de tráfico y distribuir la carga entre múltiples instancias en diferentes regiones de AWS, asegurando así la alta disponibilidad de la API.
- Integración con servicios *Backend*: API Gateway permite integrar fácilmente servicios *Backend*, como AWS Lambda, Amazon EC2, y otros servicios web, para procesar las

solicitudes de la API. Esto permite a los desarrolladores construir APIs sin tener que preocuparse por la infraestructura subyacente.
- Gestión de versiones y despliegues: API Gateway facilita la gestión de versiones y despliegues de APIs. Los desarrolladores pueden crear diferentes versiones de una API y controlar la forma en que los cambios se implementan y se hacen disponibles para los consumidores de la API.
- Seguridad y autorización: API Gateway ofrece opciones de seguridad y autorización para proteger las APIs. Los desarrolladores pueden configurar políticas de acceso, autenticación y autorización basadas en tokens de seguridad, como API keys, tokens de acceso de OAuth, y certificados SSL/TLS.
- Monitoreo y análisis: API Gateway proporciona herramientas de monitoreo y análisis para rastrear y analizar el tráfico de la API. Los desarrolladores pueden ver métricas en tiempo real, registros de acceso y generar informes para comprender el rendimiento y la utilización de la API.

Se selecciono para este proyecto el servicio de Amazon API Gateway como desencadenador de eventos, la exposición de su API, la capacidad que brinda de monitorización, la gestión de versiones e integraciones. Estas funcionalidades las ofrece este servicio, con la ventaja característica de la tecnología *serverless* que no requiere de configuración de infraestructura, sino solo configuración de funcionalidad e integración con las funciones *Lambda* a las que se desea conectar y que para este proyecto son las que estarán ofreciendo el servicio de *Backend*.

### 2.4.5 AWS Lambda

AWS Lambda es un servicio que se enmarca en el paradigma de la computación sin servidor (*serverless*). Su función principal es la ejecución de código en respuesta a eventos, con la particularidad de que se encarga de la gestión de los recursos de cómputo de forma transparente al usuario.

La implementación de un código en AWS Lambda supone la creación de lo que se denomina función Lambda, que estará siempre lista para su ejecución en cualquier momento en el que se active. La función Lambda puede definir parámetros como la cantidad de memoria requerida, el lenguaje utilizado, el tiempo máximo de duración y otros elementos relevantes como variables de entorno y roles de usuario.

Es importante tener en cuenta que AWS Lambda se encarga de diversos aspectos de la infraestructura subyacente, como el mantenimiento del servidor y el sistema operativo, aprovisionamiento de capacidad y escalado automático, implementación de parches de seguridad y código, y la monitorización de código y registros. Lo único que se necesita proporcionar a Lambda es el código de la función para su correcto funcionamiento. Las características de AWS Lambda son:

- Las funciones Lambda son *stateless* y no tienen una relación fija con la infraestructura subyacente. La tecnología de AWS Lambda permite que se creen copias de la función según sea necesario para adaptar la escala de los eventos entrantes.
- AWS Lambda integra la funcionalidad de tolerancia a fallos. Esto significa que se asegura de mantener su capacidad de cómputo en varias zonas de disponibilidad de cada región, con el fin de proteger el fragmento de código de posibles fallos en los equipos individuales o en el centro de datos.
- La invocación del código en AWS Lambda solo se realiza cuando se necesita, en respuesta a un evento previamente configurado, por lo que se escala de forma

automática para satisfacer la demanda de las solicitudes entrantes. No es necesario que el usuario haga ninguna configuración adicional.

- En AWS Lambda se utiliza una estrategia de pago basada en la duración de la ejecución, en lugar del número de unidades del servidor. Al utilizar funciones Lambda, solo se paga por las solicitudes que se atienden y por el tiempo que toma la ejecución de código.

En este trabajo se utiliza el servicio AWS Lambda debido a sus ventajas. Al ejecutar porciones de código basado en eventos, permitiendo la creación de aplicaciones con un bajo coste asociado y sin necesidad de tener infraestructura pre-aprovisionada, evitando la administración y gestión de esta, debido a que el propio servicio incluye escalabilidad automática, alta disponibilidad integrada y un modelo de facturación de pago por uso en milisegundos para evitar costes excesivos de aprovisionamiento. Específicamente en este trabajo se hace uso del servicio de AWS Lambda para la comunicación y gestión de los datos entre la base de datos (DynamoDB) y el cliente (SPA entregada a través de CloudFront).

### 2.4.6 Amazon DynamoDB

Amazon DynamoDB ofrecido por AWS, es un servicio de base de datos NoSQL [38] totalmente administrado y altamente escalable. Es una base de datos clave-valor y de documentos que proporciona un rendimiento rápido y predecible, así como una escalabilidad automática bajo demanda. Está diseñado para soportar aplicaciones que requieren una latencia muy baja y un alto rendimiento en cualquier escala [39] ofreciendo las características a continuación.

- Escalabilidad automática: dispone de la capacidad de escalar de forma automática y transparente para manejar cualquier carga de trabajo, desde algunas solicitudes por segundo hasta millones de solicitudes por segundo, sin necesidad de ajustes de capacidad manual.
- Rendimiento rápido y predecible: ofrece tiempos de respuesta mínimos en orden de milisegundos a escala, lo que posibilita comunicaciones con baja latencia para aplicaciones en tiempo real y una alta concurrencia para cargas de trabajo intensivas.
- Durabilidad y disponibilidad: replica automáticamente los datos en múltiples zonas de disponibilidad (AZ) dentro de una región de AWS para proporcionar alta disponibilidad y durabilidad de los datos.
- Modelo de datos flexible: DynamoDB admite un modelo de datos flexible, lo que significa que puede almacenar y recuperar datos en formato de clave-valor o de documento, lo que permite una mayor flexibilidad en el desarrollo de aplicaciones.
- Integración con otros servicios de AWS: DynamoDB se integra con otros servicios de AWS, como AWS Lambda, Amazon S3 y Amazon Kinesis, lo que facilita la construcción de aplicaciones escalables y sin servidor.

En síntesis, Amazon DynamoDB es una base de datos NoSQL altamente escalable y flexible que ofrece un rendimiento rápido y predecible. Es una buena opción para aplicaciones que requieren una baja latencia, alta disponibilidad y una capacidad de escala automática.

En este proyecto se utiliza Amazon DynamoDB como base de datos, considerando sus características inherentes de escalabilidad automática, su rendimiento rápido y predecible, característica muy relevante para reducir el tiempo de la ejecución de la

función Lambda que solicita la información al integrar el servicio de AWS DynamoDB con AWS Lambda. Su flexibilidad brinda una característica de mutabilidad permitiendo agregar más elementos, con información adicional y/o actualizar los elementos sin necesidad de creación de nuevas tablas o bases de datos.

### 2.4.1 Amazon CloudWatch

Amazon CloudWatch es un servicio de AWS que ofrece la capacidad de monitorizar los recursos y aplicaciones que se ejecutan en tiempo real. Este servicio permite recopilar y realizar el seguimiento de métricas, que son variables que permiten medir los recursos y aplicaciones del usuario. Además, la plataforma de CloudWatch muestra de forma automática métricas de todos los servicios de AWS que están en uso por el usuario, y también permite la creación de paneles personalizados para mostrar métricas específicas de aplicaciones y colecciones personalizadas [40].

También permite crear alarmas para monitorizar métricas en tiempo real y enviar notificaciones o ejecutar cambios automáticos en los recursos monitorizados si se superan ciertos umbrales. Por ejemplo, se puede monitorear el número de peticiones hechas al API Gateway y las funciones AWS Lambda, las lecturas y escrituras a través del servicio de DynamoDB para monitorizar la capacidad aprovisionada vs la consumida, y utilizar los datos recolectados para realizar la toma de decisiones, ya sea aumentando la capacidad aprovisionada para soportar una mayor carga de trabajo o reducir la capacidad para reducir costes.

CloudWatch permite obtener información sobre la utilización de recursos, desempeño de aplicaciones y el estado operativo del sistema. También existe el servicio Amazon EventBridge, que permite generación de eventos definidos por reglas y que supone la evolución de la funcionalidad de CloudWatch Events en un nuevo servicio. Cuando se cumplen las condiciones de una regla, se genera un evento. Las reglas pueden ser de dos tipos: el primero se genera cuando hay una operación detectada sobre un recurso o servicio de AWS, y el segundo se programa de forma cronológica para la ejecución periódica de eventos establecidos en un horario específico. Estos eventos se pueden usar para activar servicios de AWS como la ejecución de una función de AWS Lambda o el envío de un mensaje de Amazon SNS [41].

Para el desarrollo de este trabajo, CloudWatch se utiliza para monitorizar el comportamiento la activación de las funciones de AWS Lambda, que está dedicada a procesar el código del servicio *Backend* de la *web*. También para el monitoreo de rendimiento y consumo del servicio de DynamoDB, identificando la diferencia que hay entre el consumo real y el aprovisionado en el servicio.

## 2.5 Apache JMeter

Apache JMeter [42] es una herramienta de código abierto desarrollada por Apache que se utiliza para realizar pruebas de carga, rendimiento y estrés en aplicaciones web.

JMeter proporciona una amplia gama de funcionalidades para simular diferentes escenarios de carga y medir el rendimiento de una aplicación. JMeter ofrece grabación de escenario, es decir, las interacciones de un usuario con una aplicación web o *endpoint* de las API y la replicación de estas para simular la carga generada por múltiples usuarios.

También permite la configuración de escenarios definiendo diferentes parámetros para simular diversos escenarios de carga, como el número de usuarios concurrentes, el tiempo de espera entre solicitudes y la distribución de carga, muy útil para analizar el comportamiento de las funciones Lambda en este tipo de escenarios.

Para la medición de rendimiento, JMeter proporciona métricas detalladas sobre el rendimiento de una aplicación, como el tiempo de respuesta, el rendimiento de transacciones, la secuencia temporal en que se lanzan los distintos usuarios simulados y el uso de recursos del servidor.

En el análisis de resultados JMeter genera informes detallados y gráficos que permiten analizar los resultados de las pruebas y detectar posibles cuellos de botella o problemas de rendimiento.

Para este trabajo, se utiliza la herramienta de JMeter para generar escenarios de carga y medir el rendimiento de la API gestionada por el servicio de Amazon API Gateway, gracias a la generación de gráficas que contienen los tiempos de respuesta y la activación de *threads* que simulan los usuarios. Adicionalmente genera los resultados en tablas formato *csv* conteniendo el conteo de muestras lanzadas, el tiempo medio, mínimo, máximo, errores, tamaño de los datos enviados y recibidos.

## 2.6 Análisis de costes

Esta sección se centra en la arquitectura *serverless* utilizando una combinación de servicios de AWS. Los servicios sin servidor permiten a los desarrolladores concentrarse en la lógica de la aplicación sin preocuparse por la infraestructura subyacente. Sin embargo, como en cualquier solución tecnológica, es fundamental evaluar los costes asociados.

En este trabajo, se utilizarán servicios de Amazon Web Services (AWS): Amazon S3 para almacenamiento escalable y seguro de archivos estáticos, AWS Amplify Hosting para facilitar la implementación de aplicaciones web en la red de entrega de contenido (CDN) de AWS, Amazon CloudFront como CDN para acelerar la entrega de contenido, AWS Lambda para ejecución de código sin servidor en respuesta a eventos, Amazon DynamoDB como base de datos NoSQL escalable y administrada, y Amazon API Gateway para la creación de APIs seguras y escalables. En las próximas subsecciones, se profundiza en cada uno de estos servicios, analizando sus características y aprendiendo a optimizar la eficiencia económica de nuestra arquitectura.

### 2.6.1 Precios ofrecidos por Amazon para el servicio de S3

Para un análisis de los costes del servicio de Amazon S3 para este trabajo de máster, es importante comprender los componentes del coste asociados con este servicio. Amazon S3 es un servicio de almacenamiento altamente escalable y seguro que permite a los usuarios almacenar y recuperar datos de manera eficiente. Al evaluar los costes de Amazon S3, es fundamental comprender los componentes que influyen en la factura mensual.

En cuanto al almacenamiento de objetos, el coste se basa en la cantidad de datos almacenados en el servicio y la región. Según la Tabla 1, el coste de almacenamiento por mes varía en función de la cantidad de almacenamiento. Este coste se calcula en función de la cantidad de gigabytes almacenados. Teniendo que en la región del norte de virginia de estados unidos se encuentra entre los más económicos considerando las demás regiones, teniendo un precio por GB de 0.023 USD por mes, lo que es conveniente para esta arquitectura basada en *serverless*.

| S3 Standard | | S3 Intelligent - Tiering | |
|---|---|---|---|
| Almacenamiento | Precio por GB (USD) | Almacenamiento | Precio por GB(USD) |
| Primeros 50 TB/mes | 0,023 | Acceso frecuente, Primeros 50 TB/mes | 0,023 |
| Siguientes 450 TB/mes | 0,022 | Acceso frecuente, Siguientes 450 TB/mes | 0,022 |
| Más de 500 TB/mes | 0,021 | Acceso frecuente, Más de 500 TB/mes | 0,021 |

*Tabla 1. Costes en S3 del almacenamiento por mes (febrero 2024).*

Como este trabajo de máster provee una aplicación web, los datos se accederán con frecuencia, por lo que S3 Standard es la opción más económica para datos de acceso frecuente. En lo que descarta S3 Standard-IA y S3 One Zone-IA es que son opciones de bajo coste para datos de acceso menos frecuente (menos de 3 accesos por mes). Por último, Glacier es ideal para datos de archivo a largo plazo.

Por otro lado, el coste por solicitudes (GET, PUT, COPY, POST o LIST) se basa en la cantidad de solicitudes realizadas al servicio. Según la Tabla 2, el coste se establece por cada 1,000 solicitudes realizadas, y puede variar ligeramente en función de la operación específica. Afortunadamente, Amazon dispone una forma de amortizar los costes relacionados con estas solicitudes a través del servicio Amazon CloudFront, lo cual evita el coste por solicitudes desde el Bucket S3 hacia internet al integrar los dos servicios y haciendo que la comunicación sea dentro de la red privada de Amazon almacenando los objetos en cache del CND en un nodo de los nodos dispuestos en la ubicación, por lo que no se generan costes por petición, lo que permite enfocar la atención y gestión de los costes en la cantidad de almacenamiento consumido por mes.

| | Solicitudes PUT, COPY, POST LIST (por 1000 solicitudes) | GET, SELECT y el resto de las solicitudes (por 1000 solicitudes) |
|---|---|---|
| S3 Intelligent | 0,005 USD | 0,0004 USD |
| S3 Intelligent-Tiering | 0,005 USD | 0,0004 USD |
| S3 Estándar – Acceso poco frecuente | 0,01 USD | 0,001 USD |
| S3 Express One Zone | 0,0025 USD | 0,0002 USD |
| S3 Glacier Instant Retrieval | 0,02 USD | 0,01 USD |
| S3 Glacier Flexible Retrieval | 0,03 USD | 0,0004 USD |
| S3 Glacier Deep Archive | 0,05 USD | 0,0004 USD |
| S3 One Zone-Infrequent Access | 0,01 USD | 0,001 USD |

*Tabla 2. Costes S3 por 1.000 solicitudes (febrero 2024).*

En conclusión, es tomada la elección de Amazon S3 como servicio de almacenamiento para este trabajo de máster por su escalabilidad y flexibilidad. Al analizar los costes, se evidencia que el almacenamiento de objetos se basa en la cantidad de datos almacenados y la región utilizada. La opción S3 Standard es la más económica para datos de acceso frecuente, mientras que las opciones S3 Standard-IA y S3 One Zone-IA son adecuadas para datos de acceso menos frecuente. Además, la integración con Amazon CloudFront

permite amortizar los costes por solicitudes, evitando cargos desde el Bucket S3 hacia internet.

### 2.6.2 Precios ofrecidos por Amazon para el servicio de Amplify Hosting

Para este trabajo es importante la elección de la plataforma de alojamiento adecuada, por lo que Amplify Hosting se presenta como una opción atractiva para alojar aplicaciones *web*. Considerando que Amazon Amplify Hosting es un servicio de alojamiento y CI/CD completamente administrado para aplicaciones web a través de *https*. su objetivo es proporcionar una solución rápida, segura, fiable y escalable para implementar aplicaciones web. Este análisis tiene como objetivo evaluar los costes asociados con el uso de Amplify Hosting para esta memoria.

| | Gratuito por 12 meses | Pago por uso |
|---|---|---|
| CREACIÓN E IMPLEMENTACIÓN | Sin coste hasta los 1000 minutos de creación por mes | 0,01 USD por minuto |
| ALMACENAMIENTO DE DATOS | Sin coste hasta los 5 GB almacenados en la CDN por mes | 0,023 USD por GB por mes (este cargo se repite hasta que se elimine la aplicación) |
| TRANSFERENCIA SALIENTE DE DATOS | Sin coste hasta los 15 GB por mes | 0,15 USD por GB servido |
| RECUENTO DE SOLICITUDES (SSR) | Sin coste hasta 500 000 solicitudes por mes | 0,30 USD por millón de solicitudes |
| DURACIÓN DE SOLICITUDES (SSR) | Sin coste hasta los 100 GB por hora por mes | 0,20 USD por hora (GB por hora) |

*Tabla 3. Costes Amazon Amplify Hosting (febrero 2024).*

Al evaluar los costes, tal como se observa en la Tabla 3, es fundamental considerar los siguientes aspectos que componen los costes:

- Creación e Implementación:
  - El coste se basa en el tiempo de creación y despliegue de la aplicación.
  - Durante el nivel gratuito de AWS, no se aplican cargos por creación hasta los 1,000 minutos por mes.
  - Después del nivel gratuito, el coste es de $0.01 por minuto.
  - Las aplicaciones hechas en Angular para el caso de este trabajo en promedio tardan 30 segundos en compilar y alrededor de 2 a 3 minutos máximo en la implementación de la aplicación.
- Almacenamiento de Datos:
  - No se aplican cargos por almacenamiento durante el nivel gratuito.
  - Después del nivel gratuito, el coste es de $0.023 por GB por mes (este cargo se repite hasta que se elimine la aplicación).
- Transferencia Saliente de Datos:
  - Durante el nivel gratuito, no se aplican cargos hasta los 15 GB por mes.
  - Después del nivel gratuito, el coste es de $0.15 por GB servido.
  - Recuento y Duración de Solicitudes (SSR):
  - Durante el nivel gratuito, no se aplican cargos hasta 500,000 solicitudes por mes y 100 GB por hora.
  - Después del nivel gratuito, el coste es de $0.30 por millón de solicitudes y $0.20 por hora (GB por hora).

La elección de Amazon Amplify Hosting para alojar aplicaciones web serverless se justifica por su enfoque en proporcionar una solución rápida, segura, fiable y escalable para implementar aplicaciones web.

El análisis detallado de los costes asociados con el uso de Amplify Hosting revela que, durante el nivel gratuito de AWS, no se aplican cargos por creación hasta los 1,000 minutos por mes, no se aplican cargos por almacenamiento durante el nivel gratuito, y no se aplican cargos hasta los 15 GB por mes por transferencia saliente de datos.

Además, el servicio no aplica cargos hasta 500,000 solicitudes por mes y 100 GB por hora. Estos aspectos demuestran que Amazon Amplify Hosting ofrece un modelo de costes atractivo para el alojamiento de aplicaciones web siendo estos costes también amortizados por el almacenamiento en cache al integrarlo con el servicio Amazon CloudFront, lo que lo convierte en una opción viable para el desarrollo y despliegue de aplicaciones web en el contexto de este trabajo de máster.

### 2.6.3 Precios ofrecidos por Amazon para el servicio de CloudFront

Para este trabajo de final de máster, se ha elegido utilizar el servicio Amazon CloudFront por varias razones. Este servicio de AWS ofrece una solución de entrega de contenido segura, rápida y programable a nivel mundial. Amazon CloudFront está diseñado para acelerar la transferencia de archivos y la entrega de API, lo que resulta en una mejora significativa en la velocidad y la eficiencia de las aplicaciones web.

La elección de una red de entrega de contenido (CDN) eficiente y económica es crucial. Este análisis tiene como objetivo evaluar los costes asociados con el uso de CloudFront para este proyecto. Para ello es fundamental considerar dos componentes principales: los precios de transferencia por mes Tabla 4 y los costes por tipo de solicitudes Tabla 5, pero primero indicaré la generosa capa gratuita para siempre que ofrece CloudFront según su documentación.

Incluye en el nivel gratuito para siempre: Transferencia de 1 TB de datos a *Internet* por mes, 10.000.000 de solicitudes *HTTP* o *HTTPS* por mes, 2.000.000 de invocaciones de CloudFront Function por mes, 2 000 000 de lecturas de KeyValueStore de CloudFront al mes, Certificados SSL gratuitos. Esta capa gratuita es suficiente para el uso de la arquitectura de este proyecto considerando que por petición estaría descargando un aproximado de 5 MB en recursos de la SPA a su vez que con la capacidad dada en esta capa gratuita de transferencia de datos a *Internet* por mes se logra amortizar los gastos tanto de los servicios Amazon Amplify Hosting y S3 que se ha analizado previamente, considerando que los archivos de estos dos servicios se quedan almacenados en la cache del servicio CloudFront, por el tiempo que se le especifique en la configuración del servicio.

