



Universidad Politécnica
de Madrid



**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

**Desarrollo de una Plataforma de Venta
de Entradas para Eventos en AWS**

Autor: Brais Lagoa Rúa

Tutora: Sonia de Frutos Cid

Madrid, junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado

Grado en Ingeniería Informática

Título: Desarrollo de una Plataforma de Venta de Entradas para Eventos en AWS

Junio 2024

Autor: Brais Lagoa Rúa

Tutor:

Sonia de Frutos Cid

Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software

ETSI Informáticos

Universidad Politécnica de Madrid

Resumen

Este Trabajo de Fin de Grado (TFG) busca diseñar e implementar una plataforma web para la venta de entradas para discotecas, festivales y otro tipo de eventos que actúe como Software as a Service (SaaS) [1], centrándose en conseguir una implementación escalable y con alta disponibilidad usando los servicios proporcionados por Amazon Web Services [2].

El objetivo de esta plataforma no es otro que ofrecer a los usuarios una forma rápida y cómoda de adquirir entradas para los eventos a los que desean asistir, ayudando a su vez a los promotores de éstos a agilizar las ventas, evitando colas y pudiendo saber de antemano cuánta gente asistirá.

Es por esto por lo que la plataforma se presenta en tres grandes componentes: la webapp principal donde poder ver los diferentes eventos y adquirir entradas; el apartado para los promotores donde crear eventos en su perfil, editarlos y consultar las estadísticas de ventas; y por último un backend robusto que pueda gestionar todas las peticiones, resistiendo picos peticiones y flujos de visita muy cambiantes.

La webapp para los asistentes busca ser intuitiva y ágil, buscando que con pocos clicks los usuarios puedan consultar las discotecas disponibles con sus eventos y agregar las entradas a su cuenta.

La página dedicada para los promotores es un panel de administración, el cual permite que editen y agreguen eventos a su perfil. Además, también les proporciona estadísticas de venta e información sobre los asistentes, siendo útil para conocer mejor a su audiencia.

Por último, los servicios están potenciados por Amazon Web Services (AWS), una plataforma cloud que será útil para crear servicios altamente disponibles y escalables. Además, se ha optado por una arquitectura de microservicios, buscando desacoplar la lógica en componentes independientes que puedan escalar individualmente en función de sus propias necesidades.

Todos estos componentes se completan con tecnologías Infrastructure as Code (IaC)[3], las cuales permiten definir la infraestructura mediante archivos legibles por servicios que generan de forma automática el despliegue necesario y también con una estimación de costes que permita calcular de forma aproximada el importe de mantener la infraestructura en el cloud de forma mensual.

Abstract

This Final Degree Project (TFG) aims to design and implement a web platform for the sale of tickets for nightclubs, festivals and other types of events that acts as Software as a Service (SaaS) [1], focusing on achieving a scalable implementation with high availability using the services provided by Amazon Web Services [2].

The aim of this platform is none other than to offer users a fast and convenient way to purchase tickets for the events they wish to attend, helping event promoters to speed up sales, avoiding queues and knowing in advance how many people will be attending.

This is why the platform is presented in three main components: the main webapp where you can view the different events and purchase tickets; the section for promoters where they can create events in their profile, edit them and consult sales statistics; and finally, a robust backend that can manage all requests, withstanding peaks in requests and very changing visitor flows.

The webapp for attendees aims to be intuitive and agile, so that with a few clicks users can consult the available clubs with their events and add tickets to their account.

The dedicated page for promoters is an administration panel, which allows them to edit and add events to their profile. In addition, it also provides them with sales statistics and attendee information, which is useful to better understand their audience.

Finally, the services are powered by Amazon Web Services (AWS), a cloud platform that will be useful to create highly available and scalable services. In addition, a microservices architecture has been chosen, seeking to decouple the logic into independent components that can scale individually according to their own needs.

All these components are completed with Infrastructure as Code (IaC)[3] technologies, which allow infrastructure to be defined by files that automatically generate the necessary deployment and also with a cost estimate that allows an approximate calculation of the cost of maintaining the infrastructure in the cloud every month.

Tabla de contenidos

1	Introducción.....	1
1.1	Motivación y necesidad del proyecto.....	1
1.2	Objetivos	1
1.3	Planificación.....	2
1.4	Estructura de la memoria	3
2	Alcance del proyecto.....	4
3	Estado del arte	5
3.1	Tecnologías empleadas	5
4	Análisis de requisitos.....	7
5	Desarrollo	9
5.1	Diseño de la arquitectura	9
5.2	Diseño de la base de datos	10
5.3	Desarrollo de los microservicios	12
5.3.1	Ticket Api	13
5.3.1.1	Diseño de la RESTful API	13
5.3.1.2	Implementación	23
5.3.2	Ticket Payment	27
5.3.3	Ticket Generation	28
5.3.4	Ticket Signup.....	30
5.4	Desarrollo de las webapp.....	31
5.4.1	Webapp general	31
5.4.2	Webapp administración	35
5.5	Despliegue en AWS	38
5.6	Estimación de costes.....	41
6	Resultados y conclusiones	43
7	Análisis de Impacto	44
	Bibliografía	45

1 Introducción

1.1 Motivación y necesidad del proyecto

Durante los últimos 15 años la sociedad ha cambiado completamente, los dispositivos “inteligentes” paulatinamente comenzaron a llegar a la vida cotidiana de las personas: teléfonos inteligentes, relojes inteligentes, televisores inteligentes... Estos aparatos electrónicos se unieron a aquellos que llevaban algo más de tiempo entre nosotros, como por ejemplo los ordenadores y portátiles, cambiando de forma radical la manera de comunicarnos y relacionarnos.

Internet y la web tienen gran parte de culpa de este cambio, pues la facilidad de buscar información y obtener respuestas a preguntas a golpe de click ha hecho que las empresas ya no esperen a que sus clientes los busquen, sino que deben ofrecerles soluciones on-line que les permitan solventar sus necesidades desde cualquier parte del mundo.

La industria del entretenimiento no ha sido una excepción, pues ha visto en la transición tecnológica una oportunidad para ofrecer un mejor servicio a sus clientes y a su vez conseguir una reducción de costes gracias a la venta de entradas online. Las taquillas para vender entradas para un festival, o las largas colas para comprar el acceso a una discoteca se podrían ver sustituidas por una plataforma web que permitiese a cualquier persona poder obtener su ticket para el festival que quería asistir en verano o una entrada para la discoteca de su pueblo. Todo esto sin experimentar ninguna cola y únicamente usando su dispositivo móvil u ordenador.

Los beneficios son claros, y la reducción de costes también. La logística e inversión para mantener durante horas o incluso semanas taquillas abiertas que atendiesen largas colas de personas podía desaparecer y transformarse en un gasto único para disponer de una plataforma web que permitiese la compra online.

Este TFG busca brindar a esos promotores de eventos una solución Software as a Service (SaaS) para que no deban invertir miles de euros en su propia plataforma de entradas, sino que pueden contratar un software que les brinde todas las herramientas necesarias para llevar a cabo su objetivo: vender entradas a sus eventos de forma on-line.

1.2 Objetivos

El objetivo de este trabajo es claro: diseñar e implementar una plataforma web que permita a promotores colgar eventos en su perfil para que usuarios puedan comprar las entradas de forma online. Además, el software hará uso de los servicios cloud de Amazon para el despliegue de toda la infraestructura. Los objetivos concretos propuestos son los siguientes:

1. Estudio de los servicios de Amazon Web Services
2. Diseño de la base de datos
3. Diseño y desarrollo del backend
4. Diseño y desarrollo del frontend
5. Automatización de la infraestructura y servicios cloud
6. Estimación de costes

1.3 Planificación

Antes de comenzar cualquier desarrollo software es necesario establecer una planificación que abarque los meses de trabajo y que ayude a alcanzar los objetivos mencionados en el apartado anterior.

La planificación consta de la siguiente lista de tareas junto a la estimación de horas que se emplearán en cada una:

1. Estudiar servicios cloud AWS (25h)
2. Planificar el diseño de la arquitectura (16h)
3. Diseñar y desarrollar la base de datos (10h)
4. Diseñar y desarrollar el backend (90h)
5. Diseñar y desarrollar el frontend (70h)
6. Estimar los costes (4h)
7. Desplegar la aplicación en el cloud (20h)
8. Automatizar la creación de la infraestructura y servicios de AWS utilizados (20h)
9. Documentar el proyecto (40h)
10. Presentar el proyecto (7h)

Cabe destacar que la primera tarea, a pesar de no ser la que más tiempo llevó fue una de las que más importancia tenía. Conocer los servicios cloud ofrecidos por AWS y cómo funcionan es clave para poder usar la infraestructura adecuada que ayude a alcanzar los objetivos de la mejor forma posible. Es por esto por lo que se tomó un curso de AWS para obtener un nivel *Developer Associate* [4].

Usando la lista de tareas, se creó un diagrama de Gantt que repartiese el trabajo durante las semanas del proyecto.

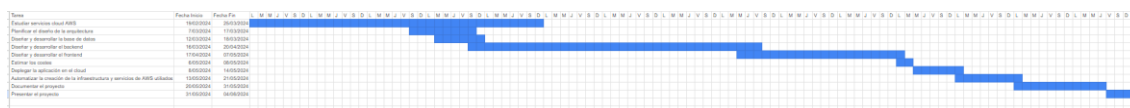


Ilustración I: Diagrama de Gantt

1.4 Estructura de la memoria

A continuación, se describe la estructura de la memoria:

En este primer capítulo se presenta el proyecto, explicando sus objetivos y de qué forma se han estructurado las semanas de trabajo para alcanzarlos.

El segundo capítulo, alcance del proyecto, analiza qué se espera de este proyecto y en qué puede convertirse en un futuro.

El tercer capítulo, estado del arte, hace un breve recorrido sobre alternativas existentes en el mercado relacionadas con el proyecto propuesto y se aprovecha para explicar las tecnologías que se usarán durante el desarrollo, justificando su elección.

En el capítulo análisis de requisitos se describen los requisitos necesarios que debe cumplir la aplicación y que deben estar implementados.