Lo que quiere decir que se deberían hacer por mes un aproximado de 200.000 peticiones por mes para superar la capa gratuita, si se considera solo lo consumido por el servicio de Amplify Hosting, pero también es importante considerar lo consumido por transferencia de datos por parte de S3.

Los factores que pueden generar costes una vez superaran la capa gratuita seria como se mencionó previamente la transferencia de datos (vea Tabla 4), donde se ha marcado con color verde el valor correspondiente a Europa para los primero 10 TB, junto con las zonas dentro de Norte América las más económicas para utilizar este servicio. También es necesario resaltar la diferencia en precios que se observa entre S3 que cobra 0,023 USD por GB, mientras que CloudFront cobra 0,085 por TB de transferencia de datos, la

diferencia es bastante, por lo que es más conveniente que la información y entrega de archivos se realice a través del servicio CloudFront.

| Al mes | Estados Unidos, México y Canadá (USD) | Europa e Israel (USD) | Sudáfrica, Kenia, Nigeria y Oriente Medio (USD) | América del Sur (USD) | Japón (USD) | Australia y Nueva Zelanda (USD) | Hong Kong, Indonesia, Filipinas, Singapur, Corea del Sur, Taiwán, Tailandia, Malasia y Vietnam (USD) | India (USD) |
|---|---|---|---|---|---|---|---|---|
| Primeros 10 TB | 0,085 | 0,085 | 0,11 | 0,11 | 0,114 | 0,114 | 0,12 | 0,109 |
| Siguientes 40 TB | 0,080 | 0,080 | 0,105 | 0,105 | 0,089 | 0,098 | 0,100 | 0,085 |
| Siguientes 100 TB | 0,060 | 0,060 | 0,090 | 0,090 | 0,086 | 0,094 | 0,095 | 0,082 |
| Siguientes 350 TB | 0,040 | 0,040 | 0,080 | 0,080 | 0,084 | 0,092 | 0,090 | 0,080 |
| Siguientes 524 TB | 0,030 | 0,030 | 0,060 | 0,060 | 0,080 | 0,090 | 0,080 | 0,078 |
| Siguientes 4 PB | 0,025 | 0,025 | 0,050 | 0,050 | 0,070 | 0,085 | 0,070 | 0,075 |
| Más de 5 PB | 0,020 | 0,020 | 0,040 | 0,040 | 0,060 | 0,080 | 0,060 | 0,072 |

*Tabla 4. Precios de transferencia por mes (febrero 2024).*

Aunque la capa de Amazon CloudFront es generosa y para este trabajo es suficiente, pues no se estima llegar a generar tantas peticiones en el mes, resulta igualmente importante abordar estos casos especiales. En este caso los costes son un poco más altos que las zonas dentro de Norte América, pero sigue siento más económica que en regiones como América del sur, China, Australia, entre otros.

| | Estados Unidos, México y Canadá (USD) | Europa e Israel (USD) | Sudáfrica, Kenia, Nigeria y Oriente Medio (USD) | América del Sur (USD) | Japón (USD) | Australia y Nueva Zelanda (USD) | Hong Kong, Indonesia, Filipinas, Singapur, Corea del Sur, Taiwán, Tailandia, Malasia y Vietnam (USD) | India (USD) |
|---|---|---|---|---|---|---|---|---|
| Solicitudes HTTP | 0,0075 | 0,009 | 0,009 | 0,016 | 0,009 | 0,009 | 0,009 | 0,009 |
| Solicitudes HTTPS | 0,01 | 0,012 | 0,012 | 0,022 | 0,012 | 0,0125 | 0,012 | 0,012 |

*Tabla 5. Costes por tipo de solicitudes (febrero 2024).*

En caso de tener una tasa de 100 personas consultadas en la aplicación web, durante los 30 días del mes, cada uno recibiría en tamaño 5 MB aproximadamente, 500 MB diariamente y al mes aproximadamente 1,5 GB. La capa gratuita sería más que suficiente para poder absorber esta tasa de transferencias de datos y de cantidad de peticiones generadas por los usuarios. Aún si en el caso hipotético de un aumento de la cantidad de peticiones a 500 por día manteniendo aproximadamente el mismo tamaño de la aplicación web entregada por Amazon CloudFront equivale a 75 GB quedando un amplio margen entre la transferencia de datos al igual que de peticiones hechas con respecto a la capa gratuita.

### 2.6.4 Precios ofrecidos por Amazon para el servicio de Funciones Lambda

Para el desarrollo de una arquitectura *web serverless* se requiere de un servicio como AWS Lambda, un servicio informático sin servidor que te permite ejecutar código sin preocuparte por aprovisionar ni administrar servidores. Este análisis tiene como objetivo evaluar los costes asociados con el uso de AWS Lambda para este trabajo. Los factores de costes de AWS Lambda son:

- Solicitudes: Se cobra por la cantidad de invocaciones a las funciones. Esto incluye eventos como notificaciones de Amazon SNS o llamadas directas desde Amazon API Gateway.
- Duración: Se calcula desde que el código comienza a ejecutarse hasta que regresa o finaliza. El precio depende de la memoria asignada a la función.
- Ejecuciones:
  - Lambda cobra por cada ejecución de una función.
  - El coste depende de la cantidad de memoria utilizada y la duración de la ejecución.
  - Los primeros 3,2 millones de segundos de tiempo de procesamiento por mes son gratuitos[43].
- Memoria:
  - Lambda cobra por la cantidad de memoria asignada a una función, incluso si no se usa toda.
  - La memoria se asigna en incrementos de 64 MB.
  - La memoria configurada para este proyecto fue de 512 MB y en promedio para la función de ubicaciones (*locations*) se toma esta función como referencia debido a que es la función que más elementos devuelve en cada ejecución, utilizando 153MB.
- Red:
  - Lambda cobra por la cantidad de datos transferidos hacia y desde la función.
  - La transferencia de datos entre Lambda y otros servicios de AWS es gratuita. Esto es una ventaja debido a que al integrarse la función con el servicio de AWS API Gateway encargado de generar los eventos necesarios para lanzar funcionalidad de cada función, el tráfico de se mantendrá de forma interna evitando así los gastos en este apartado.
- Almacenamiento:
  - Lambda cobra por el almacenamiento del código de la función y las variables de entorno.
  - El almacenamiento de código es gratuito hasta 50 MB en promedio la compilación de los archivos utilizados para este proyecto ocupa un espacio de almacenamiento de 10MB por archivo JAR.

Costes:

La estructura de precios de Lambda es de pago por uso (ver Tabla 6), lo que significa que los usuarios solo pagan por los recursos que realmente utilizan.

Esto puede ser una opción más precisa para aplicaciones que tienen un uso variable o impredecible. Lambda optimiza automáticamente el uso de recursos, lo que puede ayudar a reducir los costes.

| Arquitectura | Duración | Solicitudes |
|---|---|---|
| Precio de x86 | | |
| Primeros 6 mil millones de GB/segundo por mes | 0,0000166667 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |
| Próximos 9 mil millones de GB/segundo por mes | 0,000015 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |
| Más de 15 mil millones de GB/segundo por mes | 0,0000133334 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |
| Precio de Arm | | |
| Primeros 7,5 mil millones de GB/segundo por mes | 0,0000133334 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |
| Próximos 11,25 mil millones de GB/segundo por mes | 0,0000120001 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |
| Más de 18,75 mil millones de GB/segundo por mes | 0,0000106667 USD por cada GB/segundo | 0,20 USD por un millón de solicitudes |

*Tabla 6. Costes GB/segundo Lambda Function (febrero 2024).*

La elección de AWS Lambda Functions para el desarrollo de la arquitectura web serverless del trabajo fin de máster es una decisión estratégica que ofrece numerosas ventajas en términos de escalabilidad, flexibilidad, eficiencia y optimización de costes. La plataforma se presenta como la herramienta ideal para este tipo de proyectos, permitiendo al equipo de desarrollo enfocarse en la creación de una aplicación innovadora y robusta.

### 2.6.5 Precios ofrecidos por Amazon para el servicio de DynamoDB

Se ha seleccionado el servicio de DynamoDB debido a que ofrece una generosa capa gratuita que incluye un almacenamiento de 25 GB, 25 unidades de lectura y escritura por segundo. Para este proyecto, esta capacidad es suficiente. Además, a medida que el proyecto crezca y dependiendo de la configuración, DynamoDB se adapta automáticamente. También existe la posibilidad de escalar verticalmente aumentando la capacidad de lectura y escritura según sea necesario.

El modelo de conteo para el cobro de pago por uso se realiza por operaciones de lectura, escritura y almacenamiento. Existen dos modos ofrecidos por el servicio Amazon DynamoDB para el aprovisionamiento del servicio.

**Capacidad bajo demanda** (ideal para cargas de trabajo impredecibles):

- No requiere de especificar la capacidad inicial; DynamoDB se ajusta automáticamente según la demanda.
- Aunque su coste es más elevado en comparación con la capacidad aprovisionada, es ideal para situaciones variables.

**Capacidad aprovisionada** (ideal para cargas predecibles):

– Se especifica la capacidad esperada
– Permite un control más preciso de los costes.

Para este trabajo, se ha seleccionado la opción de capacidad aprovisionada. Configurando el servicio con la capacidad que brinda la capa gratuita. Esto permite que la implementación no incurra en gastos en la fase de desarrollo ni de despliegue. En caso de que se superare la capacidad aprovisionada, si se elige la opción de configuración aprovisionada, y sin escalado automático, las peticiones atendidas se limitaran al máximo de RCU (*Read Capacity Units*) y WCU (*Write Capacity Units*) configurados.

Es importante considerar los costes en el caso de que se supere la capa gratuita. Amazon DynamoDB ofrece 2 opciones dentro del servicio aprovisionado de lectura y escritura. El primero corresponde a la configuración estándar del servicio, que se factura mensualmente. Los valores se reflejan en la Tabla 7 y son cobrados por hora por cada unidad de lectura y escritura adicional.

| Clase de tabla DynamoDB Standard | |
|---|---|
| Tipo de rendimiento aprovisionado | Precio por hora |
| Unidad de capacidad de escritura (WCU) | 0,00065 USD por WCU |
| Unidad de capacidad de lectura (RCU) | 0,00013 USD por RCU |

*Tabla 7. Precios DynamoDB Aprovisionado por hora (febrero 2024).*

La segunda opción de pago resulta mejor opción si se tiene muchas peticiones hacia la base de datos o que el total de datos consultado sean de un tamaño muy grande debido a que el tamaño de los elementos consultados y el total de elementos requieren de unidades de lectura y escritura para ser procesados. Con esta opción se termina pagando aproximado un 80% menos que la opción anterior mostrada en la Tabla 7, ya que se realiza un contrato por un año o tres, como se observa en la Tabla 8.

| Capacidad reservada | | | | |
|---|---|---|---|---|
| Contrato mensual | Pago inicial: un año | Tarifa por hora: un año | Pago inicial: tres años | Tarifa por hora: tres años |
| 100 unidades de capacidad de escritura | 150,00 USD | 0,0128 USD | 180,00 USD | 0,0081 USD |
| 100 unidades de capacidad de lectura | 30,00 USD | 0,0025 USD | 36,00 USD | 0,0016 USD |

*Tabla 8. Precio DynamoDB reservado por año y tres años 100 RWCU (febrero 2024).*

Para este trabajo, se ha seleccionado la opción del modo de capacidad aprovisionada estándar para las tablas de datos en Amazon DynamoDB. Esto se debe a que la información que se almacenará en servicio de DynamoDB ocupará un espacio de almacenamiento inferior a 1 GB, lo cual no supera el límite brindado en la capa gratuita. Lo mismo sucede con la capacidad aprovisionada en la capa gratuita que ofrece hasta 200 millones de solicitudes de lectura y escritura al mes.

### 2.6.6 Precios ofrecidos por Amazon para el servicio de API Gateway

El desarrollo de la arquitectura serverless de este trabajo final de máster requiere la creación de una API para la gestión de eventos hacia el Backend. La elección de una plataforma de API robusta y flexible. Amazon API Gateway se presenta como una opción atractiva por su escalabilidad, seguridad y facilidad de uso. Este análisis busca evaluar

los costes asociados con el uso de API Gateway para el trabajo final de máster, junto con el servicio de AWS Lambda. Los factores de costes para este servicio son los siguientes:

- Solicitudes:
  - Se cobra por cada solicitud realizada a la API.
  - Los precios varían según el tipo de solicitud (*GET*, *POST*, *PUT*, etc.) y el plan de precios elegido.
  - Los precios cambian según el estándar utilizado para la comunicación *HTTP* o *HTTPS* los precios son más altos para *HTTPS*, los valores asociados al primer nivel de pago se encuentran en la Tabla 9.
- Almacenamiento cache (*Caching*):
  - API Gateway ofrece opciones de *caching* para mejorar el rendimiento.
  - El coste del *caching* depende de la cantidad de datos almacenados en caché y la cantidad de solicitudes realizadas a la API Para este proyecto no se ha utilizado esta característica debido a que incrementa innecesariamente los costes mensuales.

| Número de solicitudes (por mes) | Precio (por millón) |
|---|---|
| Primeros 333 millones | 3,50 USD |
| Próximos 667 millones | 3,19 USD |
| Próximos 19 mil millones | 2,71 USD |
| Más de 20 000 millones | 1,72 USD |

*Tabla 9. Costes llamados a la API Rest (febrero 2024).*

La elección de Amazon API Gateway para el desarrollo de la API del Trabajo Fin de Máster es una decisión estratégica que ofrece numerosas ventajas en términos de escalabilidad, flexibilidad, eficiencia y optimización de costes. La plataforma se presenta como la herramienta ideal para este tipo de proyectos, permitiendo al equipo de desarrollo enfocarse en la creación de una API robusta y segura.

# 3 Metodología

A lo largo de este capítulo se detalla el análisis de los requisitos, diseño, desarrollo, y unificación de la arquitectura. Comenzaremos con la identificación de los requisitos, y el diseño de la arquitectura pensando en las necesidades del problema. Acto seguido se propone y analiza en detalle la arquitectura y cada uno de sus componentes del desarrollo llevado a cabo en el trabajo. Finalmente, el capítulo concluye con la integración de los componentes del proyecto.

## 3.1 Análisis de requerimientos

En esta sección basado en la metodología agile [44], se describen los requisitos funcionales y no funcionales que motivaron al diseño e implementación de una arquitectura *web serverless* para satisfacer las necesidades teológicas de la Iglesia Bautista Reformada proporcionando a los miembros, visitantes y usuarios en general, información relevante y herramientas útiles.

### 3.1.1 Requisitos Funcionales (RF)

A continuación, se detallan los requisitos funcionales para el diseño.

- Información
  - Declaración de Fe (RF1):
    - La aplicación debe mostrar una sección con la **declaración de fe** de la iglesia.
    - Los usuarios deben poder acceder a esta información de manera rápida y sencilla.
  - Historia de la Iglesia (RF2):
    - Se debe incluir una sección que narre la **historia de la iglesia**.
    - Puede contener detalles sobre su fundación, hitos importantes y participantes en el liderazgo.
  - Actividades Próximas (RF3):
    - La aplicación debe mostrar un artículo con cada una de las **actividades próximas** de la iglesia.
    - Los usuarios podrán consultar eventos como cultos, reuniones, estudios bíblicos, etc.
- Herramientas
  - Información de Iglesias Hermanas (RF4):
    - La aplicación debe proporcionar acceso a información sobre **otras iglesias con una confesión de fe similar** en el mundo, pero principalmente dentro de España.
    - Puede incluir ubicación, horarios de culto y detalles de contacto.

- Libros de Estudio (RF5):
  - Debe haber una sección con **libros de estudio** recomendados por la iglesia y los pastores, con comentarios y recomendaciones.
  - Los usuarios podrán encontrar recursos para su crecimiento espiritual.
- Últimas Predicaciones (RF6):
  - La aplicación debe ofrecer acceso a las **últimas predicaciones** grabadas y escritas.
  - Los usuarios podrán leer, escuchar o ver los mensajes impartidos en los cultos.

### 3.1.2 Requisitos no funcionales (NRF)

A continuación, se detallan los requisitos no funcionales para el diseño.

- Rendimiento (NRF1):
  - El tiempo de respuesta del sistema debe ser inferior a 2 segundos para las operaciones críticas.
  - El sistema debe manejar al menos 1000 solicitudes concurrentes sin degradación significativa.
- Escalabilidad (NRF2):
  - El sistema debe ser capaz de escalar automáticamente según la demanda.
  - Debe manejar un aumento en la carga sin afectar el rendimiento.
- Disponibilidad (NRF3):
  - El sistema debe estar disponible cuando sea necesario (por ejemplo, 99.9% de tiempo de actividad).
  - Disponible en computadoras de escritorio y móviles.
- Facilidad de mantenimiento (NFR4):
  - El código debe ser modular y documentado.
  - Las actualizaciones y correcciones deben ser fáciles de aplicar.
  - Debe haber una separación clara entre Frontend y Backend.
- Costes (NFR5):
  - Al ser una institución sin ánimo de lucro, se debe procurar reducir los costes tanto como sea posible.

### 3.1.3 Restricciones y consideraciones

- Tiempo:
  - El proyecto debe completarse antes de la defensa en junio 2024.
  - Las restricciones temporales afectan la planificación y el desarrollo.
- Entorno:
  - La aplicación se desplegará en un entorno serverless.
  - Esto afecta la elección de tecnologías y la arquitectura.
- Separación y tecnologías:
  - Se utilizarán tecnologías como Angular para el *Frontend* y AWS Lambda para el *Backend*.

## 3.2 Estrategia para solución de requerimientos

En esta sección, considerando los requisitos funcionales y no funcionales expuestos en la sección anterior 3.1, se indica la estrategia que se siguió para diseñar la arquitectura para este trabajo y basado en el estudio de los diferentes servicios que ofrece AWS en el capítulo 2, con el objetivo de encontrar los más adecuados a las necesidades de la institución religiosa.

Para ello se toma la decisión de dividir el diseño y desarrollo en dos secciones, *frontend* y *backend*, especialmente debido a que se desplegara en un entorno serverless. Esta separación agrupara los servicios en las dos secciones y trae consigo varios beneficios, esto se logra de la siguiente manera.

### 3.2.1 Separación de Responsabilidades

La división entre el *Frontend* y el *Backend* permite una clara separación de responsabilidades. Cada parte tiene un propósito específico:

- **Frontend** (Cliente):
  - Responsable de la interfaz de usuario y la experiencia del usuario.
  - Presenta datos al usuario y recopila entradas.
  - Utiliza tecnologías como HTML, CSS y JavaScript.
  - Puede ser escalado independientemente del *Backend*.
  - Puede ser desarrollado, probado y desplegado independiente del *Backend*.
- **Backend** (Servidor):
  - Maneja la lógica institucional, la persistencia de datos.
  - Se comunica con bases de datos, servicios de almacenamiento de archivos estáticos y otros componentes.

Utiliza lenguajes como Python, Java, Node.js o C#. En nuestro caso, se seleccionó el lenguaje Java, por su independencia de plataforma, amplia adopción y difusión, su orientación a objetos y robustez, sumado a la experiencia previa adquirida en este lenguaje para el desarrollo *Backend*.

  - Puede escalar automáticamente según la demanda.
- **Escalabilidad y Costes:**

El paradigma *serverless* permite una escalabilidad automática. Los recursos se asignan dinámicamente según la carga.

Al dividir la aplicación, podemos escalar solo la parte necesaria. Por ejemplo, si el *Frontend* necesita más capacidad, solo escalamos esa parte. Y si a su vez el Frontend está alojada en un CDN, reduce mucho más la carga del servidor encargado del servicio del *Frontend* y a su vez resulta más económico los costes por la entrega de contenido por parte del servicio de Amazon CloudFront de AWS. Esto a su vez reduce los costes, ya que no pagamos por recursos subutilizados.

- **Mantenibilidad:**

La separación facilita el mantenimiento. Si hay un error en el Frontend, no afecta al Backend y viceversa así que los equipos pueden trabajar de manera independiente en cada parte facilitando las actualizaciones y correcciones.

– **Flexibilidad Tecnológica:**

Al dividir, podemos usar diferentes tecnologías para cada parte. Lo que permite utilizar Angular para el Frontend y AWS Lambda para el Backend. Esto nos permite elegir las herramientas y tecnologías de acuerdo con el caso de uso en particular para cada tarea.

– **Despliegue Continuo:**

La integración continua (CI) es esencial. Permite implementar actualizaciones de manera rápida. Podemos desplegar el Frontend y el Backend por separado sin afectar la funcionalidad.

Es por estas razones que se ha tomado la decisión de diseño y desarrollo dividiendo la arquitectura web en *Frontend* y *Backend* en un entorno *serverless* que ofrece ventajas significativas en términos de escalabilidad, mantenibilidad, costes y flexibilidad tecnológica. Es una estrategia útil y necesaria para desarrollar aplicaciones modernas en la nube.

## 3.3 Proceso de implementación

Basado en el análisis hecho en el capítulo 2 de los servicios de Amazon, se procede con el diseño de la arquitectura del trabajo. Una vez tomada la decisión de diseño para dividir la arquitectura en *Frontend* y *Backend*, es necesario identificar los servicios que corresponden a cada una de las partes y su respectiva integración.

Llegado a este punto, se indica como se llevó a cabo la fase de desarrollo, la cual se divide en dos etapas diferentes:

– Entrega de contenido web (*Frontend*): Almacenamiento y distribución la aplicación web con el modelo de desarrollo SPA hecho con el *Framework* Angular, especializado en este tipo de aplicaciones, que provee utilidades para archivos estáticos, imágenes y multimedia. También se incluye en esta fase la integración del servicio de AWS especializado en la entrega rápida de contenido.
– Funcionalidades (Backend): Componentes encargados de los servicios de la arquitectura, desarrollando principalmente bajo el paradigma *serverless*.

En la siguiente sección se describe con mayor detalle los servicios, desarrollos y configuraciones necesarias para su implementación e integración.

El desarrollo de esta fase se estructura de manera secuencial, lo que quiere decir, que se ira realizando una a una de cada pieza, terminando una para pasar a la siguiente. La metodología por etapas utilizada es la que se observa en la Ilustración 1 que se muestra a continuación:


*Ilustración 1. Metodología empleada en este trabajo.*

- Fase de análisis: Se realizó el estudio de las características de los servicios y las condiciones que debe cumplir.
- Fase de diseño: Identifica las características y pasos necesarios para abordar el trabajo final de máster.
- Fase de desarrollo: Se lleva a cabo la codificación y configuración de las fases anteriores.
- Fase de pruebas de integración: Al concluir con la fase de desarrollo se procede a realizar la integración entre los servicios centrados en *CloudFront* y *API Gateway* para la verificación de su correcto funcionamiento global de la arquitectura *web serverless*.

## 3.4 Arquitectura

En esta sección, se describe el proceso de diseño y desarrollo de la arquitectura, incluyendo las decisiones tomadas, las tecnologías y servicios utilizados para el desarrollo.

Después del estudio de los componentes necesarios para resolver los requerimientos en el capítulo 3.1 se diseñó la arquitectura para este trabajo y como resultado se obtuvo la visión general en la Ilustración 2 donde tomando como referencia la estrategia propuesta en la sección 3.2 se dividió en dos grupos para facilitar la separación de responsabilidades.

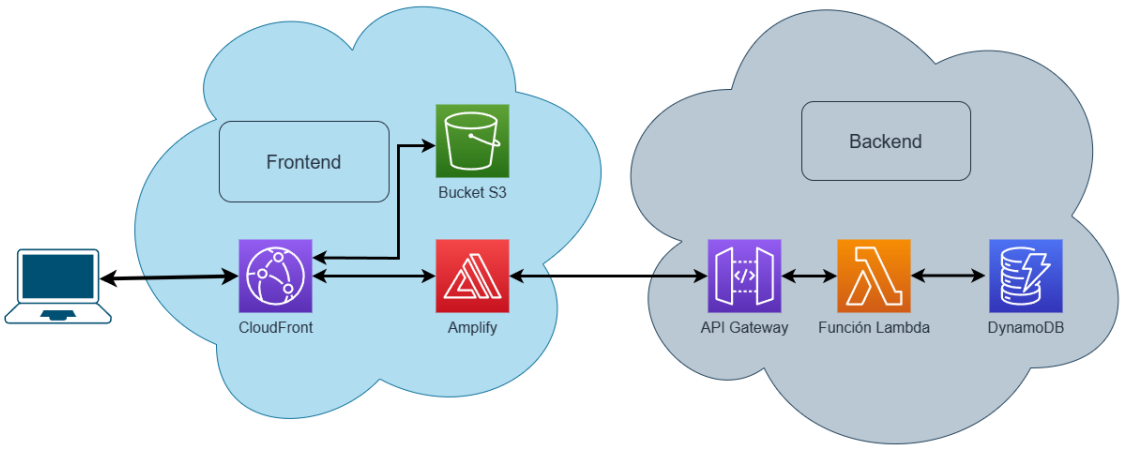


*Ilustración 2. Arquitectura web serverless en AWS.*

El diseño de la arquitectura mostrado en la Ilustración 2, se estableció así procurando satisfacer los requisitos NRF2, NRF3, NFR4 y NFR5, realizando una división en dos grupos que facilita la separación de responsabilidades, el desarrollo y posteriores actualizaciones de funcionalidades.

El primer grupo que se muestra a la izquierda de la misma ilustración contiene los servicios encargados de gestionar las peticiones de los usuarios finales a través del *Frontend*, quienes consumirán la aplicación en su navegador, ya sea desde un dispositivo móvil o un ordenador. Esta etapa se diseñó para satisfacer los requisitos funcionales RF1, RF2, RF3, RF4, RF5, RF6, de forma visual esto de describirá con mayor detalle en el capítulo 4.

Por otro lado, la segunda etapa de la arquitectura consiste en el grupo de servicios de AWS encargados de la gestión de las peticiones a los servicios *Backend* ofrecidos por las *API*s implementadas en el servicio de AWS *Lambda*. Es decir que provee toda la información necesaria para satisfacer los requisitos RF3, RF4, RF5, RF6. Esto se describirá con mayor detalle en el capítulo 4.

La división de la arquitectura en dos grupos se hizo considerando el análisis de requerimientos y el principio de separación de responsabilidades. A continuación, se explica las características de diseño tomadas tanto para el *Frontend* como el *Backend*.

### 3.4.1 Frontend

Después de la investigación de los servicios que ofrecía AWS, se seleccionaron los servicios de Amazon CloudFront, Amazon S3 y Amazon Amplify *Hosting*.

Primero se selecciona AWS Amplify como servicio para almacenar y servir la aplicación web desarrollada para la sección del *Frontend*. Este servicio permite utilizar el protocolo *HTTPS*, almacenamiento de las aplicaciones con una facturación en GBs, para el caso de este trabajo, la aplicación consume 1,27 GB con todas las dependencias instaladas, permite CI/CD, integración con otros servicios AWS para mejor rendimiento, optimización de tiempo y costes.

Se requiere de un lugar de almacenamiento de archivos estáticos de la aplicación y recursos como imágenes, archivos texto y JSON, entre otros, por lo que se selecciona el servicio de Amazon S3, que satisface esta necesidad, brindando además un servicio escalable, seguro y de bajo coste.

El último servicio escogido para la entrega del contenido del Frontend es Amazon CloudFront, este servicio es escogido debido a que brinda entre sus servicios el almacenamiento en cache en sus servidores distribuidos, lo que permite minimizar el número de peticiones a los servicios de Amazon Amplify y CloudFront, dependiendo de la configuración que tenga en su TTL (*time-to-live*) donde por defecto el tiempo para eliminar un objeto de su cache es aproximadamente después de 24 horas, y el máximo tiempo de espera corresponde a 31.536.000 segundos. Para este trabajo y por la naturaleza del proyecto, se configura bajo los niveles recomendados de Amazon CloudFront.

Este servicio brinda importantes beneficios como la mejora de rendimiento y experiencia del usuario, integración con múltiples servicios y la reducción de costes tanto del servicio de Amazon Amplify como de Amazon S3.

### 3.4.2 Backend

Para el servicio ofrecido en el *Backend* de la aplicación se compone de tres servicios: Amazon Lambda, Amazon API Gateway y Amazon DynamoDB.

El servicio de Amazon seleccionado para contener las funcionalidades de la aplicación web del lado del *Backend* es Amazon Lambda un servicio que permite ejecutar código sin necesidad de administrar servidores. Esto ofrece una gran flexibilidad y escalabilidad, ya que solo se paga por el tiempo de ejecución y se ajusta automáticamente a la demanda.

Además, Amazon Lambda soporta varios lenguajes de programación, entre ellos Java, que es el que se utilizó para desarrollar las funciones que implementan la lógica institucional de la aplicación. Este es un servicio orientado a eventos por lo que requiere su integración con un servicio que genere los eventos que desencadena la ejecución de la lógica almacenada en la función, ya sea con la modificación de elementos en un *Bucket*

S3, el servicio de Amazon SQS, o una petición hecha al servicio de Amazon API Gateway.

Como se mencionó anteriormente, se requiere de un servicio encargado capaz de recibir las peticiones hacia el *Backend* y generar los eventos que disparan la ejecución de la función Lambda razón por la cual se integra el servicio Amazon API Gateway. Un servicio que permite crear, publicar y gestionar APIs. Esto facilita la comunicación entre el *Frontend* y el *Backend*, ya que proporciona una interfaz uniforme y segura para acceder a las funciones Lambda.

Para la última sección del *Backend* se requiere de almacenamiento de la información en tablas, estas almacenaran la información de las Iglesias con una profesión de fe similar en el mundo, pero principalmente en España, la información de los libros recomendados con su respectivo resumen, y la recopilación escrita de los últimos sermones dados por los pastores a la congregación.

Basado en lo anterior se toma la elección del servicio Amazon DynamoDB, debido a su de rápida integración con el servicio de Amazon Lambda, ofrece una base de datos NoSQL de alto rendimiento y baja latencia. Esto permite almacenar y consultar los datos de la aplicación de forma eficiente y escalable, ya que se adapta al crecimiento del volumen y la velocidad de los datos. Además, Amazon DynamoDB ofrece varias ventajas como consistencia, durabilidad, disponibilidad, seguridad de los datos y la capa gratuita ofrecida para siempre según el proveedor del servicio.

En el transcurso de esta sección 3.4 Arquitectura se ha detallado el proceso de diseño y desarrollo de la arquitectura web serverless para la Iglesia Bautista Reformada, destacando la selección de servicios de AWS y la división en Frontend y Backend para una clara separación de responsabilidades. Este análisis exhaustivo sienta las bases para el siguiente capítulo, Desarrollo, donde se explorarán en detalle las etapas de implementación del proyecto. A través de este próximo capítulo, se profundizará en cómo se llevaron a cabo las decisiones de diseño, la configuración de los servicios seleccionados y la integración de las funcionalidades tanto en el Frontend como en el Backend.

# 4 Desarrollo

En este capítulo se detalla el desarrollo del diseño provisto en el capítulo 3 utilizando los servicios del proveedor *cloud* público AWS. El proceso de desarrollo descrito a continuación se detalla considerando la división de responsabilidades de la arquitectura (*Frontend-Backend*), tal y como se ha justificado previamente, empezando por el *Frontend*.

El código de la infraestructura desplegada se encuentra disponible en los repositorios públicos de la plataforma GitHub, estos se pueden acceder mediante los siguientes enlaces:

**https://github.com/Giovannisabces/ibrvtfm-alternativa**

**https://github.com/Giovannisabces/BackendServerless**

## 4.1 Desarrollo del proyecto en Angular

Para el desarrollo de la aplicación web se requiere de un *framework* web como Angular. Se muestra la arquitectura desarrollada en la Ilustración 6, que permite dos cosas importantes para el cumplimiento de los objetivos específicos: primero, al ser un *framework* diseñado para ejecutarse en el cliente (navegador del usuario) permite separar completamente la responsabilidad del *Frontend* respecto al *Backend*, segundo, permite la integración con los servicios del *Backend* a través de peticiones REST de forma ágil y flexible.

A continuación, se hará una explicación detallada de la estructura de la aplicación construida en Angular:

1. Estructura del proyecto y jerarquía de carpetas:

   Angular utiliza una estructura de proyecto predefinida que proporciona una organización clara y estable. En la carpeta raíz del proyecto, están los archivos *package.json*, *angular.json* y *tsconfig.json*, que son configuraciones esenciales para el proyecto.

   Además, Angular utiliza el concepto de módulos para organizar la funcionalidad de la aplicación. Se crea en la carpeta *app* que contiene los componentes, servicios, directivas y otros elementos necesarios para una funcionalidad especifica.

   Dentro de cada módulo, se utiliza una buena práctica en este desarrollo para crear subcarpetas como "*components*" para los componentes específicos del módulo, "*services"* para los servicios relacionados y "*models*" para definir los modelos de datos utilizados en el módulo. Lo que permite este diseño es ir desarrollando de forma incremental y siendo reutilizables los módulos en cualquier parte que se requiera de la aplicación tal como se muestra a la izquierda en la Ilustración 3.

2. Diseño de la interfaz de la aplicación:

   En cuanto al diseño de la presentación en el navegador de la aplicación web, Angular utiliza el lenguaje de marcado HTML para definir la estructura de la interfaz de usuario. Para este trabajo se desarrollaron los componentes responsables de representar esta estructura y que contienen otros componentes anidados para construir una jerarquía de elementos visuales.

Para estilizar la interfaz de usuario, se puede utilizar CSS o el preprocesador Sass. Para este trabajo se utiliza en los componentes archivos Sass debido a que permite definir estilos globales en el archivo “styles.scss”, con el objetivo de personalizar el contenido del componente en archivos separados.

Además, cada módulo cuenta con su archivo HTML, el cual contendrá el contenido que se desea mostrar en cada módulo, y el archivo “nombrecomponente.component.ts” el cual gestionará la interacción entre la interfaz de usuario y el componente (ver Ilustración 3).

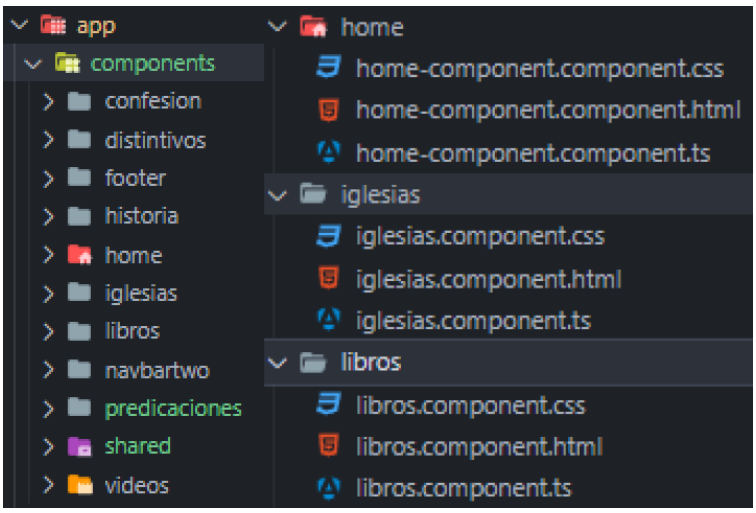


*Ilustración 3. Componentes de la aplicacion web.*

3. Enrutamiento:

Angular ofrece un sistema de enrutamiento que permite navegar entre las diferentes vistas de la aplicación. Para definir rutas utilizadas en el desarrollo se modificó el archivo *AppRoutes.ts* para agregar las rutas y asociar los componentes específicos a cada ruta. Esto facilita la navegación y la carga de componentes según la URL (ver Ilustración 4).

```
const routes: Routes = [
  {path: '', component: HomeComponentComponent},
  {path: 'historia', component: HistoriaComponent},
  {path: 'distintivos', component: DistintivosComponent},
  {path: 'confesion', component: ConfesionComponent},
  {path: 'iglesias', component: IglesiasComponent},
  {path: 'libros', component: LibrosComponent},
  {path: 'videos', component: VideosComponent},
  {path: 'predicaciones', component: PredicacionesComponent},
  {
    path: '**',
    redirectTo: "/"
  }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

*Ilustración 4. Rutas establecidas para satisfacer los requisitos funcionales.*

4. Servicios:

Los servicios dentro de Angular para este trabajo son utilizados para la entrega de información solicitada a través del servicio de Amazon *API Gateway* encargado de la gestión del *Backend*. Los servicios para este desarrollo fueron creados y almacenados en la carpeta "*services*" para posteriormente inyectarlos en los componentes que los necesiten. Esto promueve la reutilización de código y la separación de responsabilidades.

5. Gestión de solicitudes API REST:

Para gestionar el uso de solicitudes HTTP se hace uso del módulo HttpClient de Angular para realizar las solicitudes HTTP dirigidas a la API del *backend* de la arquitectura.

```
@Injectable({
  providedIn: 'root'
})
export class LocationService {

  constructor(private httpClient: HttpClient)
  { }

  public list():
  Observable<ResponseApiGateway> {
    return this.httpClient.
    get<ResponseApiGateway>("https://
    zsc3axf6q8.execute-api.us-east-1.
    amazonaws.com/Dev/locations")
  }
}
```

```
@Injectable({
  providedIn: 'root'
})
export class BookService {

  constructor(private httpClient: HttpClient)
  { }

  public list():
  Observable<ResponseApiGateway> {
    return this.httpClient.
    get<ResponseApiGateway>("https://
    zsc3axf6q8.execute-api.us-east-1.
    amazonaws.com/Dev/book");
  }
}
```