El capítulo de desarrollo es el más extenso, pues indaga en todo el proceso seguido para conseguir una implementación exitosa. En el primer apartado se analiza el despliegue en AWS, explicando cómo se han configurado los servicios y de qué forma interactúan entre sí. El siguiente apartado explica qué arquitectura se ha diseñado para desarrollar el proyecto y qué beneficios aporta. Otro apartado se centra en el diseño que se ha decidido para la base de datos utilizada, esquema, tecnología y los motivos que han justificado dichas decisiones. El diseño del backend y frontend ocupan los apartados 4 y 5. En ellos, se profundiza más sobre las decisiones de diseño e implementación, explicando la utilidad de cada uno de los componentes programados. Por último, se presenta un apartado para estimar los costes generados mensualmente a partir del despliegue en el cloud de AWS.

El capítulo 6 analiza los resultados del proyecto y se recogen las conclusiones del desarrollo del trabajo.

El penúltimo capítulo hace un análisis sobre el impacto que puede tener el en los Objetivos de Desarrollo Sostenible de la Agenda 2030.

Finalmente, se presenta un último capítulo que recoge toda la bibliografía utilizada durante el proyecto.

2 Alcance del proyecto

Este Trabajo de Fin de Grado pretende generar un software que sirva como una base que pueda evolucionar y eventualmente ser una alternativa en el mercado a las plataformas similares ya existentes.

Cuando se alcancen los objetivos planteados, este software ya contará con las funcionalidades básicas para desempeñar su labor sin problemas. Adquirir entradas, ver eventos futuros o descubrir nuevas discotecas son algunas de las características que ya tendrá implementada esta plataforma y que los usuarios podrán realizar. Por su parte, los promotores serán capaces de publicar sus próximas fiestas y ver el número de entradas vendidas en cada una.

Además, al estar desplegado en el cloud en una arquitectura escalable y de alta disponibilidad, permitirá soportar flujos de peticiones variables sin que los usuarios experimenten ningún retraso, error o caída.

La plataforma que se desarrollará a partir de este Trabajo de Fin de Grado, puede ser el inicio de una pequeña empresa que busque ampliar sus características y poder elaborar un producto software atractivo para promotores de todo el mundo que busquen utilizar esta herramienta como tiquetera.

3 Estado del arte

Actualmente existen diversas plataformas SaaS (Software as a Service) orientadas a la gestión de eventos y venta de entradas para pequeños festivales o discotecas. La demanda de este tipo de software a causa de los beneficios mencionados en la introducción, han llevado a diversas empresas a desarrollar sus propias soluciones para lanzarlas al mercado.

Una de estas plataformas es Fourvenues [5], una start-up española que en la actualidad está ganando terreno en varias ciudades de España. Su modelo se basa en una interfaz muy sencilla y amigable, donde con tan solo unos segundos se pueda adquirir una entrada para la discoteca que se oferta. Su modelo se basa en una página simple y amigable por cada uno de sus promotores donde los usuarios pueden adquirir las entradas de sus discotecas favoritas.

Este proyecto pretende ser una nueva plataforma que permita realizar a los clientes las acciones básicas requeridas por el sector: la venta de entradas y consulta de próximos eventos. Además, busca dar un paso más allá y también mejorar las prestaciones respecto a las plataformas existentes aprovechándose de las tecnologías cloud junto a una interfaz amigable y simple para todo tipo de públicos

3.1 Tecnologías empleadas

A continuación, se describen las tecnologías empleadas para la realización de este proyecto, justificando su elección y comparándolas con otras alternativas que se han tenido en consideración.

Para el proyecto se decidió utilizar la plataforma cloud de Amazon Web Services (AWS). AWS permite acceder a diversos servicios orientados a tareas específicas bajo premisas (On premises) con modelo *pay as you go*[6], que como dice el nombre, se paga sólo por los recursos que se utilizan. Este modelo de pago, además, permite al desarrollador despreocuparse del mantenimiento de la infraestructura y verla sólo como un servicio contratable.

Los servicios AWS utilizados se irán describiendo a lo largo del desarrollo del proyecto.

Existen otros proveedores cloud como Google Cloud o Azure (Microsoft) que ofrecen servicios similares. Sin embargo, se ha optado por la opción de Amazon ya que era una plataforma con la que estaba familiarizado.

En cuanto a la arquitectura para los servicios del backend, se ha optado por un modelo basado en microservicios [7]. Esta arquitectura permite desacoplar la lógica en diferentes componentes independientes que pueden interactuar entre sí. Además, permite aprovisionar con recursos diferentes cada uno de los componentes, pues no todos tienen necesidades de escalado iguales.

Cada uno de estos microservicios están programados con JavaScript para ser ejecutados sobre NodeJS [8]. Esta plataforma es un entorno de ejecución

basado en el motor V8 de JavaScript. Como cada uno de estos componentes son independientes, podrían ser programados diferentes lenguajes según las necesidades y su uso. Sin embargo, se ha optado por esta opción por la experiencia previa.

También se hace uso de Docker para automatizar el despliegue de cada uno de los microservicios en *contenedores* que contienen todos los componentes necesarios para ejecutar la aplicación.

Se ha utilizado la herramienta Swagger [9] para documentar y definir las APIs RESTful [10] usadas en algunos de los microservicios. Su curva de aprendizaje es pequeña y para definir las rutas, los métodos y los parámetros que acepta cada una se utiliza el formato YAML.

Para crear las webapp donde los usuarios y promotores pueden interactuar con la plataforma, se ha decidido implementarlo con VueJS [11], un framework de código abierto orientado a la creación de Single Page Applications (SPA). Este tipo de páginas web está cobrando popularidad en los últimos años, pues se caracterizan por contener toda la aplicación en un único archivo html. Las rutas del navegador se gestionan de forma interna y todas redirigen al mismo documento inicial.

Conviene mencionar el entorno de desarrollo usado. Debido a que el despliegue será con los servicios de AWS, se ha decidido usar el IDE basado en navegador de Amazon Web Services, Cloud 9. Este servicio es un IDE multilenguaje que permite a los desarrolladores conectarse con el resto de los servicios cloud de una forma rápida y eficaz, ya que está asociado a la cuenta AWS que se está utilizando.

Una de las ventajas de esta conexión entre IDE y despliegue es la creación de flujos de integración y despliegue continuo (CI/CD) para implementar los cambios realizados al código de forma automática.

Para organizar el trabajo se ha utilizado Trello [12], una herramienta online para administrar proyectos y con la que ya estoy familiarizado.

En cuanto al control de versiones de código se ha elegido git[13]. Un software usado casi como estándar en la industria del desarrollo software que además va a permitir ser el punto de inicio de las integraciones CI/CD que se han mencionado anteriormente.

Por último, se ha utilizado la plataforma Stripe [14] para implementar la pasarela de pago. Crear un procesador de pagos propio convertiría este proyecto en algo mucho más grande y costoso de tiempo. Además, Stripe es una plataforma reconocida para el procesamiento de pagos que diferentes flujos de conexión e integración con servicios externos.

Todas estas tecnologías permitirán llevar a cabo el proyecto propuesto teniendo en cuenta los objetivos planteados en el plan de trabajo.

4 Análisis de requisitos

Conocer los requisitos que la aplicación debe cumplir para que su implementación se considere exitosa es un paso fundamental.

La aplicación define de forma clara dos tipos de usuarios: los usuarios o clientes y los promotores. Los primeros usarán la plataforma para consultar sus entradas, consultar eventos y poder adquirir entradas. Por otro lado, los promotores tendrán la capacidad de publicar en sus perfiles diferentes eventos, cada uno de ellos independientes con su descripción, localización, precios y tramos de entradas propios. Además, también dispondrán de un panel privado para consultar el estado de las ventas.

1. Consulta de promotores y sus eventos

Los usuarios pueden ver las discotecas disponibles en la aplicación y los eventos que tienen programados. Los eventos deben proporcionar información sobre su localización, fecha de inicio y fin, así como los precios de los tramos de compra disponibles.

2. Registro e inicio de sesión

Los usuarios deben poder registrarse e iniciar sesión en la aplicación. Esto es un requisito fundamental a la hora de poder comprar una entrada, pues deben registrarse o haber iniciado sesión previamente.

3. Compra de entradas

Los usuarios deben poder comprar entradas de los eventos y pagar a través de la webapp. Esto implica, poder acceder a un evento, ver su descripción, los tramos de compra disponibles y acceder a la pasarela de pago para realizar la compra. Es necesario que los usuarios estén registrados o hayan iniciado sesión para poder realizar este proceso.

4. Consulta de entradas

Los usuarios deben poder consultar las entradas que han comprado previamente. poder consultar las entradas que ya han comprado, los usuarios deben poder registrarse en la plataforma antes de realizar el pago para que la entrada quede asociada a su perfil.

5. Gestión de eventos

Los promotores deben poder publicar, eliminar y editar los eventos, así como configurar cada uno de ellos de forma independiente. Un concepto clave son los precios y los tramos de entrada asociados a cada uno de ellos.

6. Estadísticas de evento

Los promotores deben poder consultar estadísticas de sus eventos. Esto incluye la información de los asistentes, así como el número de entradas vendidas.

5 Desarrollo

En este capítulo se profundiza en el proceso de desarrollo del trabajo para elaborar una implementación que cumpla con los objetivos y requisitos necesarios que se analizaron en capítulos anteriores.

5.1 Diseño de la arquitectura

Para conseguir una aplicación robusta y escalable, es necesario diseñar una arquitectura que también lo sea. A continuación, se presenta un esquema de cómo se distribuyen los diferentes componentes que forman esta arquitectura, y también cómo interactúan entre sí.