```
@Injectable({
  providedIn: 'root'
})
export class PreachService {

  constructor(private httpClient: HttpClient)
  { }

  public list():
  Observable<ResponseApiGateway> {
    return this.httpClient.
    get<ResponseApiGateway>("https://
    zsc3axf6q8.execute-api.us-east-1.
    amazonaws.com/Dev/preach");
  }
}
```

*Ilustración 5. Servicios encargados de solicitar información al API Gateway en Amazon.*

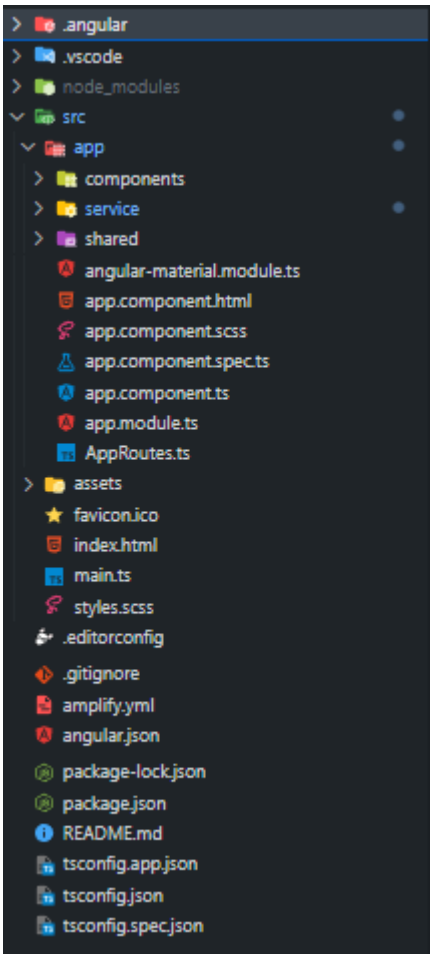


*Ilustración 6. Proyecto Angular.*

En resumen, el desarrollo de aplicaciones web en el *Framework* Angular para el trabajo final de máster implicó organizar el proyecto en una estructura de carpetas adecuada, utilizando el diseño de componentes y estilos, la implementación de enrutamiento, establecimiento de la comunicación entre componentes, utilizar servicios para compartir datos y funcionalidades de forma ágil y flexible. Lo que permite que se provea de una aplicación web ágil, interactiva y dotada con las utilidades requeridas por la institución religiosa, permitiendo seguir agregando funcionalidades según lo vayan requiriendo.

La aplicación desarrollada con la tecnología Angular se despliega en la nube utilizando los servicios seleccionados de la infraestructura precedente de AWS, lo cual permite dotar de forma y funcionalidad a la arquitectura. Para comprender completamente su funcionalidad, la forma en que atiende las solicitudes y cómo se alojan los recursos, es necesario tener un conocimiento detallado sobre su desarrollo.

A continuación, se explicarán paso a paso los servicios y herramientas utilizados, junto con su respectiva configuración y las decisiones tomadas para el despliegue completo de esta aplicación diseñada para un entorno serverless, comenzando con el servicio de Amplify.

## 4.2 Despliegue del servicio en Amplify

El despliegue del f*rontend* se ha realizado en el servicio de *hosting* Amazon Amplify. Esto se hizo a través de la integración de con los repositorios de código como GitHub o Bitbucket. Esta integración permitió que Amplify detecte automáticamente los cambios realizados en el repositorio GitHub, lo compile y despliegue automáticamente la nueva versión de la aplicación en el entorno de producción. Esto agiliza el proceso de despliegue y evita la necesidad de realizar implementaciones manuales lo que proporciona una solución al requerimiento NRF4. Además de su capacidad para crear aplicaciones escalables. Amplify ofrece un conjunto de servicios y herramientas que permiten gestionar eficientemente el crecimiento de las aplicaciones a medida que aumenta la demanda de los usuarios. Esto incluye la capacidad de ajustar automáticamente la capacidad de los servidores, así como la gestión de recursos y la distribución de carga. De esta manera, Amplify asegura que las aplicaciones puedan manejar un aumento en el tráfico sin comprometer su rendimiento para satisfacer los requerimientos NRF2 y el NRF4.

Para el despliegue de la aplicación hecha en Angular, se requiere la configuración de un archivo en formato *YAML* el cual permite realizar el despliegue de la aplicación en el servicio de Amazon Amplify indicando las distintas fases y las instrucciones que se desean ejecutar respectivamente.

Configuración para despliegue:

1. Conectar al repositorio.

Primero, con el objetivo de poder tener un CI/CD, se realizará la integración con el servicio proveedor de control de versiones GitHub, de entre las existentes (Bitbucket, GitLab y AWS codeCommit). La selección de GitHub es debido a la familiariadad y experiencia previa del autor con esta plataforma, lo que agiliza el desarrollo del proyecto y garantiza una mejor calidad del trabajo realizado.

Una vez autorizada la consola de Amplify con GitHub, Amplify obtiene un token de acceso del proveedor del repositorio, el token obtenido por GitHub tiene una duración de 8 horas antes de caducar y adicionalmente GitHub da un token de actualización para ir renovando el primer token de acceso al repositorio con el proyecto en la rama seleccionada. Amplify accede a su repositorio utilizando claves de implementación instaladas únicamente en un repositorio específico. El repositorio se encuentra en el siguiente enlace.

| Giovannisabces/ibrvtfm-alternativa (github.com) |
|---|

2. Archivo de configuración despliegue.

Una vez hecha la integración con GitHub, se procede a la configuración del archivo de configuración para el despliegue. Al observar la Ilustración 7 que contiene el archivo de configuración para la construcción de la aplicación en Amazon Amplify. Este archivo contiene las fases de ejecución "*phases*" (pre-construcción y construcción), la compilación del proyecto "*artifacts*", y la ruta en *cache*,

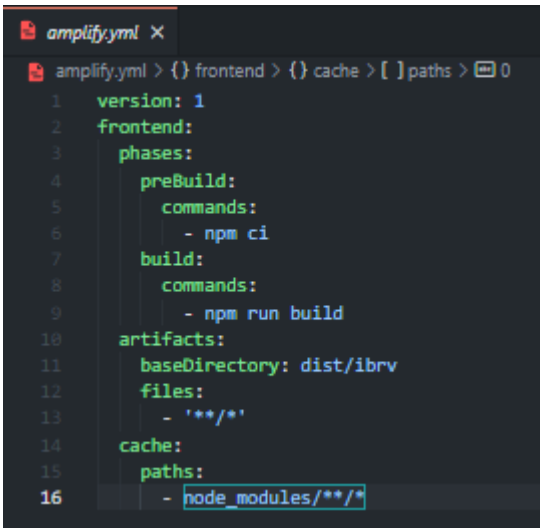


*Ilustración 7. Plantilla para construcción aplicación web.*

Se recomienda verificar el directorio de salida de compilación y los comandos de compilación. Una vez verificado su contenido que corresponda a la instalación de dependencias en Node.js y la construcción del proyecto, se procede a guardar la configuración de compilación en el repositorio. Para este trabajo final de máster, se ha añadido el archivo "amplify.yml" a la raíz del repositorio, como se puede observar en la Ilustración 6 del contenido de la Ilustración 7.

3. Guardar e implementar

Antes de que se guardara e implementara se siguió la recomendación de revisar la configuración de compilación, la correcta selección del *framework* configurado para el despliegue desde la consola de Amazon Amplify (véase Ilustración 8).

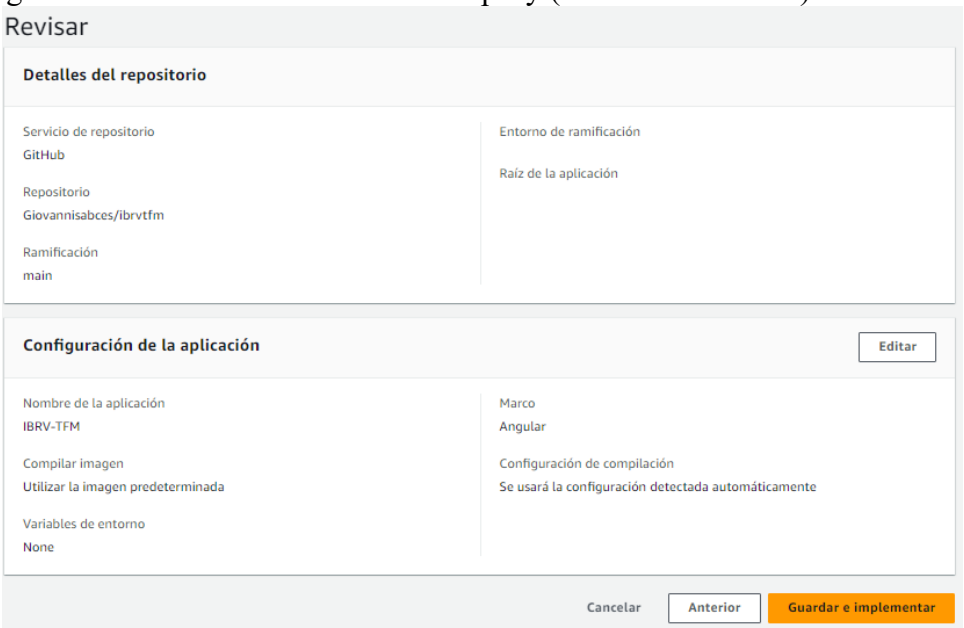


*Ilustración 8. Revisión final implementación Amplify.*

Al elegir “Guardar e implementar”, se procede a implementar la aplicación (véase Ilustración 9). Esta aplicación se integrará posteriormente con el servicio de CloudFront para la entrega de contenido (CDN) en la red global de AWS. La compilación de la interfaz generalmente tarda de 1 a 2 minutos, aunque puede variar dependiendo del tamaño de la aplicación. En el caso de la aplicación desarrollada para este trabajo final de máster, se encontraba dentro de ese rango de tiempo (1 minuto).

El proceso de implementación consta de tres fases:

1. **Aprovisionamiento:** Se realiza el aprovisionamiento del entorno de compilación mediante el uso de una imagen de Docker en un host con 4 vCPU y 7 GB de memoria. Cada compilación se ejecuta en su propia instancia de host, lo cual asegura que todos los recursos estén debidamente aislados de forma segura (véase Ilustración 9). Se tuvo en cuenta esta característica para el NFR2 debido a que se ha genera una imagen Docker disponible para la escalabilidad horizontal automática gestionada por Amplify.

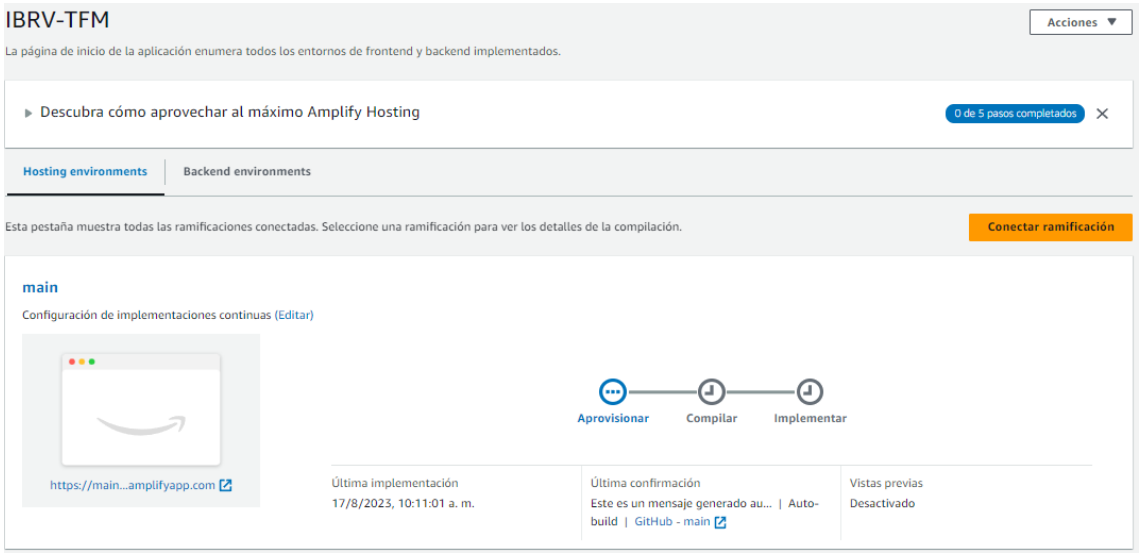


*Ilustración 9. Primer paso: Aprovisionamiento*

2. **Compilación:** La fase de compilación comprende tres etapas en las que se llevan a cabo las siguientes actividades: configuración (se realiza la clonación del repositorio en un contenedor), implementación del *Backend* (se ejecuta la CLI de Amplify para implementar los recursos del *Backend*) y compilación del *Frontend* (se generan los artefactos del *Frontend*) (véase la Ilustración 10).

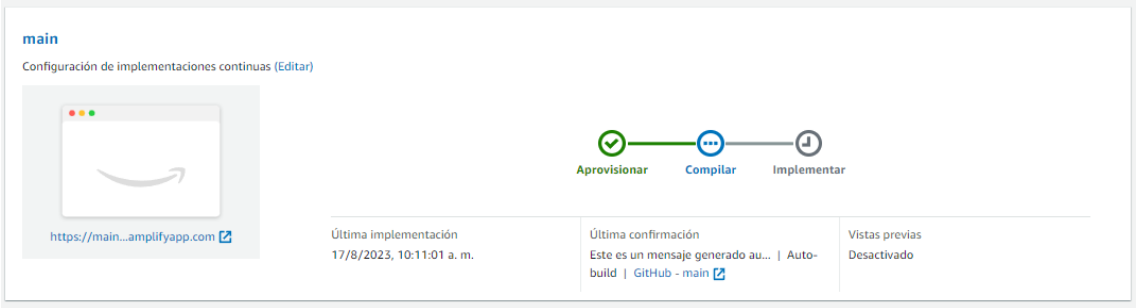


*Ilustración 10. Segundo paso: Compilación.*

3. **Implementación:** Cuando finaliza el proceso de compilación, todos los elementos generados se despliegan en un entorno de alojamiento administrado por Amplify Hosting. Cada despliegue se realiza de forma atómica, lo que significa que se eliminan los períodos de mantenimiento y se garantiza que la aplicación web solo se actualice una vez que se haya completado todo el proceso de implementación (véase Ilustración 11).


*Ilustración 11. Tercer paso: implementación.*

Aspectos importantes que considerar en la imagen anterior:

4. Dominio y URL: Amplify ofrece un dominio sin coste adicional, además de proporcionar la conveniencia de mostrar el enlace de la aplicación en la consola.
5. Fecha de implementación: Proporciona un indicador de fecha y hora de la última actualización.
6. Ultima confirmación y repositorio: Muestra con claridad con que repositorio se está realizando el despliegue y la rama sobre la que se está trabajando.

Para verificar el correcto funcionamiento del despliegue realizado, se accede a la URL que proporciona la plataforma de AWS Amplify, donde su contenido se muestra en la Ilustración 12 En la parte superior de la imagen se puede observar la URL y en la parte inferior el contenido del desarrollo hecho en Angular y desplegado en el servicio de Amplify.

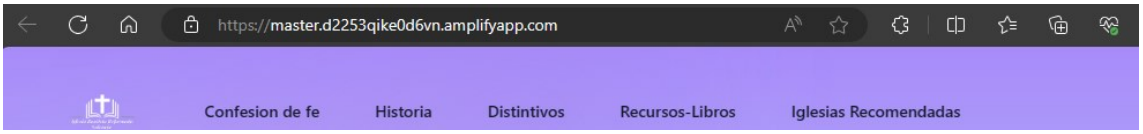


*Ilustración 12. Verificación de funcionamiento despliegue en AWS Amplify.*

A su vez, en la Ilustración 13 se muestra en la parte izquierda la versión móvil y en la derecha la versión de escritorio que aborda todo el requerimiento NRF3.

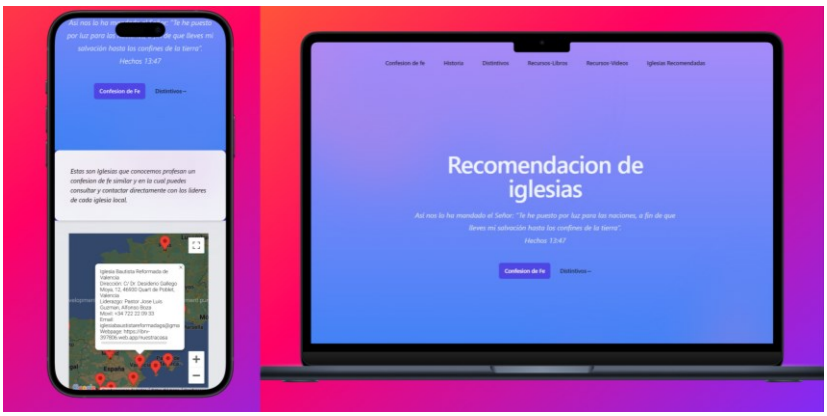


*Ilustración 13. Implementación final tanto en móvil como web.*

El resultado final, cumpliendo con todos los requerimientos funcionales se puede apreciar en la Ilustración 14. Debe tener en cuenta que las capturas se tomaron en formato de dispositivo móvil debido a que ocupa menos espacio, sin embargo, la aplicación web *serverless* funciona para computadoras de escritorio también.

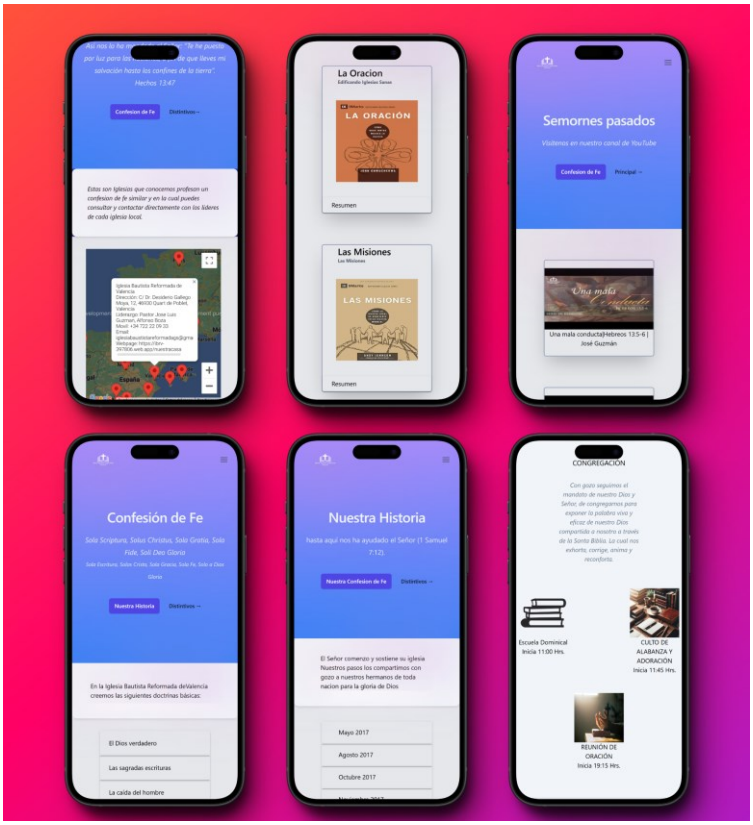


*Ilustración 14. Aplicación web final.*

## 4.3 Distribución CloudFront (CDN)

Continuando con el desarrollo de la arquitectura *serverless* para este trabajo final de máster, el siguiente paso consiste en la integración del despliegue realizado en Amplify con el servicio de Amazon CloudFront. Recordemos que éste es un servicio de red de entrega de contenido (CDN, por sus siglas en inglés) que ofrece múltiples ventajas. Tal y como se mencionó en el capítulo 2, Amazon CloudFront utiliza una red global conformada por más de 450 puntos de presencia y 13 cachés de regiones periféricas en más de 90 ciudades de 48 países [45]. Esto permite una entrega de contenido mucho más rápida utilizando su red de fibra óptica dedicada entre CloudFront y los servicios ofrecidos por AWS.

Para este trabajo se acotó el proyecto a un ámbito más local, orientado específicamente a España. Por lo tanto, se configurará para que este servicio esté disponible en Europa y Norte América. Además, es importante considerar que los costes para la región de Europa y Norte América son mucho más económicos (véase Ilustración 15). También es

importante mencionar que los costes relacionados con el servicio de Amplify se verán reducidos, ya que las solicitudes serán atendidas con mayor frecuencia por CloudFront.

**Transferencia saliente de datos regional a Internet (por GB)**

| Al mes | Estados Unidos, México y Canadá | Europa e Israel | Sudáfrica, Kenia y Oriente Medio | América del Sur | Japón | Australia y Nueva Zelanda | Hong Kong, Indonesia, Filipinas, Singapur, Corea del Sur, Taiwán, Tailandia, Malasia y Vietnam | India |
|---|---|---|---|---|---|---|---|---|
| Primeros 10 TB | 0,085 USD | 0,085 USD | 0,110 USD | 0,110 USD | 0,114 USD | 0,114 USD | 0,120 USD | 0,109 USD |
| Siguientes 40 TB | 0,080 USD | 0,080 USD | 0,105 USD | 0,105 USD | 0,089 USD | 0,098 USD | 0,100 USD | 0,085 USD |
| Siguientes 100 TB | 0,060 USD | 0,060 USD | 0,090 USD | 0,090 USD | 0,086 USD | 0,094 USD | 0,095 USD | 0,082 USD |
| Siguientes 350 TB | 0,040 USD | 0,040 USD | 0,080 USD | 0,080 USD | 0,084 USD | 0,092 USD | 0,090 USD | 0,080 USD |
| Siguientes 524 TB | 0,030 USD | 0,030 USD | 0,060 USD | 0,060 USD | 0,080 USD | 0,090 USD | 0,080 USD | 0,078 USD |
| Siguientes 4 PB | 0,025 USD | 0,025 USD | 0,050 USD | 0,050 USD | 0,070 USD | 0,085 USD | 0,070 USD | 0,075 USD |
| Más de 5 PB | 0,020 USD | 0,020 USD | 0,040 USD | 0,040 USD | 0,060 USD | 0,080 USD | 0,060 USD | 0,072 USD |

*Ilustración 15. Precios Cloud Front (agosto 2023) Fuente* [46].

Para el proceso de integración entre los servicios de Amazon Amplify y CloudFront, de acuerdo con el diseño de la arquitectura (véase Ilustración 2) ubicado en la zona izquierda designada para el *Frontend*, se utilizará la URL generada desde el servicio de Amazon Amplify Hosting (véase Ilustración 12) para integrarlo como uno de los orígenes de la configuración de Amazon CloudFront.

Para la configuración del servicio de Amazon CloudFront se deben proporcionar los siguientes datos:

1. Dominio de origen: Se utilizo la URL generada desde el servicio de Amazon Amplify Hosting (ver Ilustración 16).
2. Protocolo de respuesta: es necesario especificar a CloudFront el tipo de respuesta que dará a las peticiones que reciba y para ello hay que tener en cuenta las siguientes configuraciones:
   - Protocolo: En la consola se observan los dos tipos de protocolos a elegir (ver Ilustración 16), HTTP y HTTPS. Para este trabajo se ha seleccionado la opción de HTTPS debido a que Amazon CloudFront proporciona de forma gratuita tanto un dominio y su respectivo certificado de seguridad. Además, utilizar HTTPS proporciona una serie de ventajas frente al protocolo HTTP:
     - Encripta las comunicaciones entre el cliente y el servidor brindando una capa más de seguridad e integridad en ambos sentidos.
     - Evita penalizaciones por seguridad en el SEO por parte de buscadores como Google.
     - La identificación positiva por parte de los navegadores, debido a que ellos relacionan el contenido con HTTP como no seguros.
     - Compatibilidad con nuevas tecnologías debido a que solo funcionan en entornos seguros HTTPS debido a las restricciones de seguridad del navegador.
     - Permite la autenticación del servidor debido a que los certificados generados para HTTPS son emitidos por autoridades de certificación confiables. Esto permite a los usuarios verificar la autenticidad del servidor al que están accediendo, lo que ayuda a prevenir ataques de phishing y garantiza que están interactuando con el sitio web correcto.
   - Puerto: En este trabajo, se ha dejado la opción por defecto del puerto 443 para comunicaciones por HTTPS ya que no entra en conflicto con otras configuraciones hechas en esta distribución con otras aplicaciones y/o servicios (ver Ilustración 16).

- Protocolo SSL: al igual que el anterior se ha dejado por defecto, quedando seleccionado TLSv1.2.

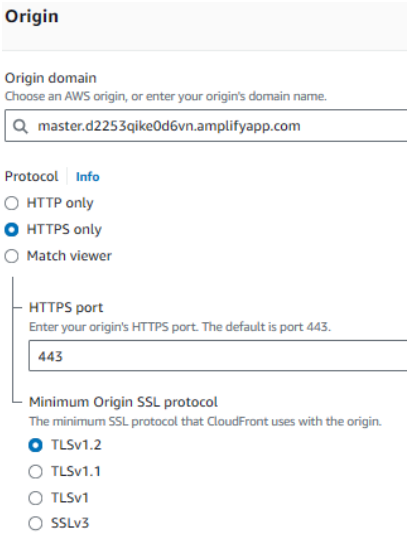


*Ilustración 16. Configuración URL origen, puerto y protocolo SSL de respuesta.*

En la Ilustración 17 se continúa detallando las configuraciones generales de la distribución CloudFront:

3. La compresión de Objetos: Se ha configurado la compresión de objetos, para obtener un mejor rendimiento y reduciendo el impacto en el conteo de la tarifa de CloudFront, reduciendo costes por parte del servicio de CloudFront y el consumo de ancho de banda por parte del cliente final, así que es importante también mencionarlo.
4. Protocolo de política del espectador: se ha seleccionado la opción que permite redireccionar peticiones HTTP a HTTPS considerando que CloudFront de forma gratuita entrega un certificado asociado a su dominio y lo configura de forma automática.
5. Selección de métodos permitidos: Se configuran los métodos permitidos para este trabajo seleccionando la última opción considerando que permite mayor flexibilidad a la hora del desarrollo y puesta en producción, así como en el futuro poder seguir agregando funcionalidades sin configuraciones adicionales en CloudFront.
6. Restricción de acceso al espectador: Se ha seleccionado que no haya restricciones debido a que no se hizo un desarrollo orientado a las restricciones de acceso relacionadas por URLs o cookies.

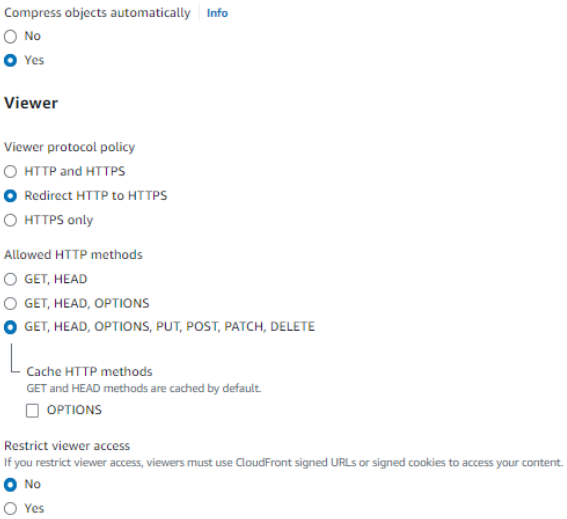


*Ilustración 17. Configuración CloudFront políticas de respuesta.*

Seguidamente aparece la configuración de cache (véase Ilustración 18, donde se ha seleccionado la opción por defecto.

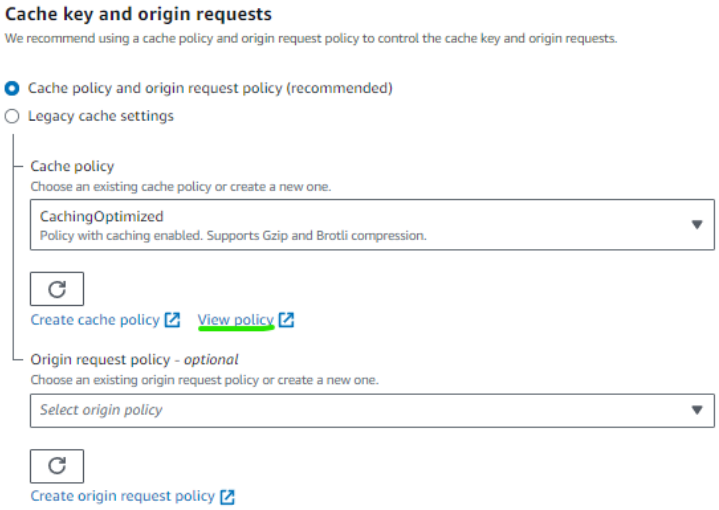


*Ilustración 18. Configuración de cache.*

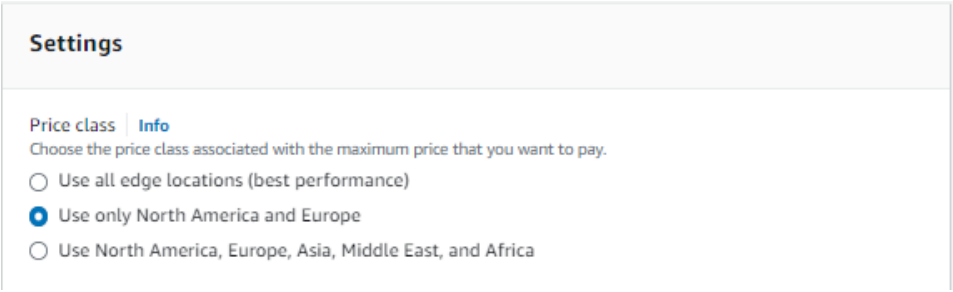


*Ilustración 19. Selección de región.*

Algo importante que se ha tenido en cuenta, como ya se mencionó previamente durante el análisis de los costes de CloudFront, es que se seleccionó la opción de utilizar únicamente las ubicaciones de Norteamérica y Europa para la distribución de este trabajo (ver Ilustración 19). Esta elección se realizó considerando que estas regiones son las más económicas y abarcan la zona objetivo correspondiente a España. Importante considerar en la Ilustración 20:

- ARN: denominado como nombre de recurso de Amazon, su función es identificar de forma única los recursos de AWS. Se dividen en campos entre los que se destacan el servicio, el número de cuenta o el nombre de la función.
- Nombre de dominio de distribución: es un identificador único que se utiliza para acceder a los recursos almacenados en el CDN de CloudFront. Cada distribución de dominio tiene un nombre único que se asigna al configurar y crear la distribución en la consola de administración de AWS. Este nombre se utiliza para acceder a los recursos, como archivos estáticos, imágenes o contenido dinámico, a través de la CDN de CloudFront.

Cuando termina el proceso de despliegue de la distribución de CloudFront, tal como se muestra en la Ilustración 20 donde se actualiza el estado y cambia de *Deploying* a última fecha de modificada entregando el mes, día, año, hora y zona horaria.

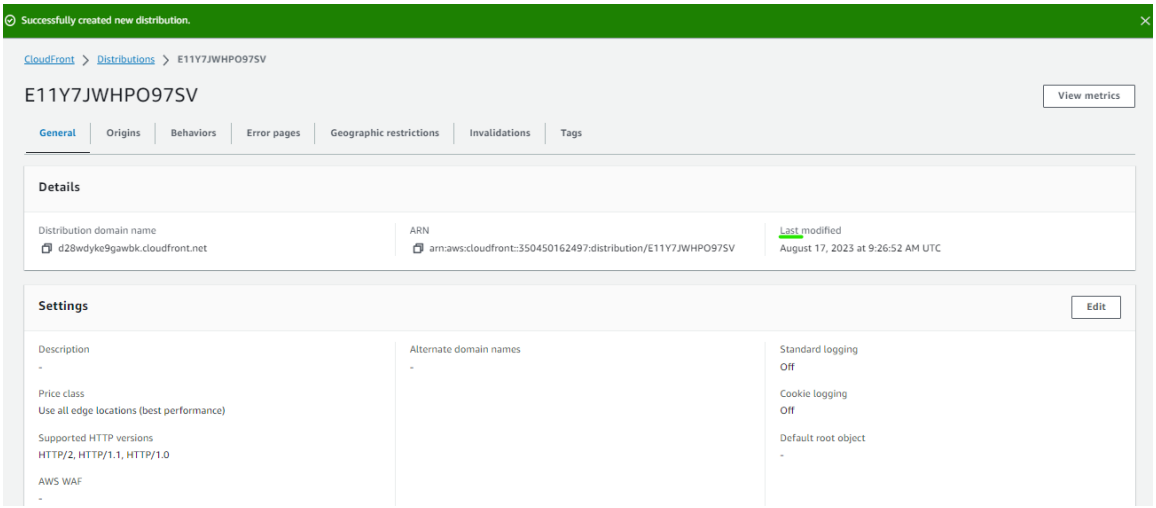


*Ilustración 20. Despliegue distribución de CloudFront.*

## 4.4 Creación del Bucket S3.

Para la arquitectura de este proyecto se requiere de un servicio de almacenamiento para archivos de texto e imágenes con el objetivo de minimizar el tamaño de la aplicación entregada por el servicio Amazon Amplify, entregando el contenido bajo demanda. Para este proyecto se ha creado un *bucket* en el servicio de Amazon S3 que no es accesible públicamente. Se ha configurado así debido a que solo se desea acceder al *bucket* a través del servicio Amazon CloudFront, lo que permite que los elementos solicitados a los *bucket* queden almacenados en la cache, reduciendo el número de solicitudes hacia el *bucket* y reduciendo los costes evitando el cargo por exponer los elementos de forma pública a internet.

En la Ilustración 21 se muestra como utilizando la herramienta SAM (Serverless Application Model), se definió una plantilla para la creación del *Bucket* S3 llamado *recursostfm* en la región *us-east-1*. Dentro de la plantilla de SAM se configuró la política *Retain* que no permite que se elimine el *bucket* S3 al eliminar el *stack* asociado a la plantilla, además esto permite poder realizar actualizaciones al *stack* según se requiera.

```
  RecursosTFM:
    Type: AWS::S3::Bucket
    DeletionPolicy: Retain
    Properties:
      BucketName: recursostfm
      Tags:
      - Key: stage
        Value: beta
      - Key: purpose
        Value: testing
Outputs:
  RecursosTFM:
    Description: The ARN of ExampleS3OutpostsBucket
    Value:
      Ref: RecursosTFM
```

*Ilustración 21. Creación Bucket S3.*

Una vez creado el *bucket* en S3, se crearon las carpetas encargadas de alojar los elementos necesarios para el funcionamiento de la aplicación web (véase Ilustración 22) para facilitar así la organización y descarga de los elementos según la necesidad del usuario.

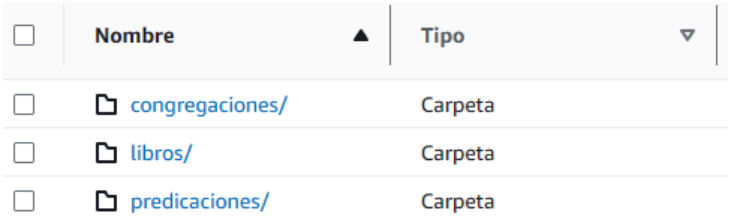


*Ilustración 22. Recursos almacenados en el Bucket S3.*

La carga a los recursos en las carpetas que se muestran en la Ilustración 22, se ha hecho manualmente. Se han seleccionado las imágenes optimizadas en su formato (jpg a webp) en la carpeta congregaciones. En la carpeta libros se subió manualmente las imágenes, y se almaceno la base de datos el URL completo hasta el recurso, con su título, serie y resumen entre otros para que se pueda solicitar según lo requiera el usuario. Por último, en la carpeta de predicaciones se cargó los archivos también de forma manual para ser consultados también según lo requiera el usuario.

## 4.5 Creación de la tabla en DynamoDB

Para que la función Lambda pueda consultar los datos relacionados con cada una de las iglesias recomendadas, dar el listado de los libros recomendados y obtener el contenido de las predicaciones, deben crearse tablas que contengan dicha información, así como definir cómo acceder a ellas a través de S3. Por lo tanto, para lograr este objetivo se va a utilizar el servicio de Amazon DynamoDB que consiste en una base de datos NoSQL. DynamoDB es un servicio de base de datos completamente administrado y altamente escalable que ofrece una estructura de almacenamiento de clave-valor flexible. Al crear una tabla en DynamoDB, debemos especificar una clave de partición primaria. Esta clave es utilizada para distribuir los datos en diferentes particiones dentro de la base de datos, permitiendo una distribución eficiente y equilibrada de la carga.

Continuando con las decisiones tomadas para la configuración de las tablas, se seleccionó para todo el tipo “eventualmente consistente para lectura y escritura”, lo que implica que habrá un tiempo de retardo para que todos los cambios se propaguen completamente a través de todos los nodos que almacenan los datos en DynamoDB. Esto

no es un inconveniente debido a que la acción más recurrente será de lectura y no de escritura.

En la configuración de DynamoDB se han aprovisionado las unidades de capacidad de lectura (RCU) y las unidades de escritura (WCU) tal como se muestra en la Tabla 10. Estas configuraciones se han seleccionado cuidadosamente para controlar los costes del servicio teniendo como principal objetivo aprovechar la ventaja de la capa gratuita de DynamoDB que no expira y que provee de 25 RCU y 25 WCU para la base de datos del proyecto.

| Tabla | Operación | Aprovisionamiento |
|---|---|---|
| location | RCU | 5 |
| | WCU | 5 |
| book | RCU | 10 |
| | WCU | 15 |
| preach | RCU | 10 |
| | WCU | 5 |

*Tabla 10. Aprovisionamiento tablas DynamoDB.*

Se ha configurado la tabla de *location* con un menor número de RCU y WCU debido a que se estima que la velocidad con la que va a ir creciendo va a ser menor con respecto a las otras dos tablas y a su vez con un menor número de acceso. Por otra parte, la tabla que más se le han configurado WCU ha sido la tabla de *book* debido a que por elemento es la que más información almacena con respecto a las otras dos. Es por esta misma razón que se configura la misma capacidad de lectura que la tabla *preach*. Esta última tabla se configura con 5 WCU debido a que los elementos almacenados en la base de datos tendrán información resumida, así como la ubicación de donde se encuentra la mayor parte de la información, que se decidió almacenarla en S3 para que el usuario acceda a todo el contenido según lo vaya requiriendo.

Es importante mencionar que las WCU se consumen con mayor facilidad debido a que una WCU equivale a 1 KB escrito mientras que para consumir una unidad de lectura se requiere de 4 KB, todo esto mediante el método estándar (con consistencia eventual). Para el caso de este trabajo, las operaciones de escritura no van a ser tan frecuentes en comparación con las operaciones de lectura por lo que la plantilla utilizada para la creación de las tablas es la que se observa en Ilustración 23.

```
location:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: location
    AttributeDefinitions:
      - AttributeName: id
        AttributeType: 'N'
    KeySchema:
      - AttributeName: id
        KeyType: HASH
    BillingMode: PROVISIONED
    ProvisionedThroughput:
      ReadCapacityUnits: 5
      WriteCapacityUnits: 5
```

```
books:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: book
    AttributeDefinitions:
      - AttributeName: id
        AttributeType: 'N'
      - AttributeName: serie
        AttributeType: 'S'
    KeySchema:
      - AttributeName: id
        KeyType: HASH
      - AttributeName: serie
        KeyType: RANGE
    BillingMode: PROVISIONED
    ProvisionedThroughput:
      ReadCapacityUnits: 10
      WriteCapacityUnits: 15
```

```
preach:
  Type: AWS::DynamoDB::Table
  Properties:
    TableName: preach
    AttributeDefinitions:
      - AttributeName: date
        AttributeType: 'S'
      - AttributeName: serie
        AttributeType: 'S'
    KeySchema:
      - AttributeName: date
        KeyType: HASH
      - AttributeName: serie
        KeyType: RANGE
    BillingMode: PROVISIONED
    ProvisionedThroughput:
      ReadCapacityUnits: 10
      WriteCapacityUnits: 5
```

*Ilustración 23. Configuración tablas DynamoDB Backend.*

En la plantilla mostrada en la Ilustración 23 se ha configurado los campos principales de las tablas. Estos campos son Clave de partición y clave de ordenación(opcional); la clave de partición por tabla son las siguientes: "*location*" es el campo (id), al igual que para "*book*", para la tabla "*preach*" es el campo (date).

En el caso de las claves de ordenación se definieron solamente para dos tablas con el mismo nombre de campo (serie) tanto en la tabla "*book*" como "*preach*". Para este trabajo no se ha configurado GSI (Global Secondary Index) ni LSI (Local Secondary Index), debido a que no se encuentran dentro de la capa gratuita provista por AWS.

Sin embargo, como se había comentado previamente, al ser DynamoDB una base de datos flexible, en la plantilla no se define necesariamente la estructura de cada una de las tablas, sino que estas se especifican en la función de java, esto se muestra más a detalle en la sección 4.6. Una vez que se finalizó la creación de las tablas, estas quedaron como se observa en la Ilustración 24.

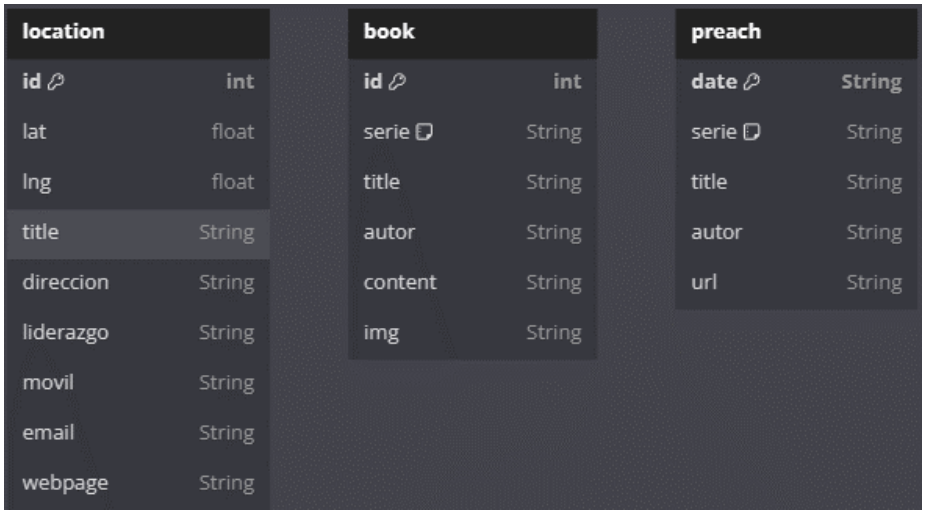


*Ilustración 24. Tablas almacenadas en DynamoDB.*

De acuerdo con lo expuesto en la Ilustración 24, La tabla "*location*" es la encargada de administrar al Frontend toda la información de cada una de las iglesias almacenadas que se considero tiene una confesión de fe similar, proveyendo de la ubicación (para Google ubicar el marcador en un mapa de Google Maps dentro de la página) con los campos (lat) y (lng), el titulo corresponde al nombre de la iglesia, se indica su dirección, sus lideres, número móvil si lo tienen, dirección de correo y una página web en caso de que la tengan.

Para la tabla de "*book*" se almacena la información, de la serie a la que pertenece el libro, debido a que hay editoriales que lanzan un conjunto de libros con el fin de abordar temas en específicos. Además, se indica el nombre del libro, el autor, el contenido y la URL de la imagen almacenada en el *bucket* S3, para ser cargada solo cuando la necesite el usuario.

Por último, está la tabla "*preach*", donde se indica la fecha en la que fue hecha la predicación, la serie, en este caso se refiere a que libro de la biblia pertenece (de entre los 66 en total) se indica el título, el autor y por último el URL del archivo almacenado en un *bucket* S3 que será enviado al cliente con todo el contenido escrito de la predicación para su estudio y análisis.

En resumen, esta configuración de DynamoDB procura satisfacer las necesidades de este trabajo utilizando la capa gratuita para siempre que provee Amazon, se toma en

cuenta esto para el ajuste de la capacidad de lectura y escritura al igual que el almacenamiento de elementos. Adicional a esto, DynamoDB permite tener tablas dinámicas, lo que permite definir las tablas desde la función *lambda* con Java con la posibilidad de futuras actualizaciones en la estructura sin tener que rehacer o migrar todas las tablas para agregar más información o atributos que provean mayor precisión.

## 4.6 Desarrollo y despliegue función Lambda con java.

En esta sección se detalla el proceso de implementación de las funcionalidades RF4, RF5 y RF6, en la arquitectura web serverless utilizando el lenguaje Java. Para este propósito, se empleó el AWS SAM *template*(plantilla).

La plantilla del conjunto de herramientas AWS SAM define las características de las funciones Lambda encargadas de atender las peticiones en el Backend. A continuación (ver Ilustración 25) se presenta un resumen que contiene las características de configuración de las funciones Lambda.