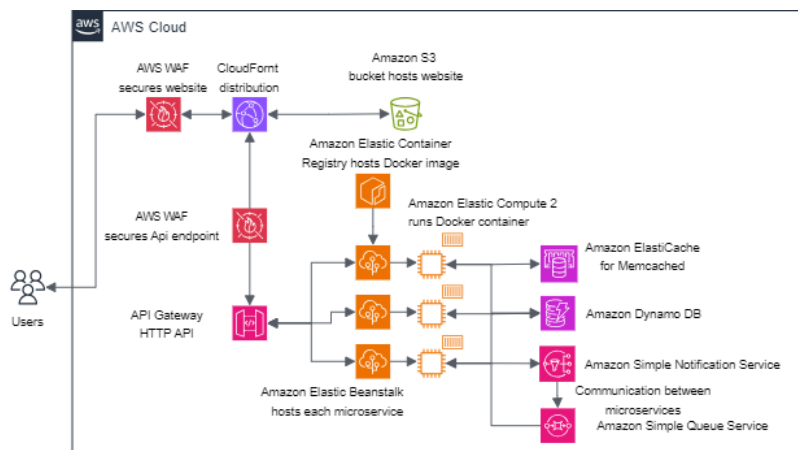


Ilustración II: Arquitectura de la aplicación

El punto de entrada como se puede observar es AWS Cloudfront, una CDN que servirá los archivos estáticos (webapps y tickets) a los usuarios. Además, delante se posiciona AWS WAF, un firewall que se encarga de filtrar las peticiones maliciosas para que no lleguen al servicio. Los archivos que sirve están alojados en un cubo o bucket de Amazon S3.

La entrada al backend se produce a través de una HTTP API Gateway, la cual es el punto común de entrada que se encarga de distribuir las peticiones a los diferentes microservicios en función de su URI. Cada uno de estos microservicios está desplegado mediante AWS Elastic Beanstalk, que se encarga de aprovisionar los recursos necesarios como un balanceador de carga, instancias EC2 para alojar las imágenes Docker o los grupos de autoescalado. Cada uno de estos servicios se ejecuta sobre un contenedor Docker que contiene las aplicaciones escritas en NodeJS para llevar a cabo la lógica de negocio. Además, cada microservicio está configurado con reglas de autoescalado distintas en función de sus características para conseguir un despliegue altamente disponible y escalable.

Un objetivo importante para reducir la carga sobre la base de datos es poder almacenar en caché durante cierto tiempo algunos de los datos que más se sirven a los usuarios a través de las RESTfulAPI. Esto implica analizar los datos que se envían y establecer de cuales se sacaría ventaja gracias al uso de un

servidor de caché como Amazon ElastiCache for Memcached junto a estrategias para mantener los datos coherentes en la base de datos alojada en Amazon DynamoDB.

Finalmente, los microservicios usarán Amazon Simple Notification Service para publicar eventos sobre la lógica de negocio y poder mantener comunicaciones entre sí de forma asíncrona. Algunos de los suscriptores a estos eventos usarán colas proporcionadas por Amazon SQS que permitan que sólo uno de los microservicios reciba el evento.

5.2 Diseño de la base de datos

La base de datos utilizada, como ya se ha mencionado anteriormente, es proporcionada por Amazon DynamoDB, de tipo NoSQL. Este tipo de tecnología están diseñadas para escalar fácilmente y almacenar documentos con esquemas flexibles. Están diseñadas para ofrecer un alto rendimiento, con tasas de lectura muy altas que permitan escalar de forma flexible.

Para implementar la aplicación, se han definido tres tablas: *clients*, *venues* y *tickets*.

La primera almacena la información de los promotores: su dirección, descripción, nombre e icono del perfil. A su estructura se le agrega un campo *client_id* el cual funciona como *partition key* de la tabla y como identificador en la lógica de negocio.

Attribute name	Value	Type
client_id - Partition key	discoteca_upm	String
address	C/Padre Damián 23	String
description	Discoteca oficial de la UPM gestionada por alumnos de la ETSIINF	String
name	Discoteca Oficial UPM - ETSIINF	String
profile_icon	https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcSy9oFLiu-f3i3tfu5-xb6LdWT7wV9HZeWk0sbVNdmeQ&s	String

Ilustración III: Elemento en la tabla *clients*

La segunda tabla es una de las más importantes y que más consumo tendrá ya que almacenará todos los eventos disponibles en cada uno de los clientes. La *partition key* es el *client_id*, y la *sort key* es el *venue_id*. Estas dos claves son fundamentales para diseñar una tabla eficiente y que escale correctamente ya que determinan cómo se distribuye la información en los nodos de almacenamiento y de qué forma se ordena dentro de cada uno de estos nodos.

Otros de los atributos que contiene cada evento es su dirección, el título y una descripción. Las fechas de inicio del evento y fin se almacenan usando la norma ISO8601 [16].

Attribute name	Value	Type
client_id - Partition key	prueba_1	String
venue_id - Sort key	fiesta_inauguracion_numer_2027_5_29	String
address	C/Cea Bermudez 17	String
description	Evento increible para estrenar la discoteca	String
end_date	2027-05-31T20:21	String
image	https://i.imgur.com/d3sVdHS.jpeg	String
start_date	2027-06-29T08:21	String
tickets_sections	<button>Insert a field ▼</button>	List
title	Fiesta Inauguracion numero 2	String

Ilustración IV: Elemento en la tabla venues

Además, como se puede observar en la Ilustración IV, cada evento también dispone de un atributo denominado *tickets_sections* que almacena la información de los tramos de entradas disponibles para comprar. Es una lista que contiene objetos con atributos sobre el precio (en céntimos) de el tramo, el título junto a una descripción, el número de entradas disponibles, así como la cantidad restante que quedan por vender. Además, también dispone de otro atributo tipo *boolean* para activar o desactivar el tramo.

tickets_sections	<button>Insert a field ▼</button>	List
0	<button>Insert a field ▼</button>	Map
available	<input checked="" type="radio"/> True <input type="radio"/> False	Boolean
description	2 copas hasta las 3, después 1 copa	String
id	0	Number
price	2000	String
tickets_left	300	Number
title	Tramo normal	String
total_tickets	300	Number

Ilustración V: Atributo tickets_sections de un elemento en la tabla venues

Por último, está la tabla *tickets*, encargada de almacenar las entradas compradas por los usuarios en cada uno de los eventos. La *partition key* es la id de usuario generada por AWS Cognito y la *sort key* el identificador único generado para la entrada. Cada elemento también contiene la id del cliente y la id del evento a la que pertenece la entrada junto al identificador del tramo que se ha comprado. Otro atributo es la confirmación de pago, que está en falso cuando aún no se ha completado el pago y en verdadero cuando ya se ha procesado. Cuando la transacción está hecha, se agrega un campo de fecha cuando se ha pagado y también la cantidad. Si no se llega a realizar el pago, el

atributo *time_to_expire* indica a partir de qué hora se puede cancelar la reserva. Toda esta lógica de negocio se tratará más adelante cuando se explique el desarrollo de los microservicios.

user_id - Partition key	<input type="text" value="31d970de-3031-70e9-7541-aada543ba774"/>	String
ticket_id - Sort key	<input type="text" value="efd7e4dc-9ca4-4a26-a237-7ebd281eed80"/>	String
client_id	<input type="text" value="prueba_1"/>	String
confirmed_payment	<input checked="" type="radio"/> True <input type="radio"/> False	Boolean
ticket_section_id	<input type="text" value="0"/>	String
time_to_expire	<input type="text" value="2024-06-02T21:14:33.298Z"/>	String
venue_id	<input type="text" value="fiesta_inauguracion_2024_4_11"/>	String
purchase_date	<input type="text" value="2024-06-02T21:15:33.298Z"/>	String
paid_price	<input type="text" value="2000"/>	Number

Ilustración VI: Elemento en la tabla tickets

Esta tabla presenta dos patrones de acceso distintos, en base al identificador de usuario o en base al identificador del evento. Es por este motivo por el que esta tabla también contiene un index global secundario, característica de DynamoDB que permite tener una segunda proyección de los datos en base a otra clave; en este caso usando como *partition key* el identificador del evento y como *sort key* el identificador del ticket.

5.3 Desarrollo de los microservicios

Como ya se ha mencionado anteriormente en esta memoria, el backend sigue una arquitectura de microservicios, piezas independientes que contienen parte de la lógica de negocio que pueden comunicarse entre sí. Durante el diseño se definieron los siguientes 5 microservicios:

- **Ticket Api:** Este microservicio será uno de los más importantes y también el que más carga soportará. Su función principal es exponer una RESTful API para ser consumida por las webapps.
- **Ticket Payment:** Este microservicio gestionará la lógica de la pasarela de pago: generar una sesión y notificar al resto de microservicios cuando se haya completado el pago de una entrada.

- **Ticket Generation:** Su función es generar y almacenar en un almacenamiento persistente las entradas de un usuario una vez que haya completado el pago.
- **Ticket Signup:** Es el encargado de ofrecer una API para que los usuarios puedan registrarse e iniciar sesión en las webapps, generando tokens de sesión. Este microservicio se basa en exponer las funcionalidades que son ofrecidas por el servicio Amazon Cognito, donde estarán las cuentas de los usuarios.

5.3.1 Ticket Api

Programado en NodeJS, utiliza ExpressJS, una librería para desarrollar aplicaciones web y APIs de forma sencilla y ordenada.

Antes de implementar la RESTful API que este servicio expone, se definió utilizando la herramienta Swagger mediante OpenAPI. Se documentó cada método soportado en cada URI junto a sus parámetros de entrada y salida para generar un archivo de referencia que se pudiese utilizar durante la implementación para agilizar el proceso. Generar documentación para cualquier proyecto software es un proceso fundamental para permitir a otros compañeros, o futuros desarrolladores en el proyecto conocer mejor su funcionamiento de una forma sencilla.

Todas las peticiones aceptan y producen únicamente contenido en JSON ya que agiliza las tareas en el backend y frontend ya que están programados en JavaScript.

5.3.1.1 Diseño de la RESTful API

Una de las partes más importantes en el diseño de la API fue establecer qué tipo de paginación usar. En la industria se utilizan diversas soluciones, siendo la más conocida la paginación basada en páginas de tamaño fijo. El servidor devuelve siempre páginas de un tamaño determinado y el cliente es el encargado de especificar a partir de qué página se debe empezar (offset). Sin embargo, aprovechando que DynamoDB trabaja con cursores para devolver más resultados, la paginación de esta API también trabaja con cursores.

El cliente debe especificar el límite de elementos que quiere como máximo, y en caso de que tenga un cursor, también puede agregarlo a la petición para informar al backend desde qué ítems se debe empezar a trabajar.