```
BookFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: BookFunction
    Handler: book.App::handleRequest
    Runtime: java17
    FunctionName: BookFunction
    Architectures:
      - arm64
    MemorySize: 512
    Events:
      ApiEvents:
        Type: Api
        Properties:
          Path: /book
          Method: GET
          RestApiId: !Ref ServiciosApiGateway
    Policies:
      - DynamoDBCrudPolicy:
          TableName: !Ref books
    Environment:
      Variables:
        BOOKS_TABLE_NAME: !Ref books
        BOOKS_TABLE_ARN: !GetAtt books.Arn
```

```
LocationFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: LocationFunction
    Handler: location.App::handleRequest
    Runtime: java17
    FunctionName: LocationFunction
    Architectures:
      - arm64
    MemorySize: 512
    Events:
      ApiEvents:
        Type: Api
        Properties:
          Path: /location
          Method: GET
          RestApiId: !Ref ServiciosApiGateway
    Policies:
      - DynamoDBCrudPolicy:
          TableName: location
    Environment:
      Variables:
        BOOKS_TABLE_NAME: !Ref location
        BOOKS_TABLE_ARN: !GetAtt location.Arn
```

*Ilustración 25. Archivo de configuración y despliegue AWS SAM.*

Dentro de la plantilla se especificó la ruta y el nombre de la función que se desea ejecutar dentro del archivo compilado en *bytecode* (instrucciones en unos y ceros) con extensión .JAR. Este compilado es generado a través del JDK (*Java Development Kit*) listo para pasarse a un entorno de ejecución en otras palabras un entorno donde este instalado la JVM (*Java virtual Machine*), para realizar la tarea de interpretar el *bytecode* que contiene el archivo compilado .JAR y ejecutarlo.

También se indica el entorno de ejecución (*runtime*) java17 como se explicó previamente, con una la versión más estable y soporte de larga duración que existía al momento de realizar este trabajo, que es la versión LTS (*Long Term Support*).

Durante el desarrollo y despliegue de la aplicación se experimentó con dos entornos de ejecución; con la versión JDK 11 y JDK 17, la versión JDK 11 tardaba aproximadamente 15 segundos con el arranque en frio, y a la versión 17 le toma aproximadamente 5 a 7 segundos siendo aproximadamente 400 a 500 milisegundos el tiempo de arranque en frio y los otros 200 milisegundos en la ejecución del código.

Adicional de esto, se tomó la decisión de utilizar la arquitectura arm64. Esta arquitectura presenta un tiempo de arranque similar al anteriormente mencionado, con un arranque en frio de 400 a 500 milisegundo. Además, ofrece una mayor eficiencia en el consumo de energía, ya que se basa en una arquitectura de diseño RISC (*Reduced Instruction Set Computing*). Esta arquitectura es más eficiente, pero no tan potente en los cálculos complejos como X86_64 [47].

Sin embargo, en las pruebas realizadas no fue posible apreciar una diferencia en el rendimiento. Por lo tanto, en cuanto a este aspecto, fue indistinto utilizar una arquitectura u otra. No obstante, si se obtiene el beneficio de utilizar una arquitectura más sostenible, al utilizar los procesadores con arquitectura ARM.

La configuración de la memoria de 512 MB se hizo evaluando el consumo de las funciones (aproximadamente 160 MB) en las pruebas al igual que el tiempo de ejecución entre 180 ms y 300 ms (milisegundos) contando con el tiempo de arranque cada vez que se inicializa un nuevo contenedor tarda alrededor de entre 5.5 s y 6.3 s (segundos), teniendo en cuenta que esto puede y se verá afectado con el paso del tiempo conforme vaya aumentado la cantidad de información almacenada en la base de datos.

Para las funciones desarrolladas en este trabajo, se requirió configurar algunos parámetros adicionales: *Events* y *Policies*. Ambas características son necesarias para el correcto funcionamiento de la configuración comienza con la definición del tipo de evento que desencadena la función. En este caso, se trata de un evento generado desde una API REST. Se le especificó el tipo, la ruta (*path)* que corresponde al servicio que se desea consumir y el método HTTP/HTTPS al que atenderá la petición.

Para este caso, es GET, por lo que la configuración de estos parámetros es fundamental para la integración con el servicio de API Gateway, el cual se describe en el capítulo siguiente.

Luego, se configura las políticas de seguridad necesarias para tener acceso de lectura y escritura a su respectiva tabla creada y almacenada en DynamoDB (ver Ilustración 26).


*Ilustración 26. Integración entre AWS Lambda y Base de datos.*

En esta plantilla de configuración (ver Ilustración 25), se especifican los parámetros necesarios para la función Lambda, como las tablas de la base de datos a utilizar y se define la configuración del API Gateway tal como se muestra en Ilustración 26.

Por otro lado, como se mencionó en la sección anterior 4.5, DynamoDB es una base de datos de estructura flexible. Por lo tanto, la estructura de la tabla se definió a través de una clase hecha en Java que utiliza un ORM (*Object Relational Mapping*). Esta clase contiene la estructura específica para cada tabla y se encargue de gestionar la interacción con los elementos de cada tabla (ver Ilustración 27).

El mapeo relacional de objetos, u ORM, es una técnica de programación que permite convertir datos entre el sistema de tipos utilizado en un lenguaje de programación

orientado a objetos y una base de datos. Para este trabajo, se ha utilizado el ORM que provee AWS para el lenguaje Java y la interacción con el servicio DynamoDB.

Es importante mencionar que se creó una clase para cada tabla, para poder interactuar con la información almacenada en base de datos y así abordar los requisitos funcionales RF4, RF5 y RF6, que necesitaban de un servicio *Backend* (ver Ilustración 27).

```
@DynamoDBTable(tableName = "location")
public class Location {
    @DynamoDBHashKey
    private int id;
    @DynamoDBAttribute
    private float lat;
    @DynamoDBAttribute
    private float lng;
    @DynamoDBAttribute
    private String title;
    @DynamoDBAttribute
    private String direccion;
    @DynamoDBAttribute
    private String liderazgo;
    @DynamoDBAttribute
    private String movil;
    @DynamoDBAttribute
    private String email;
    @DynamoDBAttribute
    private String webpage;
```

```
@DynamoDBTable(tableName = "book")
public class Book {
    @DynamoDBHashKey
    private int id;
    @DynamoDBAttribute
    private String title;
    @DynamoDBAttribute
    private String autor;
    @DynamoDBIndexRangeKey
    private String serie;
    @DynamoDBAttribute
    private String content;
    @DynamoDBAttribute
    private String img;
```

```
@DynamoDBTable(tableName = "preach")
public class Preach {
    @DynamoDBHashKey
    private String date;
    @DynamoDBAttribute
    private String title;
    @DynamoDBAttribute
    private String autor;
    @DynamoDBIndexRangeKey
    private String serie;
    @DynamoDBAttribute
    private String url;
```

*Ilustración 27. ORM utilizado para la interacción con las tablas en DynamoDB.*

El requisito RF4 exige que la aplicación web provea acceso a información de otras iglesias con una confesión de fe similar en el mundo, especialmente dentro de España. Debe proveer la geolocalización para ubicarla en un sistema de GPS, o en un mapa de Google con el servicio de Google Maps, el nombre de la iglesia, su dirección, los lideres y su información de contacto.

Esta problemática se abordó con la "*LocationFunction*". Esta función se encarga de leer la tabla "*location*" donde se almacena cada una de las iglesias y la convierte en un archivo JSON. Este archivo se entrega al *frontend* de la aplicación web, donde se encuentra el usuario final. El *frontend* gestiona la información para posicionar las iglesias en el mapa de Google; Además, si el usuario selecciona una iglesia, podrá obtener toda la información de ella.

El requisito RF5 exige que la aplicación web provea una sección con libros de estudio recomendados por la iglesia y los pastores. Estos libros pueden incluir comentarios o resúmenes. La información para mostrar incluye el título del libro, la serie a la que pertenece, un resumen del libro y el URL de la imagen de portada, esta se encuentra almacenada en el *bucket* S3 recursostfm dentro de una carpeta llamada "libros".

Esta problemática se abordó con la "*BookFunction*". Esta función se encarga de leer la tabla "*book*", donde se almacena cada uno de los libros recomendados y convierte la información en un archivo JSON. El contenido de archivo JSON se entrega al *frontend* de la aplicación, el cual recibe la información y solicita las imágenes de las portadas de los libros al servicio de CloudFront según lo vaya requiriendo el usuario lo que brinda una mejor experiencia de usuario y se obtiene una mejor puntuación en el SEO.

El último requisito, RF6, exige que la aplicación web provea una sección acceso a las ultimas predicaciones grabadas y escritas. Debe proveer la fecha en la que hizo, el título de la predica, el autor, la serie y el URL del video. Es necesario aclarar que para este proyecto el listado de las ultimas 10 predicaciones se realiza en el *frontend* a través del api que expone YouTube. Por otro lado, toda la demás información debe proveerse a través del *backend*.

Esta problemática se abordó con la "*PreachFunction*". Esta función se encarga de leer la tabla "*preach*", donde se almacena las predicaciones dadas a la congregación, y convierte la información en un archivo JSON. Al igual que las anteriores funciones, este contenido se entrega al *frontend* de la aplicación. Una vez que el *frontend* recibe la información, solicita el archivo con el contenido en texto de las palabras mencionadas durante la predicación al *bucket* S3 llamado recursostfm a través del servicio CloudFront, en la carpeta llamada "*predicaciones*". Este proceso se realiza según lo vaya requiriendo el usuario.

La integración de las funciones Lambda con Java en el proyecto se realizó de manera ágil, permitiendo una fácil gestión y despliegue de los recursos desde la terminal. Esta metodología de desarrollo facilitó la implementación de las funcionalidades del *backend* y garantizó un correcto funcionamiento de la arquitectura sin servidor.

En resumen, el capítulo de desarrollo y despliegue de las funciones Lambda con Java, se centró en la configuración detallada de los recursos necesarios para el *backend* de la arquitectura web *serverless*, utilizando el AWS SAM para simplificar el proceso de implementación y garantizar la gestión de los servicios en la nube de AWS.

## 4.7 Despliegue de API Gateway

En la arquitectura descrita en el capítulo 3, el servicio API Gateway desempeña un papel crucial al actuar como el punto de entrada para las solicitudes externas, permitiendo una comunicación efectiva entre el *frontend* y el b*ackend*. Para esto es necesario integrar las funciones *lambda* con el servicio API Gateway, lo que provee una solución a los requerimientos no funcionales NRF2 de escalabilidad, NRF3 de disponibilidad y NRF4 facilidad de mantenimiento.

En esta sección se describe la definición de la API que ofrece el *backend* de la arquitectura, así como su despliegue en el servicio API Gateway. Se indica como son las peticiones a los recursos expuestos por la API del *backend*, solicitados desde el lado del cliente. Este *backend* brinda respuesta a tres servicios en específico (vea Ilustración 28. Servicios ofrecidos por la API del Backend.).

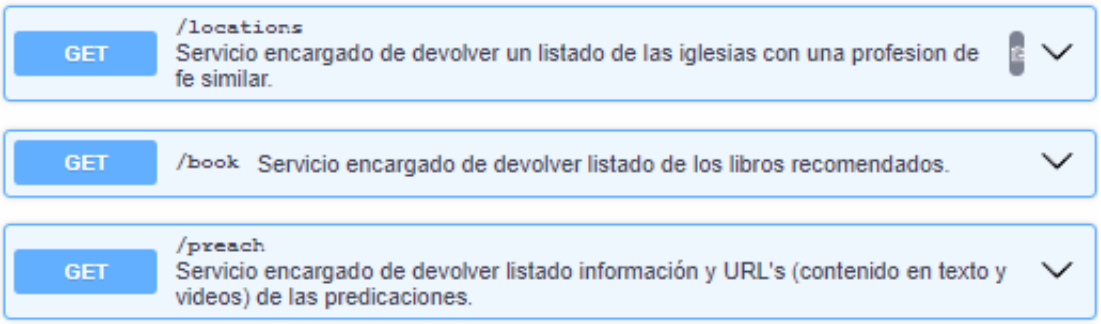


*Ilustración 28. Servicios ofrecidos por la API del Backend.*

El primer servicio consiste en entregar al *frontend* de la aplicación un listado de iglesias con una profesión de fe similar, denominado "*locations*". Cada elemento del listado incluirá la información de su iglesia respectiva. El esquema completo de respuesta se encuentra en la Ilustración 29. Esto se hace mediante el método "GET".

Es conveniente mencionar que no es necesario especificar ningún parámetro al solicitar la petición. Por último, hay que mencionar que la lista de iglesias se encuentra en el atributo *body* de la respuesta.

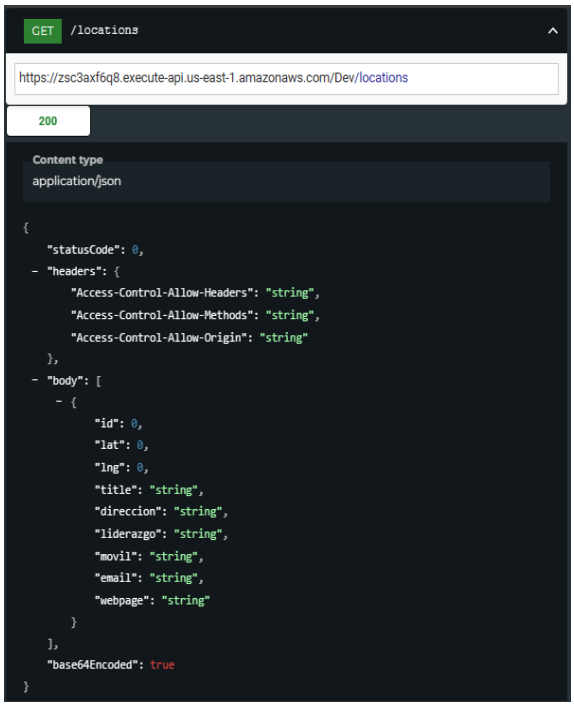


*Ilustración 29. Endpoint para el servicio de información de iglesias por método GET.*

El segundo servicio se encarga de devolver un listado de los libros recomendados para los usuarios de la aplicación. Este servicio se denomina “book” y se accede a él mediante el método “GET”.

No es necesario especificar ningún parámetro al realizar la petición. La respuesta del servicio se encuentra en el atributo “body” y contiene un listado de objetos JSON con la información de cada libro (vea Ilustración 30).

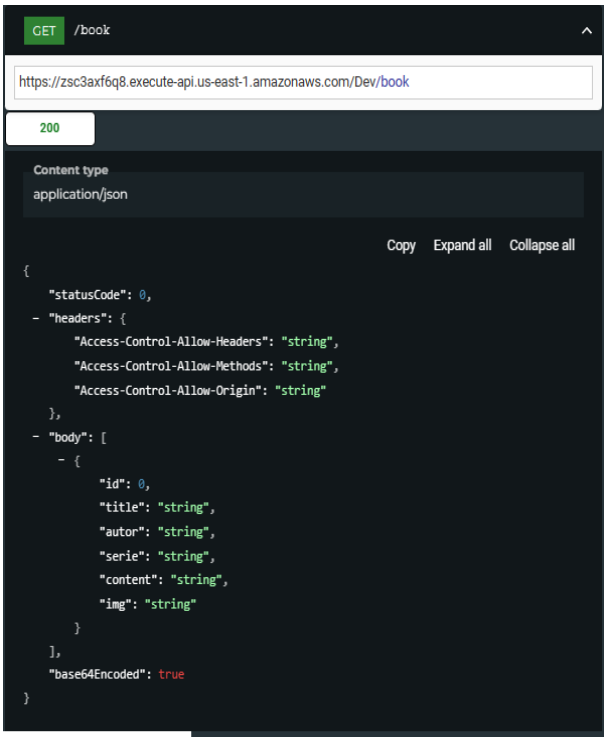


*Ilustración 30.. Endpoint para el servicio de libros recomendados por el método GET.*

El tercer servicio se encarga de devolver un listado de las predicaciones dadas a la congregación. Este servicio se denomina “predications” y se accede a él mediante el método “GET”.

No es necesario especificar ningún parámetro al realizar la petición. La respuesta del servicio se encuentra en el atributo “body” y contiene un listado de objetos JSON con la información de cada predicación (vea Ilustración 31).

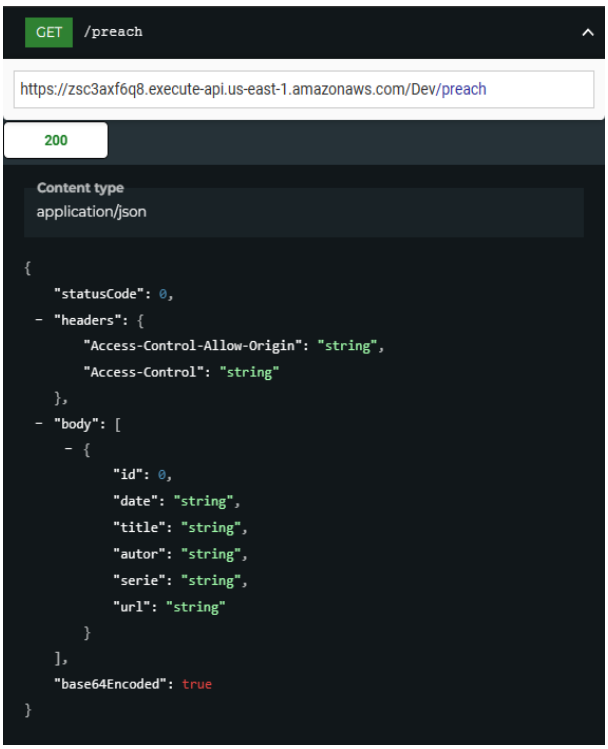


*Ilustración 31. Endpoint para el servicio de predicaciones por el método GET.*

En cuanto al proceso por el cual se realizó el despliegue del servicio API Gateway, que está encargada de recibir todas las peticiones del *frontend* hacia el *backend* e invocar la función que sea requerida según el servicio solicitado en el path (“/locations”, “/book”, “/preach”) del URL. Esto se hizo empleando SAM, que permitió la configuración e integración con el servicio AWS Lambda de manera que pueda configurar de forma ágil el servicio de API Gateway para recibir solicitudes HTTP y desencadenar eventos en las funciones Lambda correspondientes en el *backend*.

Para la configuración dentro del ecosistema de AWS para el API Gateway fue necesario la elección entre dos métodos de comunicación a través de la API:

- REST: Este método de comunicación utiliza el protocolo HTTP y sigue una arquitectura de solicitud-respuesta. Cada solicitud del cliente a la API se realiza de forma independiente y no se requiere mantener una conexión abierta entre el API Gateway y el cliente. Esto lo hace adecuado para escenarios en los que no se necesita una conexión persistente entre el cliente y el servicio de la API.
- WebSocket: Este método de comunicación permite una conexión bidireccional persistente entre el cliente y el API Gateway. Es útil en casos en los que se necesita una comunicación en tiempo real y la capacidad de enviar y recibir datos de forma continua. Sin embargo, en este caso particular, dado que no se requiere mantener una conexión abierta entre el API Gateway y el cliente que solicita información a través de la API REST, se selecciona el método REST.

Debido a la arquitectura diseñada para este trabajo, donde no se requiere una conexión persistente entre el cliente y el servicio de la API, se opta por la opción de utilizar la API REST.

Para facilitar la gestión y el despliegue de la API, se utiliza la herramienta API Gateway. Esta herramienta permite configurar la API en diferentes fases Ilustración 32, como desarrollo, puesta en escena y producción, que se encuentran aisladas unas de las otras. Esto facilita la realización de pruebas al modificar la fase en la que se está trabajando. Para este trabajo se ha definido en la fase de desarrollo el despliegue de la API en el servicio de API Gateway.