Todas las respuestas con paginación devuelven un atributo *count* con el número de resultados devuelto, un atributo *content* que siempre es un array con los elementos devueltos y un atributo *cursor*, que puede ser null si no hay más elementos que retornar o ser el valor que hay que pasar al servidor en la siguiente llamada para obtener más resultados.

client Todo sobre clientes/promotores. Un cliente puede tener en su perfil varios eventos activos			^
GET	/client	Obtener clientes	▼
GET	/client/{clientId}/	Obtener un cliente	▼
GET	/client/{clientId}/venue/	Obtener los eventos asociados al cliente	▼
POST	/client/{clientId}/venue/	Crea un evento	▼
GET	/client/{clientId}/venue/{venueId}	Obtener el evento asociado al cliente	▼
PUT	/client/{clientId}/venue/{venueId}	Crea un evento	▼
GET	/client/{clientId}/venue/{venueId}/ticket	Obtener los tickets asociados al evento	▼
GET	/client/{clientId}/venue/{venueId}/ticket/{ticketId}	Obten el ticket asociado al evento	▼

Ilustración VII: Documentación de Swagger

tickets Todo sobre entradas			^
GET	/user/{userId}/ticket	Obtener los tickets de un usuario	▼
POST	/user/{userId}/ticket	Crea un ticket para el usuario	▼
GET	/user/{userId}/ticket/{ticketId}	Obtener los tickets de un usuario	▼

Ilustración VIII: Documentación de Swagger 2

A continuación, se describen las URIs diseñadas para este microservicio:

- **GET /client**
 - Devuelve los clientes existentes en la plataforma
 - Usa paginación
 - Respuesta 200 OK si la operación se ha realizado con éxito

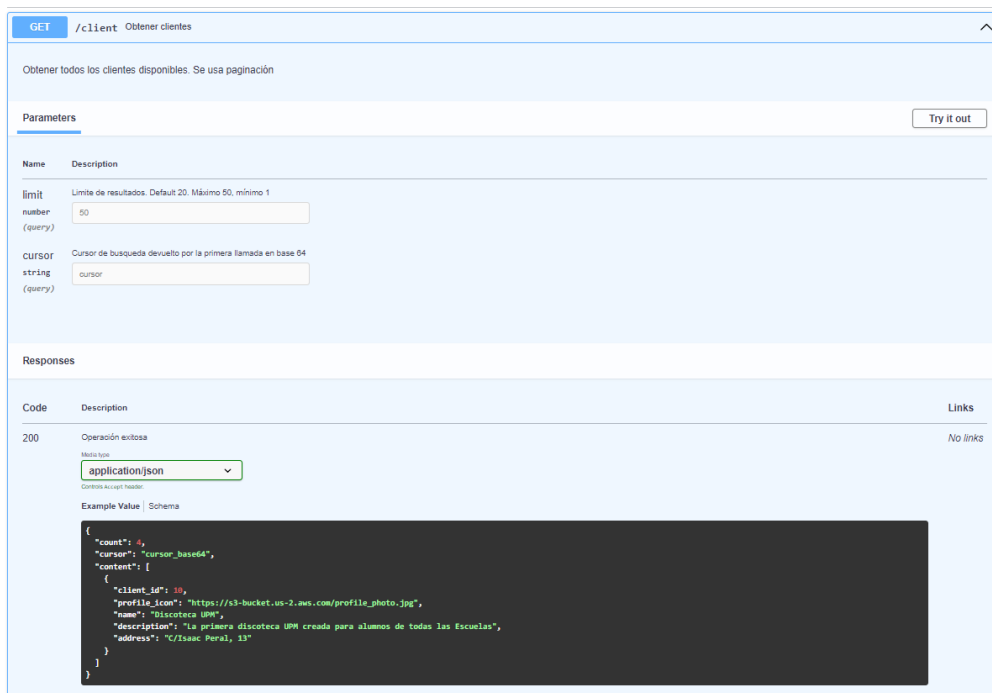


Ilustración IX: GET /client

- **GET /client/:clientId**
 - Devuelve el cliente con id :clientId
 - Respuesta 200 OK si el cliente existe o 404 Not Found si no se ha encontrado.

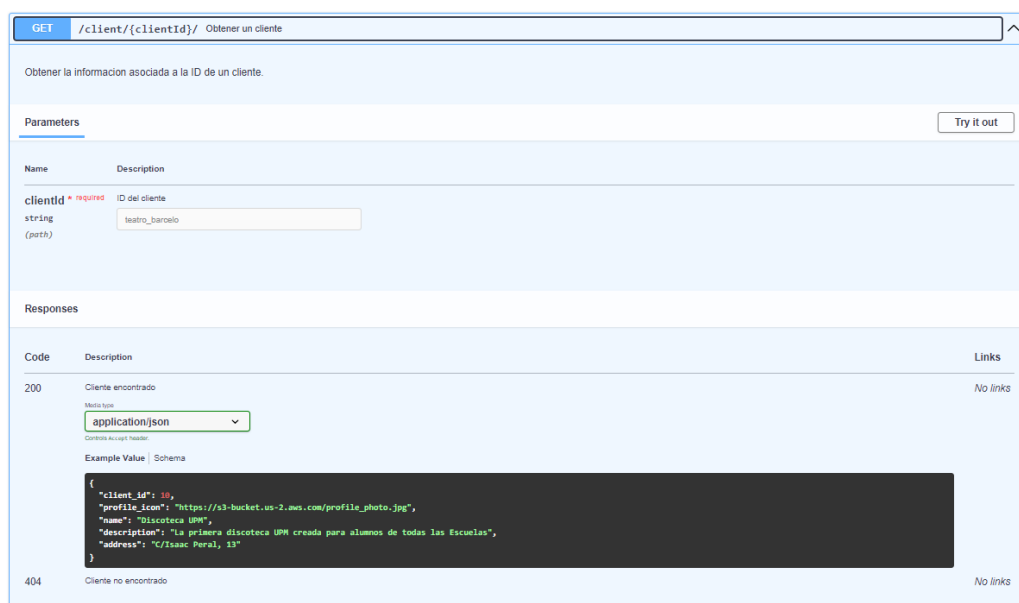


Ilustración X: GET /client/:clientId

- **GET /client/:clientId/venue/**
 - Devuelve los eventos registrados en el perfil del cliente con id :clientId
 - Usa paginación
 - Respuesta 200 OK si el cliente existe o 404 Not Found si no se ha encontrado.

GET /client/{clientId}/venue/ Obtener los eventos asociados al cliente

Se devuelven todos los eventos activos asociados al cliente. Se usa paginación

Try it out

Name	Description
clientId ^{required} string (path)	ID del cliente Isidro_barcelo
limit number (query)	Límite de resultados. Default 20. Máximo 50, mínimo 1 50
cursor string (query)	Cursor de búsqueda devuelto por la primera llamada en base 64 cursor

Responses

Code	Description	Links
200	Operación exitosa Mediante: application/json Content-Accept Header: Example Value Schema	No links
404	Cliente no encontrado	No links

```
{
  "count": 4,
  "cursor": "cursor_base64",
  "content": [
    {
      "venue_id": 10,
      "image": "https://textimgtest.com",
      "start_date": "10/10/2020",
      "end_date": "11/10/2020",
      "address": "C/Isaac Peral, 13",
      "title": "Evento especial fin exámenes ETSINF",
      "description": "Ven a celebrar que ya terminaron los exámenes de la ETSINF",
      "tickets_section": [
        {
          "id": 1,
          "title": "TRAMO 1 ENTRADAS",
          "price": 1000,
          "available": true,
          "tickets_left": 100,
          "total_tickets": 400
        }
      ]
    }
  ]
}
```

Ilustración XI: GET /client/:clientId/venue/

- **POST /client/:clientId/venue/**
 - Crea un evento en el perfil de :clientId
 - El cuerpo de la petición lleva la información del evento
 - Respuesta 201 Created se si ha creado el evento correctamente. Se devuelve 401 si no está autenticado o 403 si no está autorizado a crear un evento.

POST: /client/{clientId}/venue/ Crea un evento

Crea un evento en la cuenta del cliente

Try it out

Parameters

Name	Description
clientId <small>required</small>	ID del cliente
string	
(path)	

Request body

application/json

El evento a crear:

Example Value | Schema

```
{
  "venue_id": 0,
  "image": "https://testingtest.com",
  "start_date": "20/04/2020",
  "end_date": "21/04/2020",
  "address": "C/Gran Perla, 50",
  "title": "Evento especial fin semana STYLING",
  "description": "Ven a celebrar que ya tendremos los salones de la STYLING",
  "ticket_section": [
    {
      "id": 1,
      "title": "SEÑO i ENTRADA",
      "price": 1000,
      "available": true,
      "tickets_left": 100,
      "total_tickets": 100
    }
  ]
}
```

Response

Code	Description	Link
201	Evento creado	Link
401	No autorizado	Link
403	No tiene permisos. Puede ser que esté creando un evento a otro cliente	Link

Ilustración XII: POST /client/:clientId/venue/

- **GET /client/:clientId/venue/:venueId**
 - Devuelve el evento con la id :venueId del perfil del cliente :clientId
 - Respuesta 200 OK si existe o 404 Not Found si no existe el cliente o el evento.

GET /client/{clientId}/venue/{venueId} Obtener el evento asociado al cliente

Se devuelve la información del evento

Parameters Try it out

Name	Description
clientId required	ID del cliente
string (path)	teatro_barcelo
venueId required	ID del evento
string (path)	evento_03_marzo_embassy

Responses

Code	Description	Links
200	Operación exitosa	No links
	<p>Media type: application/json</p> <p>Content-accept header:</p> <p>Example Value Schema</p> <pre>{ "venue_id": 10, "image": "https://textingtest.com", "start_date": "10/10/2024", "end_date": "11/10/2024", "address": "C/ Isaac Peral, 13", "title": "Evento especial fin exámenes ETSIINF", "description": "Ven a celebrar que ya torcedaron los exámenes de la ETSIINF", "tickets_sections": [{ "id": 1, "title": "TRAMO 1 ENTRADAS", "price": 1500, "available": true, "tickets_left": 100, "total_tickets": 400 }] }</pre>	
404	Evento no encontrado	No links

Ilustración XIII: GET /client/:clientId/venue/:venueId

- **PUT /client/:clientId/venue/:venueId**
 - Actualiza el evento con id :venueId del perfil del cliente con id :clientId.
 - Por motivos de reglas de negocio y coherencia con la información, no se permite borrar tramos de entradas ni editar el título ni la fecha.
 - Devuelve el recurso actualizado
 - Respuesta 200 OK si existe; 404 Not Found si no existe el cliente o el evento; 403 Forbidden si no está autorizado a acceder al recurso; 401 Unauthorized si no está autenticado

PUT /client/{clientId}/venue/{venueId} Crear un evento

Actualiza un evento en la cuenta del cliente

Parameters Try it out

Name	Description
clientId <small>required</small>	ID del cliente
string (path)	
venueId <small>required</small>	ID del evento
string (path)	

Request body application/json

El evento se actualizará.

Example Value Schema

```
{
  "venue_id": 10,
  "image": "https://fakeimgtest.com",
  "start_date": "04/04/2020",
  "end_date": "11/04/2020",
  "address": "C/Alonso Peral, 55",
  "title": "Evento especial sin asientos STL123",
  "description": "Ven a celebrar que ya tendremos los asientos de la STL123",
  "tickets_section": [
    {
      "id": 1,
      "title": "SÓLO 1 ENTRADA",
      "price": 1000,
      "available": true,
      "tickets_left": 100,
      "total_tickets": 100
    }
  ]
}
```

Responses

Code	Description	Link
201	Evento creado	No limit
401	No autorizado	No limit
403	No tiene permisos. Puede ser que esté creando un evento a otro cliente	No limit
404	No existe el recurso	No limit

Example Value Schema

```
{
  "venue_id": 10,
  "image": "https://fakeimgtest.com",
  "start_date": "04/04/2020",
  "end_date": "11/04/2020",
  "address": "C/Alonso Peral, 55",
  "title": "Evento especial sin asientos STL123",
  "description": "Ven a celebrar que ya tendremos los asientos de la STL123",
  "tickets_section": [
    {
      "id": 1,
      "title": "SÓLO 1 ENTRADA",
      "price": 1000,
      "available": true,
      "tickets_left": 100,
      "total_tickets": 100
    }
  ]
}
```

Ilustración XIV: PUT /client/:clientId/venue/:venueId

- **GET /client/:clientId/venue/:venueId/ticket**
 - Devuelve las entradas asociadas al evento con la id :venueId del perfil del cliente :clientId
 - Usa paginación
 - Respuesta 200 OK si existe; 404 Not Found si no existe el cliente o el evento; 403 Forbidden si no está autorizado a acceder al recurso

GET /client/{clientId}/venue/{venueId}/ticket Obtener los tickets asociados al evento

Se devuelve los tickets asociados al evento

Parameters Try it out

Name	Description
clientId required string (path)	ID del cliente teatro_barcelo
venueId required string (path)	ID del evento evento_03_marzo_embassy

Responses

Code	Description	Links
200	Operación exitosa Media type: application/json Content type: json Example Value Schema	No links
403	No puedes acceder a esta información	No links
404	Evento no encontrado	No links

```
{
  "count": 1,
  "cursor": "cursor_base64",
  "content": [
    {
      "user_id": "31d9796e-3031-70e9-7541-aada543ba774",
      "venue_id": "evento_03_marzo",
      "ticket_id": "1d986a7c-4c8d-4618-a179-c237666473e6",
      "client_id": "discofeca_1",
      "ticket_section_id": 1,
      "paid_price": 1000,
      "purchase_date": "2024-06-02T21:09:39.041Z",
      "time_to_expire": "2024-06-02T21:09:39.041Z",
      "confirmed_payment": true
    }
  ]
}
```

Ilustración XV: GET /client/:clientId/venue/:venueId/ticket

- **GET /client/:clientId/venue/:venueId/ticket/:ticketId**
 - Devuelve la entrada con id :ticketId asociada al evento con la id :venueId del perfil del cliente :clientId
 - Respuesta 200 OK si existe; 404 Not Found si no existe el cliente, el evento o la entrada; 403 Forbidden si no está autorizado a acceder al recurso

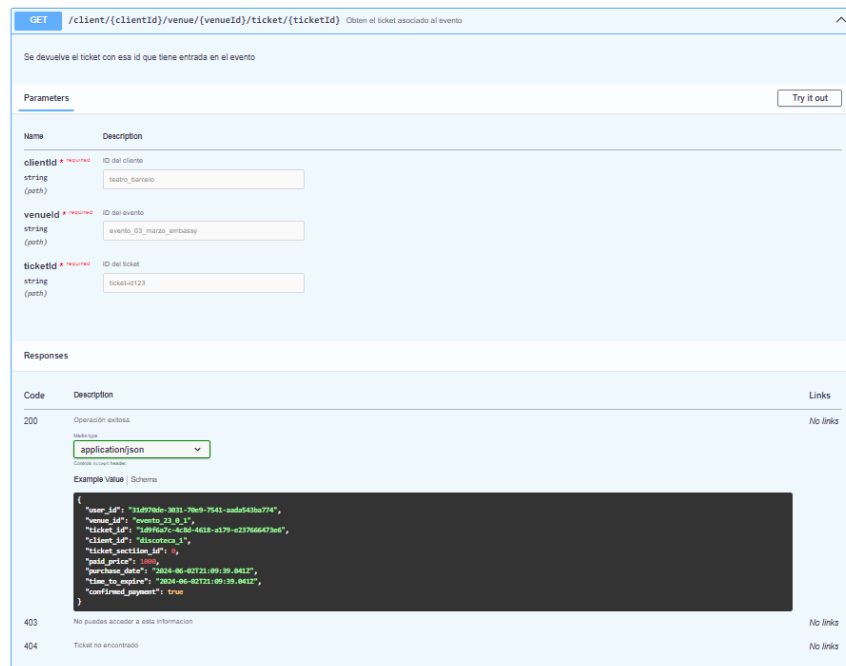


Ilustración XVI: GET /client/:clientId/venue/:venueId/ticket/:ticketId

- **GET /user/:userId/ticket**
 - Devuelve las entradas que pertenecen al usuario con id :userId
 - Usa paginación
 - Respuesta 200 OK si la operación es exitosa; 403 Forbidden si no está autorizado a acceder al recurso

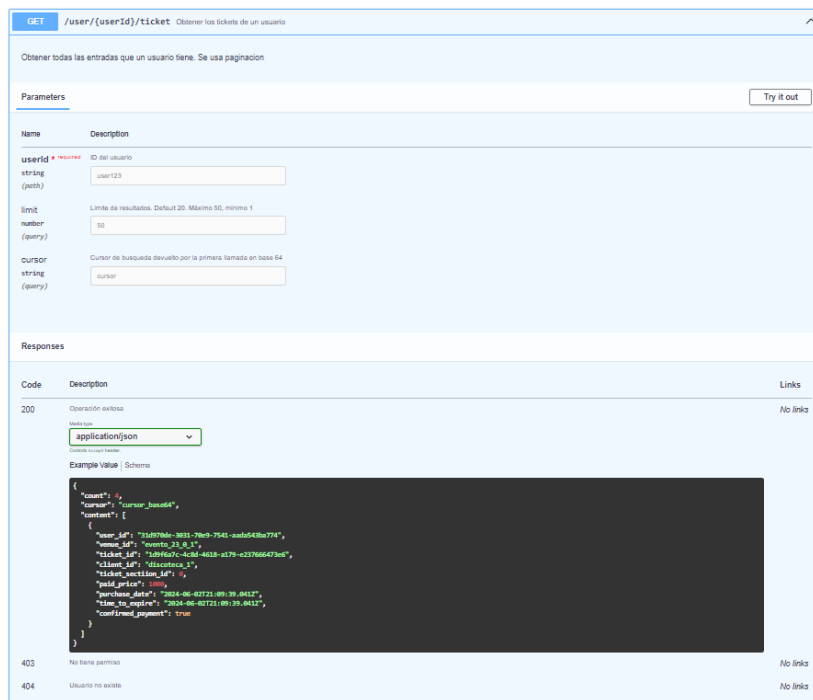
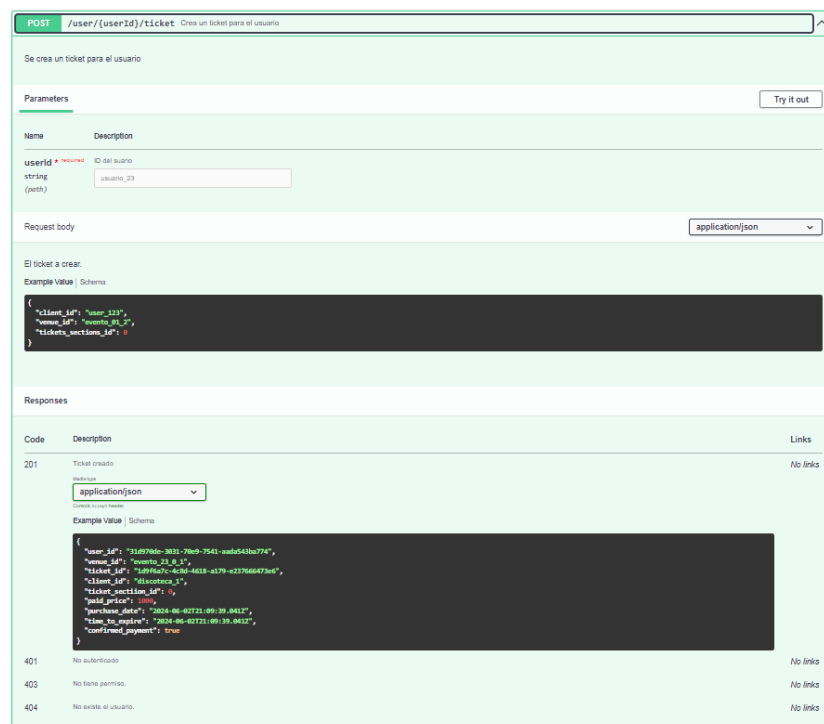


Ilustración XVII: GET /user/:userId/ticket

- **POST /user/:userId/ticket**

- Crea una reserva de ticket para el usuario con id :userId
- El cuerpo de la petición es información sobre la entrada que se quiere comprar
- Devuelve la URL para realizar el pago.
- Respuesta 200 OK si la operación es exitosa; 403 Forbidden si no está autorizado a acceder al recurso; 404 Not Found si el usuario no existe.