```
Resources:
  ServiciosApiGateway:
    Type: AWS::Serverless::Api
    Properties:
      StageName: dev
      Name: servicios-api-gateway
```

*Ilustración 32. Configuración servicio API Gateway.*

La implementación se hace tal y como se muestra en la Ilustración 32. La definición del servicio es breve, pero permite la integración ágil de las funciones *lambda*. Esta integración se realiza mediante el protocolo HTTP o HTTPS (GET, POST, PUT, PATCH, DELETE, OPTIONS, entre otros) tal como se mostró en las Ilustración 25 Ilustración 33. En estas ilustraciones se observa la integración del servicio API Gateway, la ruta(*path*) con la que se accede al recurso deseado generando el evento para lanzar la función deseada y por último la integración con la base de datos.


*Ilustración 33. integración Endpoint Backend con Api Gateway.*

# 5 Validación

En este capítulo se procede a validar la aplicación con el fin de verificar el correcto funcionamiento de la arquitectura, así como medir el rendimiento de la entrega de contenido y el tiempo de respuesta del servicio *serverless*.

## 5.1 Validación de entrega de contenido Cliente (SPA).

La primera validación consiste en verificar la entrega correcta del contenido proporcionada por la arquitectura web serverless y su diferencia en rendimiento en la entrega del contenido entre los servicios Amplify y CloudFront. Esta verificación se realiza en el navegador del usuario final, utilizando las herramientas de desarrollador del navegador Edge de Microsoft.

Primero se verificará el tiempo de respuesta al realizar la petición en el servicio de AWS Amplify. Este valor corresponde a 6,68 segundos y se encuentra en el *DOMContentLoaded,* como se observa en la parte inferior de la Ilustración 34 (en color marrón), indicando el tiempo de carga de todos los recursos necesarios para la construcción de la página (tardando un total de 10.98s para construirla en su totalidad).

Téngase en cuenta que se ha deshabilitado la memoria caché del navegador para no afectar las mediciones entre una medición y otra debido a la caché local del navegador, que por defecto si se consulta la misma página, primero realiza la búsqueda en su caché local y si no encuentra el contenido recurre a generar la petición al URL especificado.

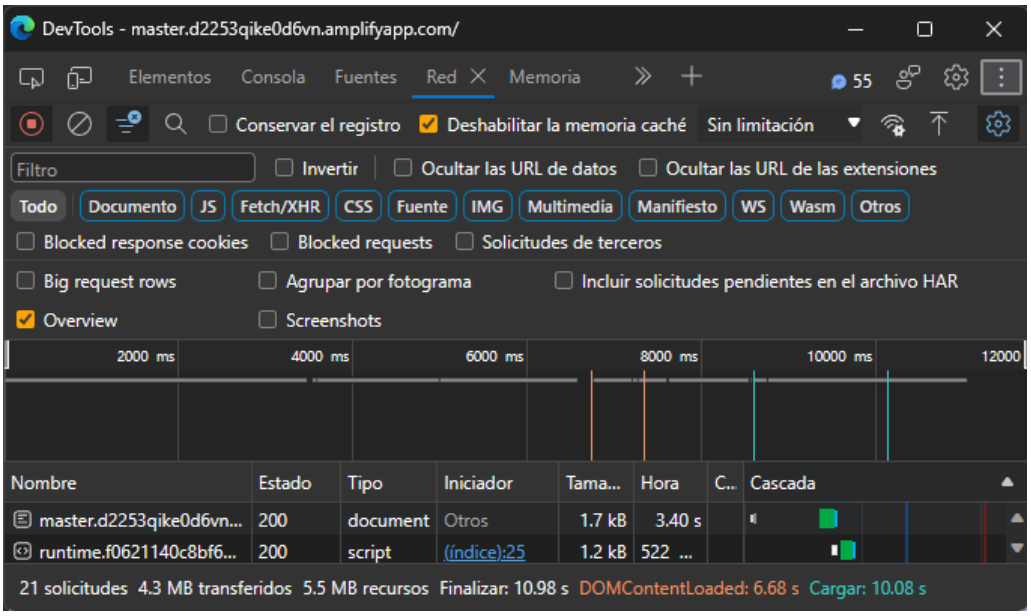


*Ilustración 34. Tiempo de respuesta servicio AWS Amplify.*

Por otra parte, al realizar la petición al servicio de Amazon CloudFront, en esta medición al igual que en la anterior se ha deshabilitado la caché local para evitar posibles alteraciones en los resultados debido a la configuración del navegador. Los tiempos de

respuesta que se observan en la Ilustración 35, identificando una diferencia notable al tardar apenas 1.81 s en recibir el *DOMContentLoaded*.

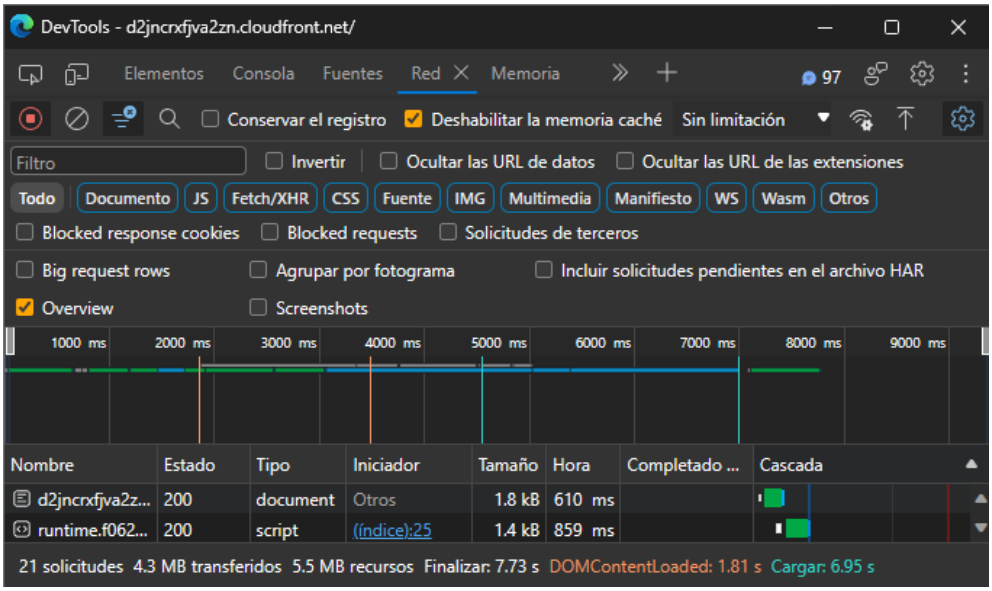


*Ilustración 35. Tiempo de respuesta servicio Amazon CloudFront.*

La mejora en la entrega del contenido por parte de CloudFront es debido a que cuenta con una red global con más de 550 puntos de persistencia, de los cuales dentro de España hay 2 en Barcelona y 10 en Madrid. Al disminuir la distancia recorrida entre el cliente y el servidor que entrega el contenido (en este caso uno de los puntos de persistencia presentes en España), reduce el tiempo de latencia considerablemente y mejora la experiencia de usuario reduciendo el tiempo de espera de carga de contenido.

## 5.2 Validación de entrega de datos API Serverless.

Para evaluar el rendimiento del servicio de API Gateway, se utilizan dos enfoques diferentes. El primero consiste en observar y analizar la simulación de un cliente final, mientras que el segundo enfoque implica analizar el comportamiento de los servicios de AWS, como la función Lambda y DynamoDB.

### 5.2.1 Validación en el cliente

La segunda validación consiste en hacer una prueba de estrés de carga al servicio de API Gateway en el lado del cliente/usuario a través la herramienta JMeter. Esta herramienta se configura para simular 40 usuarios. Se va a realizar la prueba con dos duraciones diferentes. La primera será con una duración de 60s y la segunda prueba será de 120s. Adicionalmente, se ha configurado un retardo por cada hilo de 5 milisegundos para evitar el aglutinamiento de ejecución de hilos en el mismo instante.

La activación de los hilos durante el transcurso de las pruebas se lleva a cabo de manera progresiva, como se puede observar en la Ilustración 36 y Ilustración 37. En ambos casos, se produce un incremento en la simulación de usuarios que realizan solicitudes a la API, aumentando gradualmente el numero hasta alcanzar un total de 40 usuarios. Cada hilo utilizado en la simulación de usuarios recibe el resultado de la API y vuelve a lanzar la solicitud de forma continuamente, repitiendo este ciclo hasta que se cumpla el intervalo de tiempo configurado.

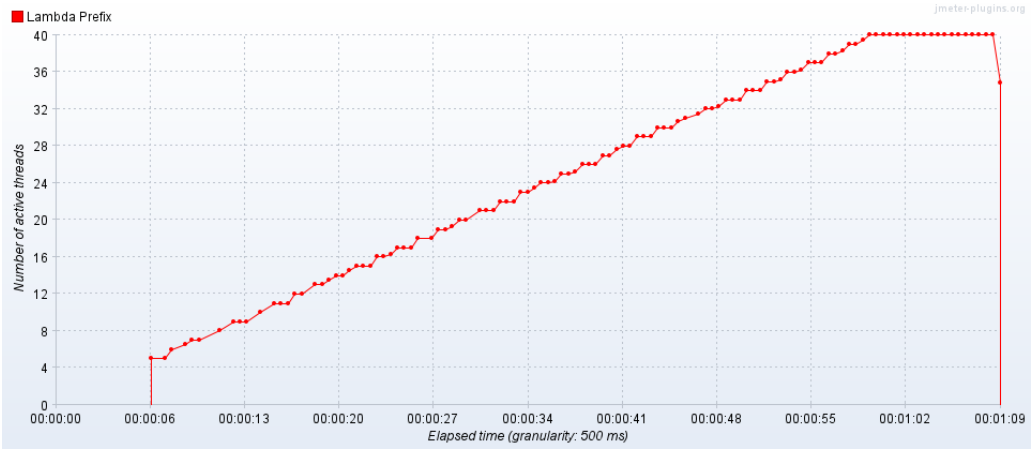


*Ilustración 36. Activación de los hilos a través del tiempo 60s.*

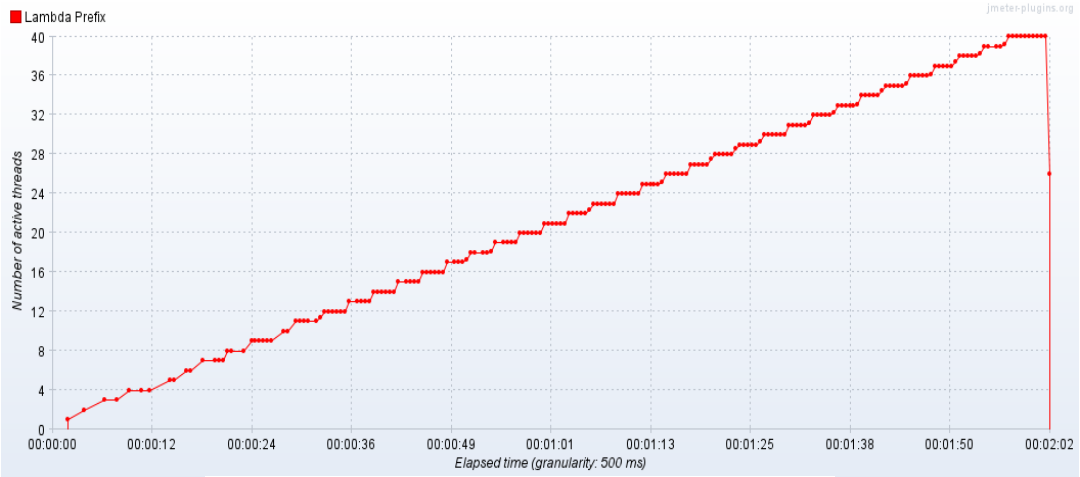


*Ilustración 37. Activación de los hilos a través del tiempo 120s.*

El comportamiento observado en la Ilustración 38, muestra algunos picos durante la ejecución, mientras que los demás tiempos de ejecución y respuesta son inferiores a los 4 segundos. Estos picos altos pueden describirse como el efecto de un arranque en frio de una función Lambda, repitiéndose aproximadamente 4 veces más. Esta observación es consistente con el hecho de que las funciones Lambda se ejecutan, pero queda un lapso disponible para posibles llamadas adicionales a la misma función lo que reduce el tiempo de las posteriores llamadas al mismo contenedor de la función disponible. Esto proporciona el beneficio de no incurrir en costes adicionales, mejorando el tiempo de respuesta al evitar el tiempo de creación y carga de la imagen del contenedor, así como las configuraciones necesarias para el funcionamiento de la función Lambda por parte de la infraestructura de AWS.

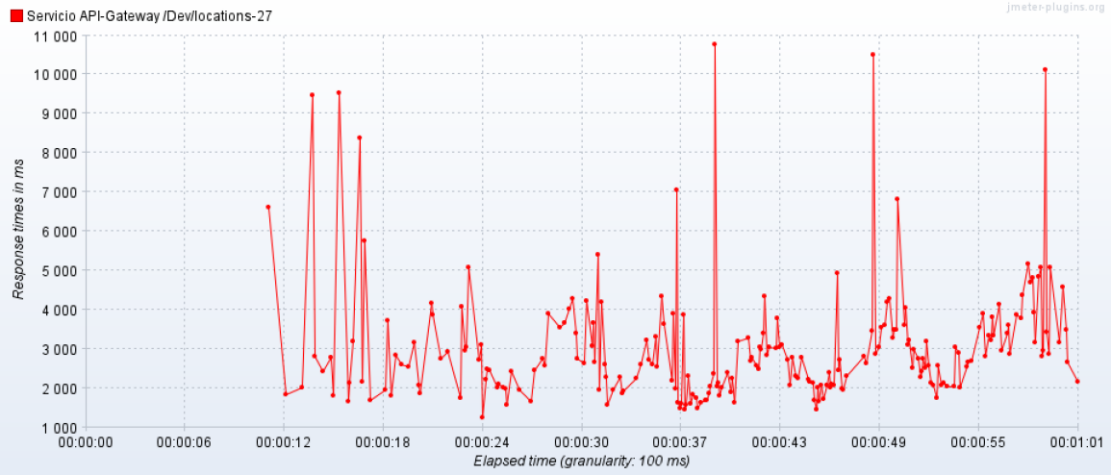


*Ilustración 38. Tiempos de respuesta a lo largo del tiempo 40 Hilos - 60s.*

La tendencia observada en la Ilustración 38, muestra una media de tiempo de respuesta de las solicitudes por parte del servicio API Gateway entre 1.5 s y 4 s. Es importante destacar que estos resultados no están directamente relacionados con el tiempo de ejecución de la función Lambda, el cual se mostrará en la siguiente sección. En cambio, estos resultados indican el tiempo que tarda en llegar la respuesta desde que se lanzó la petición. Por otra parte, en la Ilustración 39, se muestra los resultados de la segunda prueba extendiendo el tiempo hasta los 120 s, donde al igual que en la anterior los resultados se encuentra entre 1.5 s y 4 s en los primeros 60 s. Después el intervalo en el que se encontró los resultados de respuesta se encuentra entre 1.5 s y 2.5 s durante 50 s.

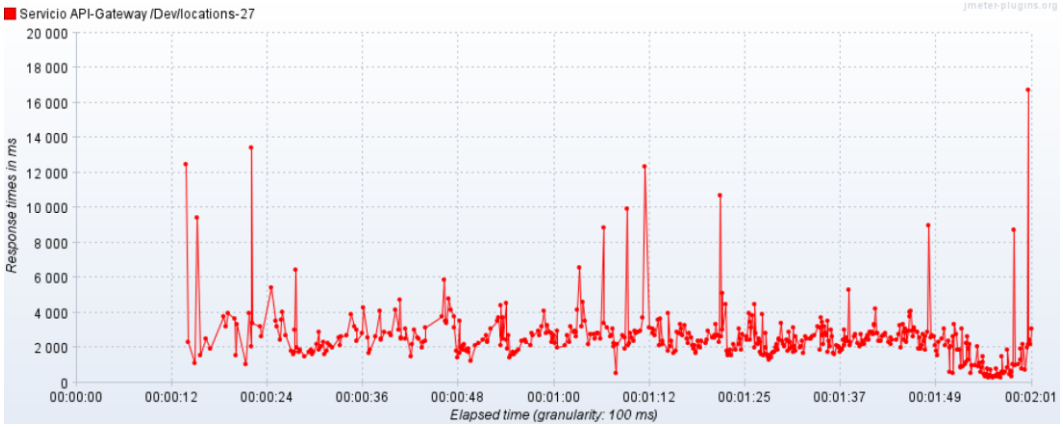


*Ilustración 39. Tiempos de respuesta a lo largo del tiempo 40 Hilos - 120s.*

La Tabla 11 muestra un resumen ampliado de los resultados obtenidos en las dos ilustraciones anteriores. Un dato importante que destacar es que la media en la primera prueba, con 2,892 s, es superior a la obtenida en la segunda prueba, con 2,225 s. Esto sugiere que al aumentar el intervalo en que se mantienen realizando solicitudes al servicio API Gateway en una distribución mayor de tiempo se obtienen mejores resultados.

| Servicio API Gateway | #Muestras | Media | Min | Max | %Error | Rendimiento | Kb /sec | Sent KB/sec |
|---|---|---|---|---|---|---|---|---|
| /Dev/locations (60s) | 447 | 2892 | 496 | 13883 | 8,95% | 7,25 | 660,89 | 3,31 |
| /Dev/locations (120s) | 1128 | 2225 | 61 | 17780 | 7,25% | 9,29 | 654,41 | 4,48 |

*Tabla 11. Resumen métricas pruebas estrés JMeter.*

Además de lo anterior, la tasa de recepción de información por segundo aumentó un 44.5% en la segunda prueba de estrés, considerando que las muestras fueron 2.5 veces mayor en el segundo intervalo de tiempo con 1558 vs 608. Estas características presentan ventajas para el usuario final, reduciendo el tiempo de espera.

**Validación del Servicio Serverless.**

Para esta sección del capítulo se evaluará el comportamiento de los servicios encargados de responder a las solicitudes hechas en la prueba de estrés de carga simulando 40 usuarios realizando solicitudes en las ventanas de tiempo de 60 s y 120 s respectivamente a la API expuesta. Con el fin de responder a las solicitudes de cada cliente, los servicios que intervienen son inicialmente API Gateway, disparando el evento que invoca la ejecución de la función Lambda. Es la función Lambda la que se encarga de consultar la base de datos DynamoDB, y se mantiene activa hasta que recibe la respuesta de la base de datos y responde a la solicitud a través del servicio de API Gateway.

Primero analizaremos el comportamiento y consumo de unidades de capacidad de lectura medio por minuto de la tabla almacenada en el servicio de DynamoDB para verificar que no sobrepasa el aprovisionamiento que se ha asignado, y supere la capa gratuita proporcionada por AWS para el servicio de DynamoDB de por vida según sus términos y condiciones.

Considerando que el tamaño aproximado de cada elemento almacenado en la tabla *location*, alojada en DynamoDB es de 229.28 Bytes (B) y la tabla ocupa 70.6 KB (Kilo Bytes) en su totalidad con 308 elementos almacenados. Cabe destacar que esta es la tabla con mayor cantidad de elementos de las tres tablas. También es importante recordar que el contenido de estas tablas no se almacena directamente dentro de las funciones *lambda*, debido a que esto permite añadir nuevos elementos, modificarlos o eliminarlos sin tener que modificar la estructura.

La lectura completa de la tabla consume 2 RCU configuradas en modo eventualmente consistente, lo que permite realizar un mayor número de peticiones por segundo. Sin embargo, es importante tener en cuenta la restricción de AWS en cuanto al número máximo de RCU-Hrs (ReadCapacityUnit-Hours) por sus siglas en inglés, que está limitado a 18600.0 gratuitas por mes.

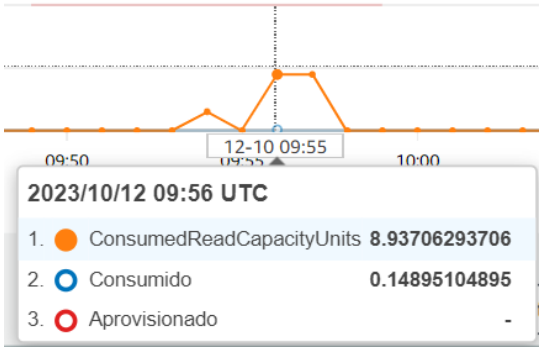


*Ilustración 41. Métricas RCU consumidas en DynamoDB 60s.*


*Ilustración 40. Métricas RCU consumidas en DynamoDB 120s.*

Como se puede observar en la Ilustración 41 y Ilustración 40, al realizar las dos pruebas de estrés, el consumo de unidades aprovisionadas consumido es de 9 RCU . Estas lecturas son del tipo eventualmente consistente, lo que implica que está atendiendo una tasa de 4 peticiones por segundo de forma simultánea, ya que se requiere de 2 RCU por cada lectura de la tabla con una arquitectura eventualmente consistente. Se estima una capacidad de atención de solicitudes de lectura, de la tabla completa de hasta 12 por segundo, sin sobrepasar el límite de la capa gratuita de DynamoDB en AWS, que es de 25 RCU por segundo. Esto representa una ventaja, considerando que se podrían admitir 720 peticiones por minuto al servicio AWS sin generar un coste adicional ni requerir de administración, gestión o configuración adicional de la infraestructura, aparte del aprovisionamiento de las RCU de la tabla en cuestión.

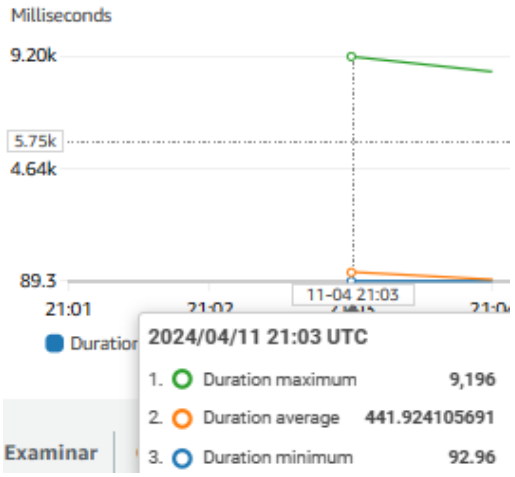


*Ilustración 42. Tiempo de ejecución Función 60s inicio.*

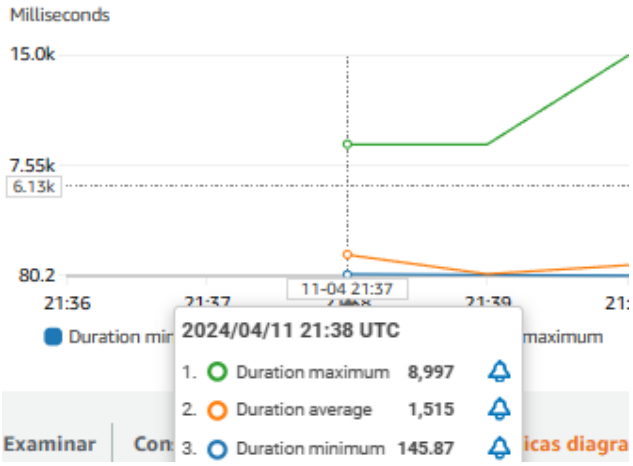


*Ilustración 43. Tiempo de ejecución Función 120s inicio.*

Los resultados obtenidos al someter a la prueba de estrés a la función Lambda, con ventanas de tiempo entre 60s y 120s, (Ilustración 42 y Ilustración 43), revelan una duración máxima de aproximadamente 9 segundos. Esto se debe al tiempo que la crear el contenedor, solicitar la información al servicio de DynamoDB, esperar su respuesta y terminar de ejecutar el código dentro de la función. Este tiempo se reduce en las invocaciones posteriores como se explica más adelante.

La documentación de AWS no ofrece el dato exacto del tiempo que se mantiene el beneficio de las funciones preaprovisionadas y aunque no es posible identificarlo con total precisión, se han obtenido resultados experimentales relacionados con la precarga de la función de hasta 120 segundos. Esta función, al estar desplegada y lista para ejecutarse, evita la necesidad de crear el contenedor desde cero, lo que ahorra tiempo en segundos pasando de 6s o 9s a tan solo (200 a 300) milisegundos.

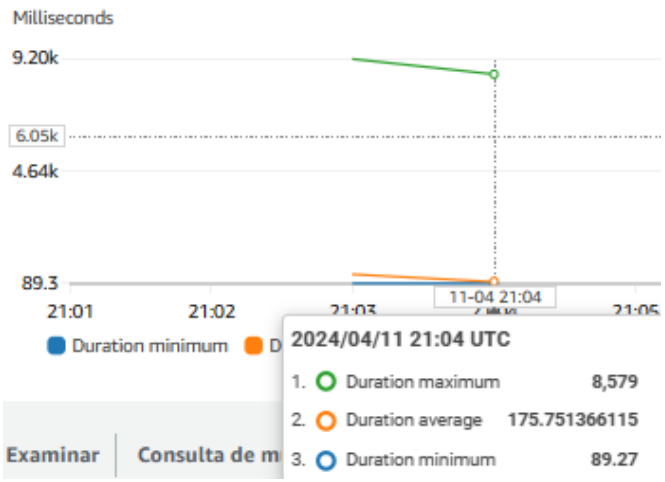


*Ilustración 44. Tiempo de ejecución Función 60s final.*

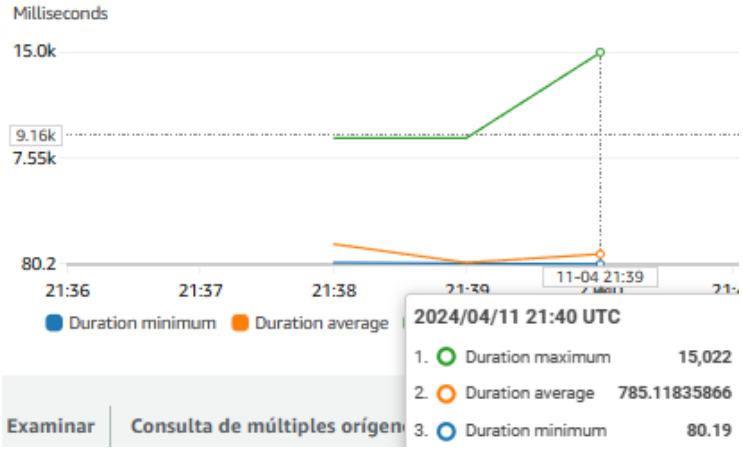


*Ilustración 45. Tiempo de ejecución Función 120s final.*

El comportamiento que se observa en las Ilustración 44 y 45 de los resultados de las pruebas de carga, muestra que conforme se utiliza más seguido la función Lambda disminuye el tiempo medio de ejecución de la función Lambda arrojando valores medios de (89.3 y 80.2) milisegundos lo que indica una mejora en rendimiento del 89,42 % en el tiempo de las posteriores ejecuciones de las funciones *lambda* una vez creado el contenedor en el que se ejecutará el código que responde al evento generado por el API Gateway.

En conclusión, el capítulo de validaciones ha sido fundamental para garantizar la calidad de la arquitectura web serverless diseñada y desplegada para la institución religiosa.

A lo largo del capítulo, se han presentado las diferentes técnicas y herramientas utilizadas para validar la solución tecnológica, incluyendo pruebas unitarias, pruebas de integración, pruebas de carga y pruebas de seguridad. Además, se ha destacado la importancia de la automatización de las pruebas para garantizar la eficiencia y la consistencia en el proceso de validación. En general, el capítulo de validaciones ha permitido asegurar que la arquitectura web serverless cumpla con los requisitos de calidad y seguridad necesarios para satisfacer las necesidades tecnológicas de la Iglesia Bautista Reformada.

# 6 Costes mantenimiento

En este capítulo, se describirán los costes asociados de mantenimiento actuales del proyecto, considerando la capa gratuita ofrecida por el proveedor de servicios Amazon. Además, se indica que servicios se verán afectados una vez se supere la capa gratuita de 12. Después de explicar el mantenimiento de cada uno de los servicios por separado, se presenta un consolidado de los costes de mantenimiento de la aplicación web serverless.

Comenzando con Amazon CloudFront, donde se muestra el valor de la capa gratuita que provee Amazon. Es importante destacar que esta capa gratuita es para siempre, pero una vez superado esas características, se incurrirá en los costes de la capa de pago por uso (ver Tabla 12). Actualmente, los costes del proyecto se mantienen dentro de los parámetros de la capa gratuita.

| Por mes | Transferencia saliente GB | Precio por Métodos HTTP (10 000) | Precios Metodos HTTPS |
|---|---|---|---|
| Capa Gratuita | 0 primer TB | 0 primeros 10 millones (cada mes) | 0 primeros 10 millones (cada mes) |
| De pago | 0,085 USD | 0,0090 USD | 0,0120 USD |

*Tabla 12. Costes mantenimiento Amazon CloudFront.*