Swagger UI for the endpoint **POST /user/:userId/ticket**. The interface shows the following details:

- Operation:** POST /user/:userId/ticket. Description: Crea un ticket para el usuario.
- Parameters:** A table with columns 'Name' and 'Description'. It lists a required parameter `userId` (string, path) with the value `usuario_23`.
- Request body:** Set to `application/json`. The description is 'El ticket a crear.' An example JSON body is shown:

```
{  "client_id": "user_123",  "venue_id": "evento_01_0",  "tickets_section_id": 0}
```
- Responses:** A table with columns 'Code', 'Description', and 'Links'.
 - 201:** Ticket creado. Link: No links. Example JSON response:

```
{  "user_id": "31d976dc-3831-70eb-7541-aada543be774",  "venue_id": "evento_23_0_1",  "ticket_id": "889f8a7c-82d1-4828-a579-e23766847bed",  "client_id": "discooteca_1",  "ticket_section_id": 0,  "paid_price": 1000,  "purchase_date": "2024-06-03T21:09:39.861Z",  "time_to_expire": "2024-06-03T21:09:39.861Z",  "confirmed_payment": true}
```
 - 401:** No autorizado. Link: No links.
 - 403:** No tiene permiso. Link: No links.
 - 404:** No existe el usuario. Link: No links.

Ilustración XVIII: POST /user/:userId/ticket

- **GET /user/:userId/ticket/:ticketId**

- Devuelve el ticket con id :ticketId del usuario con id :userId
- Respuesta 200 OK si la operación es exitosa; 403 Forbidden si no está autorizado a acceder al recurso; 404 Not Found si el usuario no existe.

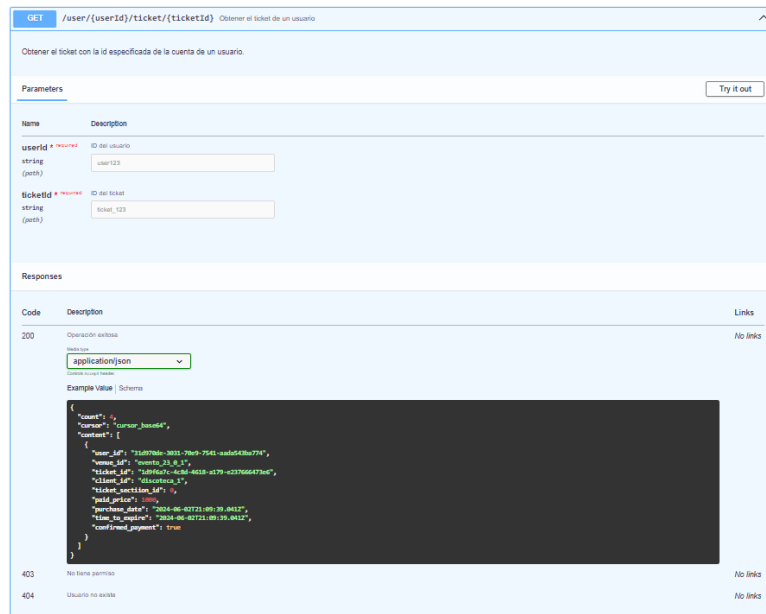


Ilustración XIX: GET /user/{userId}/ticket/{ticketId}

5.3.1.2 Implementación

Para realizar la implementación de este microservicio se siguió una metodología lo más parecida a Test Driven Development [15]. Esta técnica de desarrollo de software consiste en guiar la implementación escribiendo primero los tests. Se elige una funcionalidad que se desea implantar, se escriben sus casos de prueba y a continuación se escribe el código.

Cuando la funcionalidad pase los casos de prueba definidos, comienza la parte de refactorizar el código.

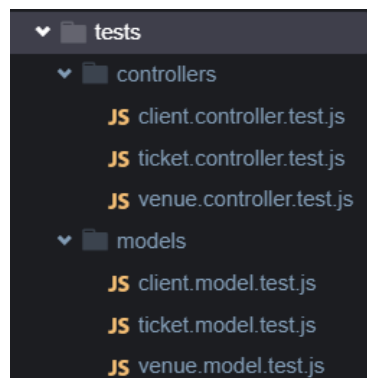


Ilustración XX: Carpeta con los testers del microservicio

El proyecto se divide en diferentes carpetas según los componentes que forman el código:

- **Routes:** En esta carpeta se almacenan los archivos que definen las rutas accesibles en la API.

```
const express = require('express');
const router = express.Router();

const paginationMiddleware = require('../middlewares/pagination.middleware');

const ticketController = require('../controllers/ticket.controller');
const authenticationMiddleware = require('../middlewares/authentication.middleware');
const authorizationMiddleware = require('../middlewares/authorization.middleware');

router.get("/user/:userId/ticket", authenticationMiddleware.verifyAwsUserMiddleware, authorizationMiddleware.performActionOnSelfUser,
  paginationMiddleware.paginationLimit, paginationMiddleware.paginationFullCursor, ticketController.getAllByUserId);
router.get("/user/:userId/ticket/:ticketId", ticketController.getTicketByUserId);
router.post("/user/:userId/ticket", authenticationMiddleware.verifyAwsUserMiddleware, authorizationMiddleware.performActionOnSelfUser, ticketController.create);
module.exports = router;
```

Ilustración XXI: ticket.router.js

Como se observa en la ilustración, se crea un enrutador en el cual se definen las rutas disponibles, junto a la cadena de middlewares por la cual parará la petición hasta llegar al controlador.

- **Middlewares:** Esta carpeta agrupa archivos que contienen funciones reutilizables para realizar ciertas tareas como verificar y sanear las peticiones antes de que alcancen al controlador.

```
exports.verifyAwsUserMiddleware = async (req, res, next) => {
  try {
    if(isTest){
      req.auth = { access_id: {} };
      req.auth.given_name = "test";
      req.auth.family_name = "test";
      return next();
    }

    const token = req.headers["authentication"];
    if(!token || token == null) {
      return res.status(401).send();
    }

    const decoded = await verifyToken(token);

    req.auth = {
      access_id: decoded;
    };
    next();
  } catch(err) {
    console.log("401", err);
    res.status(401).send();
  }
};
```

Ilustración XX: Fragmento de authentication.middleware.js

Este fragmento de uno de los middlewares presentes en este microservicio es el encargado de verificar el token de autenticación de las peticiones que así lo requieran. En caso de no ser válido o no estar presente, devolvería un error tipo 403.

- **Controllers:** Aquí se encuentran los archivos que definen los controladores que recibirán las peticiones de la API, los cuales contienen la lógica de negocio.

```
exports.getOne = (req, res) => {
  try {
    const clientId = req.params.clientId;

    memcached.get('client:' + clientId, (err, data) => {
      if(err) {
        return res.status(500).json({message: 'Error occurred while fetching client'});
      }

      if(data) {
        res.setHeader('Content-Type', 'application/json');
        res.setHeader('Cache-Status', 'From Memcached');
        res.end(data);
      } else {
        client.getClientById(clientId).then((data) => {
          if(data) {
            const stringifiedData = JSON.stringify(data);
            memcached.set('client:' + clientId, stringifiedData, cacheTtlInSec, (err) => {
              if(err) {
                return res.status(500).json({message: 'Error occurred while caching client'});
              }

              res.setHeader('Content-Type', 'application/json');
              res.end(stringifiedData);
            });
          } else {
            res.status(404).json({message: 'Client not found'});
          }
        }).catch((err) => {
          console.log(err);
          return res.status(500).json({message: 'Error occurred while fetching client'});
        });
      }
    });
  } catch(err) {
    console.log(err);
    res.status(500).json({message: 'Unknown error'});
  }
}
```

Ilustración XX: Fragmento de client.controller.js

Este fragmento de código pertenece a uno de los controladores, y es una función encargada de la URI `/client/:clientId`. Usa un servidor de caché para aplicar lógica write-through. Si no encuentra en la caché el cliente, lo busca en DynamoDB y lo lleva a caché en caso de que exista. Devuelve al usuario código 200 con el cliente en caso de que exista, o código 400 si no.

En el proceso pueden ocurrir excepciones, que son manejadas por el controlador y devueltas al cliente como un error 500.

- **Services:** En esta carpeta se proporciona un servicio para crear una sesión de pago a través de otro microservicio. La conexión requiere que sea síncrona, por lo que se opta por el uso de una petición HTTP GET que retorne la URL donde redirigir al usuario para el pago.

```
{const axios = require('axios');
const config = require('../../config/config');

const createPaymentSession = (userId, given_name, family_name, ticketId, venue, price, ticketSectionId) => {
  return new Promise((resolve, reject) => {
    axios.post(config.PAYMENT_PROCESSOR_SERVICE_URL + "/session", {
      user: { user_id: userId, given_name: given_name, family_name: family_name },
      ticket_id: ticketId,
      venue: venue,
      price: price,
      ticket_section_id: ticketSectionId
    }).then((response) => {
      resolve(response.data.url);
    }).catch((err) => {
      reject(err);
    });
  });
}

module.exports = {
  createPaymentSession: createPaymentSession
}
```

Ilustración XXI: payment.service.js

- **Config:** Contiene un archivo con las variables de configuración predeterminadas. Primero busca si la variable existe en las variables de entorno del sistema para usar su valor, pero si no se encuentra toma el valor por defecto.
- **Models:** Esta carpeta contiene un archivo por cada tabla que define las operaciones que se pueden realizar sobre ella: crear, eliminar actualizar...

```
exports.getClientById = (clientId) => {
  return new Promise(async (resolve, reject) => {

    const input = { // GetItemInput
      TableName: config.DYNAMO_CLIENTS_TABLE, // required
      Key: { // Key // required
        "client_id": clientId
      }
    }

    try {

      const docClient = createClient();
      const command = new GetCommand(input);
      const response = await docClient.send(command);
      resolve(response.Item);
    } catch(err){
      reject(err);
    }
  });
};
```

Ilustración XXII: Fragmento de client.model.js

Esta función pertenece a una de las carpetas de modelo. Sirve para buscar en la tabla *clients* de DynamoDB el elemento cuya *partition key*: *client_id* sea igual al primer argumento.

- **Cache:** Contiene la lógica para establecer una conexión al servidor caché deseado. En este despliegue es el servicio mencionado anteriormente Amazon ElastiCache For Memcached.

Como se mencionó anteriormente, este microservicio hace uso de un servidor cache en ciertos controladores. Tras analizar el patrón de acceso a los datos, resalta que muchas de las operaciones que más se repetirán son lecturas. Es por este motivo por el que para aliviar la carga de la base de datos y así reducir su consumo, estos valores se almacenan en caché durante un tiempo Time-To-Live (TTL). Cuando este valor se excede, la información se considera obsoleta y la siguiente petición alcanzará a la base de datos.

Esta solución es muy eficaz, sin embargo, conlleva un riesgo, ya que si se almacena en caché información incorrecta o no se añade la lógica necesaria para invalidar la información manualmente cuando se produce una actualización o se elimina, puede producir graves incoherencias dentro de la aplicación.

5.3.2 Ticket Payment

Este microservicio es el encargado de gestionar las sesiones de pago. Para ello usará el SDK de Stripe.

La estructura de carpetas es igual a la del primer servicio: config, controllers, middlewares, models, routes, services y tests.

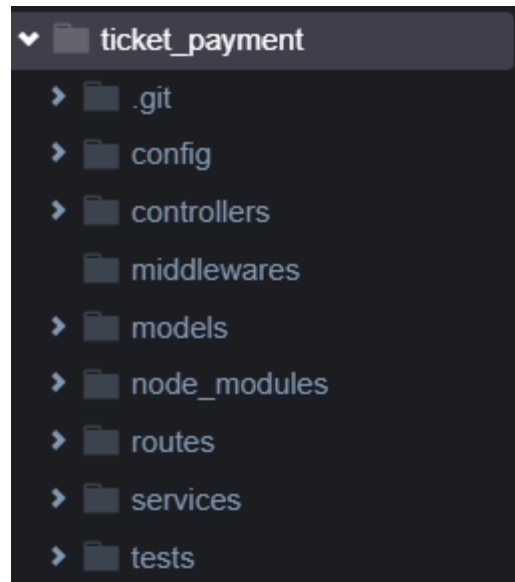


Ilustración XXIII: Carpetas del microservicio ticket payment

Además, también se usa una metodología Test Driven Development.

Este servicio expone un pequeño servidor HTTP por el que recibe peticiones desde el microservicio *ticket api*, proporcionando una conexión síncrona. Cuando recibe una petición, crea una sesión de pago en Stripe, y devuelve la URL del pago.

```
const StripeService = require('../services/stripe');
const { body, validationResult } = require('express-validator');

//Creado por nuestros servicios
exports.createSession = [
  body('price').trim().isNumeric(),
  body('ticket_id').trim().isString().isLength({min: 1}),
  body('venue').notEmpty(),
  body('venue.image').trim().isString(),
  body('venue.title').trim().isString(),
  body('venue.address').trim().isString(),
  body('user.user_id').trim().isString(),
  body('user.given_name').trim().isString(),
  body('user.family_name').trim().isString(),
  body('ticket_section_id').trim().isString(),
  async (req, res) => {
    try {
      const errors = validationResult(req);
      if (!errors.isEmpty()) {
        const errArray = errors.array();
        let textErr = '';
        for (let i = 0; i < errArray.length; i++) {
          textErr += `${errArray[i].msg} + " x " + errArray[i].param + " ,";`
        }
        return res.status(400).send({message: textErr});
      }
      const url = await StripeService.createPaymentSession(parseInt(req.body.price), req.body.user, req.body.ticket_id, req.body.venue, req.body.ticket_section_id);
      return res.status(201).json({ url: url });
    } catch (err) {
      console.log(err);
      res.status(500).json({message: "Unknown error"});
    }
  }
];
```

Ilustración XXIV: payment.controller.js


```

exports.createPaymentSession = async (stripe, price, givenName, familyName, userId, ticketId, venue, ticketSectionId) => {
  try {
    let id = -1;
    for(let i = 0; i < venue.tickets_sections.length && id == -1; i++){
      if(venue.tickets_sections[i].id == ticketSectionId) {
        id = i
      }
    }

    const session = await stripe.checkout.sessions.create({
      payment_method_types: ['card'],
      line_items: [
        {
          price_data: {
            currency: 'eur',
            product_data: {
              name: venue.title + " \n" + venue.tickets_sections[id].title + " \n" + venue.tickets_sections[id].description,
              images: [venue.image],
            },
            unit_amount: price, // en centimos
          },
          quantity: 1,
        },
      ],
      modes: 'payment',
      success_url: config.SUCCESS_URL + "?session_id={CHECKOUT_SESSION_ID}&ticket_id=${ticketId}",
      cancel_url: config.CANCEL_URL,
      metadata: {
        ticket_id: ticketId,
        user_id: userId,
        given_name: givenName,
        family_name: familyName,
        venue_title: venue.title,
        venue_address: venue.address,
        venue_start_date: venue.start_date,
        venue_end_date: venue.end_date,
        ticket_section_id: ticketSectionId
      }
    });
    return session.url;
  } catch (error) {
    throw error;
  }
}

```

Ilustración XXV: Fragmento de código para crear una sesión de pago en Stripe

Por otro lado, ese microservicio también usa el servidor HTTP para recibir eventos desde Stripe, especialmente un evento para cuando se realiza un pago.

Cuando esto sucede, el servicio comprueba que el evento es realmente generado por los servidores de Stripe y si es así, actualiza la entrada en la base de datos para confirmar el pago de esta. Además, genera un evento a través de Amazon SNS para que otros microservicios puedan obtener la notificación si lo desean.

```

exports.webhook =
  async (req, res) => {
    try {
      const payload = req.body;
      const sig = req.headers['stripe-signature'];

      let event;

      event = stripe.webhooks.constructEvent(payload, sig, endpointSecret);

      switch (event.type) {
        case 'checkout.session.completed':
          let sessionWithLineItems = await stripe.checkout.sessions.retrieve(
            event.data.object.id,
            {
              expand: ['line_items'],
            }
          );

          const venue = {
            title: sessionWithLineItems.metadata.venue_title,
            address: sessionWithLineItems.metadata.venue_address,
            start_date: sessionWithLineItems.metadata.venue_start_date,
            end_date: sessionWithLineItems.metadata.venue_end_date
          };

          const user = {
            user_id: sessionWithLineItems.metadata.user_id,
            given_name: sessionWithLineItems.metadata.given_name,
            family_name: sessionWithLineItems.metadata.family_name
          };

          StripeService.fullfillOrder(user, sessionWithLineItems.metadata.ticket_id, venue, payload(sessionWithLineItems.line_items.data[0].price.unit_amount));
          break;
      }
    }
  }

```

Ilustración XXVI: Fragmento de código para completar una sesión de pago

5.3.3 Ticket Generation

Este componente es uno de los más sencillos y es el encargado de generar las entradas de las compras.

Guiado como el resto de los servicios por Test Driven Design, este servicio recibe a través de una cola de Amazon SQS que está conectada al evento *ticket-paid* para detectar cuando se ha producido una compra. Cuando esto sucede, este servicio genera un código QR asociado a la entrada, y acto seguido genera un archivo PDF a partir de una plantilla HTML que contiene información sobre la compra, el evento y agrega la imagen generada.

Acto seguido, sube el pdf a un bucket S3 configurado para almacenar las entradas de los eventos.

```
const QRCode = require('qrcode');

exports.createQr = (ticketId) => {
  return new Promise((resolve, reject) => {
    QRCode.toDataURL(ticketId, function (err, url) {
      resolve(url);
    })
  });
};
```

Ilustración XXVII: Fragmento de código para generar un QR

```
exports.createTicketsPdf = (ticketId, venue, user, price, qrUrl) => {
  const html = fs.readFileSync(__dirname+"/../../templates/ticket.html", "utf-8");

  var options = {
    format: "A4",
    childProcessOptions: {
      env: {
        OPENSSL_CONF: '/dev/null',
      },
    }
  };

  venue.start_date = new Date(venue.start_date).toDateString();
  venue.end_date = new Date(venue.end_date).toDateString();

  var document = {
    html: html,
    data: {
      venue: venue,
      user: user,
      price: price,
      qr: qrUrl
    },
    path: __dirname+"/../../tmp/${ticketId}.pdf",
    type: ""
  };

  return pdf.create(document, options);
}
```

Ilustración XXVIII: Fragmento de código para generar un ticket pdf

```
<!DOCTYPE html>
<html lang="es">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Entrada de Evento</title>
  <style>██</style>
</head>
<body>
  <div class="ticket">
    <div class="ticket-header">
      <h1>Entrada de Evento</h1>
    </div>
    <div class="ticket-body">
      <h2>Nombre del Evento: <span id="eventName">{{venue.title}}</span></h2>
      <p>Nombre: <span id="userName">{{user.name}}</span></p>
      <p>Apellidos: <span id="userSurname">{{user.surname}}</span></p>
      <p>Dirección del Evento: <span id="eventLocation">{{venue.address}}</span></p>
      <p>Hora de Inicio: <span id="startTime">{{venue.start_date}}</span></p>
      <p>Hora de Fin: <span id="endTime">{{venue.end_date}}</span></p>
      <p>Precio Pagado: <span id="price">{{price}}</span></p>
    </div>
    <div class="ticket-footer">
      
    </div>
  </div>
</body>
</html>
```

Ilustración XXIX: Fragmento de la plantilla del pdf

```

exports.uploadTicketPdfToBucket = async (bucketName, filePath, key) => {
  try {
    // Lee el archivo desde el sistema de archivos
    const fileStream = fs.createReadStream(filePath);

    // Configuración de los parámetros para la subida
    const uploadParams = {
      Bucket: bucketName,
      Key: key,
      Body: fileStream,
      ContentType: "application/pdf", // Especifica el tipo de contenido
    };

    // Realiza la subida del archivo
    const upload = new Upload({
      client: s3Client,
      params: uploadParams,
    });

    // Monitorea el progreso de la subida
    upload.on("httpUploadProgress", (progress) => {
      console.log(`Progreso de subida: ${progress.loaded} / ${progress.total}`);
    });

    // Espera a que la subida termine
    const result = await upload.done();
    console.log(`Archivo subido con éxito: ${result.location}`);

    return result.Location;
  } catch (error) {
    console.error("Error subiendo el archivo:", error);
  }
}

```

Ilustración XXX: Fragmento del servicio de subida a S3

Una vez que la entrada se encuentra subida en el bucket S3, el usuario puede acceder a ella desde la webapp.