El siguiente servicio es Amazon S3, que provee una capa gratuita durante 12 meses. Durante los primeros 12 meses, no ha habido ningún costo por la implementación de este trabajo (ver Tabla 13). Sin embargo, una vez superado ese plazo, se debe tomar en cuenta la característica que se ve afectada. En este caso, se trata del pago por almacenamiento en GB. Actualmente, el tamaño de los objetos no supera 1 GB, pero se estima que este valor aumentará a medida que se vayan agregando más datos a cada uno de los servicios ofrecidos.

| Por mes | Almacenamiento GB | Transferencia de datos GB |
|---|---|---|
| Capa Gratuita | 0 (12 meses) USD | primero 100 GB (siempre) |
| De pago | 0,023 USD | 0,09 USD |

*Tabla 13. Costes por capas Amazon S3.*

La síntesis por el servicio Amazon S3 se estaría pagando mensualmente lo que se indica en la Tabla 14

| Total Mantenimiento | Valor |
|---|---|
| Actualmente | 0 USD |
| Mes (en adelante) | 0,046 USD |

*Tabla 14. Resumen mantenimiento por mes Amazon S3.*

Los costes asociados al servicio encargado de almacenar y disponer la aplicación web del *frontend* se detallan en la Tabla 15. Se puede observar los costes de las dos capas: la capa gratuita cubre 12 meses, y después de este periodo, es necesario considerar los costes de la capa de pago.

| Por mes | Creación e implementación (por minuto) | Almacenamiento GB | Transferencia saliente de datos GB |
|---|---|---|---|
| Capa Gratuita (12 meses) | 0 USD, Gratis 1000 minutos de creación | Sin costo hasta los 5 GB | Sin costo hasta los 15 GB |
| De pago | 0,01 USD | 0,023 USD | 0,15 USD |

*Tabla 15. Costes por capas AWS Amplify.*

Resumiendo, los costes para este trabajo actualmente y una vez vencida el plazo de la capa gratuita (ver Tabla 16).

| Total Mantenimiento | Valor |
|---|---|
| Mes (actualmente) | 0 USD |
| Después de capa gratuita | 0,146 USD |

*Tabla 16. Resumen mantenimiento por mes AWS Amplify.*

Los costes de mantenimiento asociados al servicio encargado de exponer el *backend* de la aplicación (Amazon API Gateway) también se divide en dos capas. Una de ella es gratuita durante los primero 12 meses, como se muestra en la Tabla 17.

| Capa | Numero de solicitudes (Por mes) | Precio (por millón) |
|---|---|---|
| Gratuita (12 meses) | 333 millones | 0 USD |
| De pago | 333 millones | 3,50 USD |

*Tabla 17. Costes por capas Amazon API Gateway.*

Lo que quiere decir que los costes de mantenimiento para este trabajo son los que se observan en la Tabla 18.

| Total Mantenimiento | Valor |
|---|---|
| Mes (actualmente) | 0 USD |
| En adelante | 3,50 USD |

*Tabla 18. Resumen mantenimiento por mes API Gateway.*

Los costes de mantenimiento del servicio encargado de tener un servicio *Backend serverless* (AWS Lambda).

| Capa | Arquitectura | Duración (GB/segundo) | Solicitudes | Memoria (MB/ms) |
|---|---|---|---|---|
| Gratuita | ARM y x86 | 3,2 millones de segundos | 1 000 000 (por mes) | 128 - 528 |
| De Pago | ARM | 0,0000133334 USD | 0,20 USD por un millón de solicitudes | (17, 67) *10^-9 USD |

*Tabla 19. Costes por capas AWS Lambda.*

El resumen de los costes de mantenimiento para el servicio de AWS Lambda se encuentra en la Tabla 20.

| Total Mantenimiento | Valor |
|---|---|
| Mes | 0 USD |

*Tabla 20. Resumen mantenimiento por mes AWS Lambda.*

El ultimo servicio por mencionar sus costes de mantenimiento es Amazon DynamoDB, Comenzando por la capa gratuita ofrecida por Amazon, que se mantiene de forma permanente (ver Tabla 21).

| Caracteristica | Almacenamiento | Solicitudes R/W | Transferencia saliente de datos |
|---|---|---|---|
| Aprovisionado gratuitamente | 25 GB | 200 millones | 1 GB, fuera de la región |

*Tabla 21. Capa Gratuita Amazon DynamoDB.*

Una vez superadas las características de la capa gratuita en necesario pagar los costes que se indican en la Tabla 22 que corresponden a la capa de pago por uso.

| Almacenamiento (GB/Mes) | Transferencia saliente de datos |
|---|---|
| 0,25 USD | 1 GB, acumulado para los servicios de AWS en distintas AZ's |

*Tabla 22. Capa de pago Amazon DynamoDB.*

En este proyecto, el coste de mantenimiento del servicio DynamoDB ha sido el que se muestra en la Tabla 23, debido a que no se superó la capa gratuita, incluso durante las pruebas de estrés realizadas a los servicios y, por ende, a la base de datos.

| Total Mantenimiento | Valor |
|---|---|
| Mes | 0 USD |

*Tabla 23.Resumen mantenimiento por mes DynamoDB.*

Como resultado, el mantenimiento final de todos los servicios utilizados en este trabajo se muestra en la Tabla 24.

| Servicio | Valor (USD) |
|---|---|
| CloudFront | 0 |
| S3 | 0,046 |
| Amplify | 0,146 |
| API Gateway | 3,5 |
| Funciones Lambda | 0 |
| DynamoDB | 0 |
| Total | 3,692 |

*Tabla 24. Resumen costes mantenimiento proyecto.*

Este valor se estima una vez que se ha superado el periodo de 12 meses de servicios gratuitos en aquellos servicios que solo ofrecían una capa gratuita durante ese periodo de tiempo. Es importante mencionar que, durante el desarrollo y entrega de este trabajo, el coste de mantenimiento fueron de 0 USD, debido a que no se superó la capa gratuita en ningún servicio, incluso durante las múltiples las múltiples pruebas de estrés.

# 7 Conclusiones y Trabajos Futuros.

En este trabajo final de máster se ha abordado el diseño y despliegue de una arquitectura web *serverless* para la Iglesia Bautista Reformada, utilizando los servicios de Amazon Web Services (AWS) en el contexto de la computación en la nube. En este proyecto se ha logrado el objetivo principal de desarrollar una arquitectura ágil, escalable y eficiente en costes que permita a la institución religiosa ofrecer su contenido de estudio en línea, comunicarse con su comunidad, dar sugerencias de Iglesias con la misma profesión de fe y darse a conocer eficazmente.

Durante el proceso de diseño de la arquitectura, se ha realizado un análisis exhaustivo de los servicios de AWS disponibles, seleccionando aquellos que mejor se adaptan a las necesidades específicas de la Iglesia Bautista Reformada de Valencia. La división en *Frontend* y *Backend* ha permitido una clara separación de responsabilidades, facilitando la gestión de las funcionalidades y mejorando la escalabilidad del sistema.

La implementación de servicios como AWS Lambda, Amazon Amplify, Amazon CloudFront y API Gateway ha sido fundamental para garantizar la ejecución eficiente de código, el almacenamiento seguro de contenido estático y la gestión de las peticiones de los usuarios. La evaluación de la arquitectura mediante pruebas de carga ha demostrado la robustez y fiabilidad del sistema, preparándolo para enfrentar un alto volumen de tráfico de manera efectiva.

En conclusión, este trabajo ha representado un paso significativo en la modernización de la infraestructura tecnológica de la Iglesia Bautista Reformada, brindando una plataforma digital innovadora que potencia su presencia en línea y fortalece su interacción con la comunidad. Las lecciones aprendidas durante este proceso sientan las bases para futuras mejoras y optimizaciones en la arquitectura web serverless implementada, asegurando su continuidad y evolución en un entorno tecnológico en constante cambio.

**Trabajos Futuros**

Como posibles trabajos futuros se considera soportar búsquedas indexadas para investigación por versión de la Biblia que permita encontrar todos los pasajes en las que se encuentra una palabra en específico o sus conjugaciones. Implementación de un sistema de autenticación ya sea con Amazon Cognito o cualquier otro proveedor de servicio de autenticación, un CMS (*Content Management System*) para actualización de elementos de la aplicación web

# Bibliografía


[1] Javier Rodríguez Domínguez, "'Serverless Computing', Funciones como Servicio (FaaS) para el soporte de cargas computacionales en la nube," Universidad Politécnica de Valencia, Valencia, 2020. Accessed: Jul. 14, 2023. [Online]. Available: https://riunet.upv.es/bitstream/handle/10251/159165/Rodriguez%20-%20Serverless%20Computing%2c%20Funciones%20como%20Servicio%20%28FaaS%29%20para%20el%20soporte%20de%20cargas%20comp....pdf

[2] E. Jonas *et al.*, "Cloud Programming Simplified: A Berkeley View on Serverless Computing," 2019, Accessed: Jul. 13, 2023. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html

[3] "Informática sin servidor: Amazon Web Services." Accessed: Mar. 27, 2024. [Online]. Available: https://aws.amazon.com/es/serverless/

[4] "Sin servidores | Google Cloud." Accessed: Mar. 27, 2024. [Online]. Available: https://cloud.google.com/serverless

[5] "¿Qué es la informática sin servidor? | IBM." Accessed: Mar. 27, 2024. [Online]. Available: https://www.ibm.com/es-es/topics/serverless

[6] "Azure sin servidor | Microsoft Azure." Accessed: Mar. 27, 2024. [Online]. Available: https://azure.microsoft.com/es-es/solutions/serverless

[7] "Informática sin servidor: Amazon Web Services." Accessed: Jul. 13, 2023. [Online]. Available: https://aws.amazon.com/es/serverless/

[8] H. Schildt, "Java: manual de referencia, 7ma Edición", Accessed: Sep. 06, 2023. [Online]. Available: www.HerbSchildt.com.

[9] "Oracle | Cloud Applications and Cloud Platform." Accessed: Oct. 17, 2023. [Online]. Available: https://www.oracle.com/

[10] O. Corporation, "Oracle Java SE Universal Subscription: Protect Your Investment in Java SE," 2023, Accessed: Oct. 17, 2023. [Online]. Available: https://www.oracle.com/support/premier/

[11] "Licencias - Proyecto GNU - Free Software Foundation." Accessed: Oct. 17, 2023. [Online]. Available: https://www.gnu.org/licenses/licenses.es.html

[12] "Presentación de la licencia gratuita de Java." Accessed: Oct. 17, 2023. [Online]. Available: https://blogs.oracle.com/oracle-latinoamerica/post/presentacion-de-la-licencia-gratuita-de-java

[13] "Eclipse Downloads | The Eclipse Foundation." Accessed: Oct. 17, 2023. [Online]. Available: https://www.eclipse.org/downloads/

[14] "IntelliJ IDEA: el IDE líder para Java y Kotlin." Accessed: Oct. 17, 2023. [Online]. Available: https://www.jetbrains.com/es-es/idea/

[15] "Visual Studio Code - Code Editing. Redefined." Accessed: Oct. 17, 2023. [Online]. Available: https://code.visualstudio.com/

[16] "JSON." Accessed: Oct. 17, 2023. [Online]. Available: https://www.json.org/json-es.html

[17] “TypeScript: JavaScript With Syntax For Types.” Accessed: Oct. 17, 2023. [Online]. Available: https://www.typescriptlang.org/

[18] “Angular.” Accessed: Oct. 20, 2023. [Online]. Available: https://angular.io/

[19] “React.” Accessed: Oct. 20, 2023. [Online]. Available: https://es.react.dev/

[20] “Node.js.” Accessed: Oct. 20, 2023. [Online]. Available: https://nodejs.org/es

[21] “Angular - Introduction to the Angular docs.” Accessed: Aug. 15, 2023. [Online]. Available: https://angular.io/docs

[22] “AWS | Cloud Computing - Servicios de informática en la nube.” Accessed: Oct. 20, 2023. [Online]. Available: https://aws.amazon.com/es/

[23] “¿Qué es AWS?” Accessed: Jul. 17, 2023. [Online]. Available: https://aws.amazon.com/es/what-is-aws/

[24] “AWS | Lambda - Gestión de recursos informáticos.” Accessed: Oct. 20, 2023. [Online]. Available: https://aws.amazon.com/es/lambda/

[25] “Machine Learning – Amazon Web Services.” Accessed: Oct. 20, 2023. [Online]. Available: https://aws.amazon.com/es/sagemaker/

[26] “Infraestructura global.” Accessed: Oct. 20, 2023. [Online]. Available: https://aws.amazon.com/es/about-aws/global-infrastructure/

[27] A. S. GUAYAS ESPIN, “ANÁLISIS COMPARATIVO DE LAS PLATAFORMAS AMAZON CLOUD, GOOGLE CLOUD, AZURE CLOUD,” UNIVERSIDAD TÉCNICA DE BABAHOYO, BABAHOYO LOS RIOS, 2023.

[28] “Amplify Documentation - AWS Amplify Documentation.” Accessed: Apr. 14, 2024. [Online]. Available: https://docs.amplify.aws/

[29] “Amazon API Gateway | API Management | Amazon Web Services.” Accessed: Oct. 21, 2023. [Online]. Available: https://aws.amazon.com/es/api-gateway/

[30] “Monitoreo de infraestructuras y aplicaciones – Amazon CloudWatch – Amazon Web Services.” Accessed: Oct. 21, 2023. [Online]. Available: https://aws.amazon.com/es/cloudwatch/

[31] “AWS | Servicio de base de datos gestionada NoSQL (DynamoDB).” Accessed: Oct. 21, 2023. [Online]. Available: https://aws.amazon.com/es/dynamodb/

[32] “AWS | Servicio de entrega de contenidos y transferencia de datos.” Accessed: Oct. 21, 2023. [Online]. Available: https://aws.amazon.com/es/cloudfront/

[33] “AWS | Almacenamiento de datos seguro en la nube (S3).” Accessed: Oct. 21, 2023. [Online]. Available: https://aws.amazon.com/es/s3/

[34] AWS, “Amazon S3.” Accessed: Nov. 03, 2023. [Online]. Available: https://aws.amazon.com/es/pm/serv-s3

[35] “AWS | Servicio de entrega de contenidos y transferencia de datos.” Accessed: Aug. 15, 2023. [Online]. Available: https://aws.amazon.com/es/cloudfront/

[36] “What is Amazon CloudFront? - Amazon CloudFront.” Accessed: Aug. 15, 2023. [Online]. Available: https://docs.aws.amazon.com/en_en/AmazonCloudFront/latest/DeveloperGuide/Introduction.html

[37] "Desarrollo de pila completa - Aplicaciones web y móviles - AWS Amplify." Accessed: Aug. 12, 2023. [Online]. Available: https://aws.amazon.com/es/amplify/

[38] "Bases de datos no relacionales | Bases de datos de gráficos | AWS." Accessed: Aug. 11, 2023. [Online]. Available: https://aws.amazon.com/es/nosql/

[39] "AWS | Servicio de base de datos gestionada NoSQL (DynamoDB)." Accessed: Aug. 11, 2023. [Online]. Available: https://aws.amazon.com/es/dynamodb/

[40] "Guía de Configuración segura para Monitorización y gestión AWS Guía de Seguridad de las TIC CCN-STIC 887G", Accessed: Jul. 18, 2023. [Online]. Available: https://www.ccn-cert.cni.es/es/pdf/guias/series-ccn-stic/800-guia-esquema-nacional-de-seguridad/6882-ccn-stic-887g-guia-de-configuracion-segura-para-monitorizacion-y-gestion-aws/file.html

[41] J. Ortiz Amaya, "Plataforma Serverless de Procesado de Datos Abiertos," Oct. 2020, Accessed: Jul. 18, 2023. [Online]. Available: https://riunet.upv.es:443/handle/10251/153189

[42] "Apache JMeter - Apache JMeter™." Accessed: Nov. 03, 2023. [Online]. Available: https://jmeter.apache.org/

[43] "Capa gratuita de AWS | Cloud computing gratis |AWS." Accessed: Apr. 15, 2024. [Online]. Available: https://aws.amazon.com/es/free

[44] "Chapter 15: Requirements and user stories." Accessed: Mar. 29, 2024. [Online]. Available: https://www.agilebusiness.org/dsdm-project-framework/requirements-and-user-stories.html

[45] "Características clave de una red de entrega de contenido – Rendimiento, seguridad – Amazon CloudFront." Accessed: Aug. 31, 2023. [Online]. Available: https://aws.amazon.com/es/cloudfront/features

[46] "CDN de Amazon CloudFront - Planes y precios - Probar de forma gratuita." Accessed: Aug. 31, 2023. [Online]. Available: https://aws.amazon.com/es/cloudfront/pricing

[47] S. López and J. Manuel, "UNIVERSIDAD POLITÉCNICA DE SINALOA PROGRAMA ACADÉMICO DE INGENIERÍA EN INFORMÁTICA 'PROCESADORES ARM EL FUTURO DEL DESARROLLO SOCIAL Y LABORAL'", Accessed: Apr. 08, 2024. [Online]. Available: http://repositorio.upsin.edu.mx/Fragmentos/tesinas/A002SALAZARLOPEZJESUSMANUEL6188.pdf