5.3.4 Ticket Signup

Este componente es el encargado de proporcionar a los métodos para iniciar sesión y registrarse.

Proporciona un pequeño servidor http configurado con Express para llevar a cabo los métodos mencionados anteriormente.

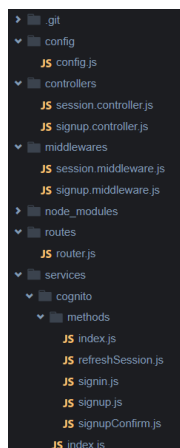


Ilustración XXXI: Jerarquía de carpetas de ticket signup

Los controladores de las rutas son los encargados de usar el SDK de Amazon para Amazon Cognito y realizar las operaciones necesarias para registrar, iniciar sesión o verificar el correo electrónico de un usuario.

Si una sesión es generada, Cognito devolverá tres tokens: Access token, Id token y Refresh token, los cuales son pasados al usuario.

```

const signin = (poolData, body, callback) => {
  const userPool = new CognitoUserPool(poolData);

  const { username, password } = body;

  const authenticationData = {
    Username: username,
    Password: password,
  };

  const authenticationDetails = new AuthenticationDetails(authenticationData);

  const userData = {
    Username: username,
    Pool: userPool,
  };

  const cognitoUser = new CognitoUser(userData);

  cognitoUser.authenticateUser(authenticationDetails, {
    onSuccess: (res) => {
      const data = {
        refreshToken: res.getRefreshToken().getToken(),
        accessToken: res.getAccessToken().getJwtToken(),
        accessTokenExpiresAt: res.getAccessToken().getExpiration(),
        idToken: res.getIdToken().getJwtToken(),
        idTokenExpiresAt: res.getAccessToken().getExpiration(),
      };
      callback(null, data);
    },
    onFailure: (err) => {
      callback(err);
    },
    mfaRequired: () => {
      const data = {
        nextStep: 'MFA_AUTH',
        loginSession: cognitoUser.Session,
      };
      callback(null, data);
    },
    totpRequired: () => {
      const data = {
        nextStep: 'SOFTWARE_TOKEN_MFA',
        loginSession: cognitoUser.Session,
      };
      callback(null, data);
    },
    newPasswordRequired: () => {
      const data = {
        nextStep: 'NEW_PASSWORD_REQUIRED',
        loginSession: cognitoUser.Session,
      };
      callback(null, data);
    }
  });
};

```

Ilustración XXXII: Fragmento de código del método signin.js

5.4 Desarrollo de las webapp

Las webapp son las encargadas de interactuar con los microservicios para ofrecer las funcionalidades requeridas en la plataforma.

5.4.1 Webapp general

El desarrollo de una página fluida y simple es fundamental para garantizar un proceso de compra y descubrimiento de eventos óptimo. Implementado con VueJS, la webapp se desarrolló como una Single Page Application (SPA) donde toda la lógica está contenida en un único archivo html.

El diseño es simple pero intuitivo y amigable. Las diferentes páginas intentan no distraer al usuario y permitirle observar lo que realmente toma relevancia en la página.

La pantalla principal muestra en el centro todos los clientes disponibles en la plataforma.



Ilustración XXXIII: Página principal de la WebApp

Cada uno de estos perfiles pueden ser seleccionados para ver más información sobre los eventos que están programados para las siguientes semanas.

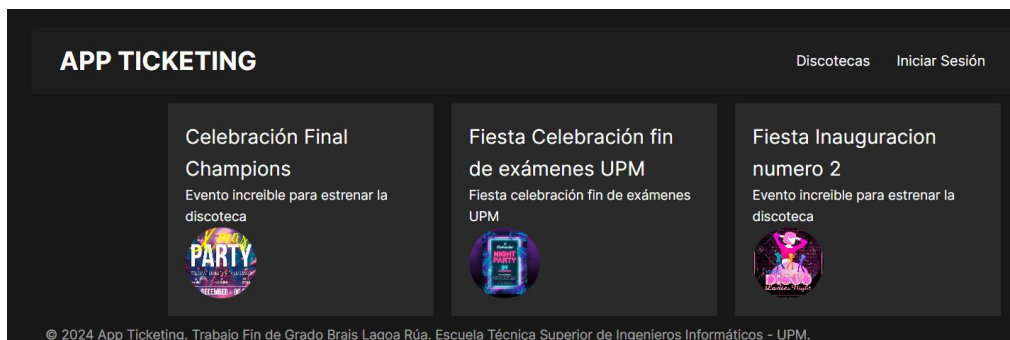


Ilustración XXXIV: Perfil de un cliente en la WebApp

La página para cada uno de los eventos muestra información sobre su lugar, descripción, hora de inicio y fin, así como los diferentes tramos disponibles para comprar entrada.

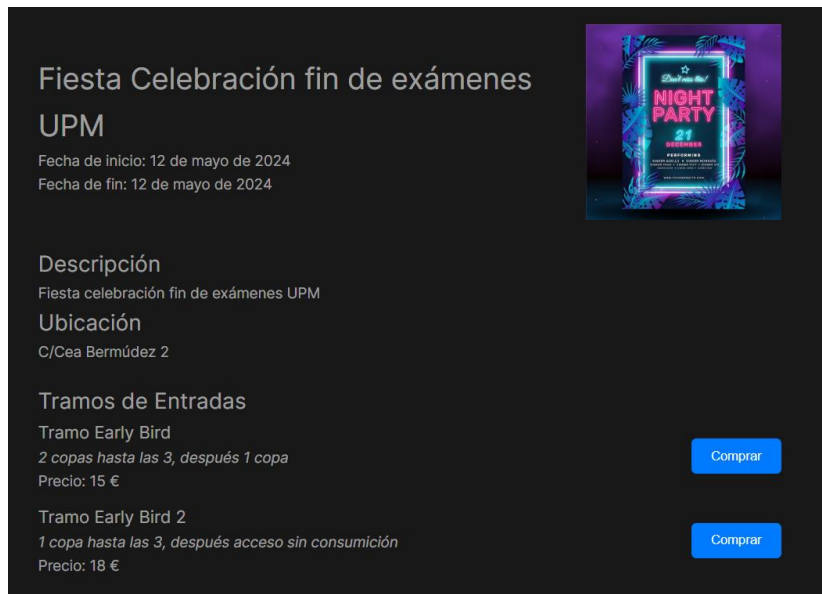


Ilustración XXXV: Página de un evento en la WebApp

Cuando se compra un ticket, se agrega al carrito. Para acceder al carrito hay que iniciar sesión y, en caso de no tener cuenta, es necesario registrarse. Una vez que haya una sesión válida iniciada, se accede al carrito.

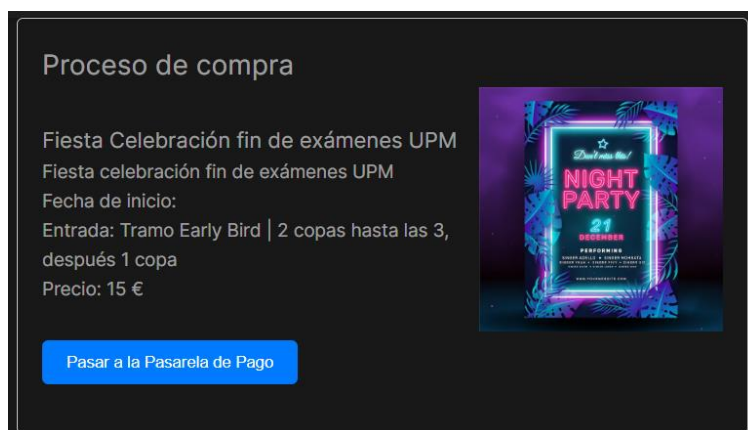


Ilustración XXXVI: Carrito de la WebApp

Tras pasar a la pasarela de pago y finalizar la compra, el ticket se agrega a la cuenta, y el usuario puede acceder al apartado *Mis Tickets* para comprobar y descargar las entradas.

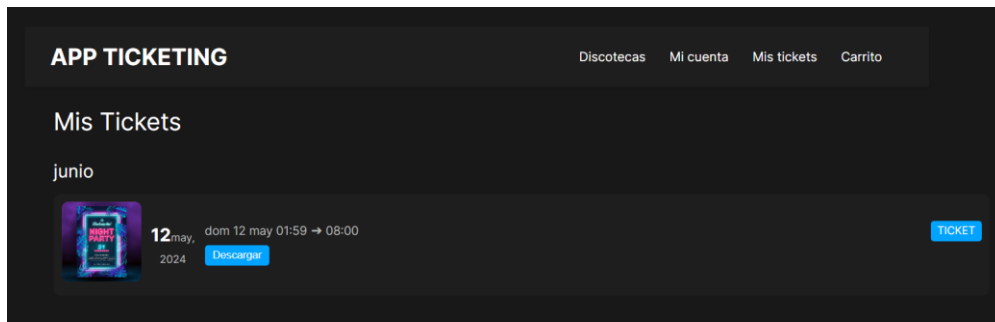


Ilustración XXXVII: Apartado Mis Tickets de la WebApp

Las sesiones y el carrito son persistentes, se guardan en el almacenamiento local del navegador y se recuperan cada vez que se accede a la página. Gracias a esto, los usuarios pueden mantener su carrito y acceder a él sin tener que iniciar sesión cada vez que acceden a la plataforma.

Además, la gestión de la sesión tiene implementado un refresco automático de la sesión mediante el token de refresco. Esto permite prolongar la sesión mucho más tiempo del que se concede inicialmente en el Access Token de Amazon Cognito.

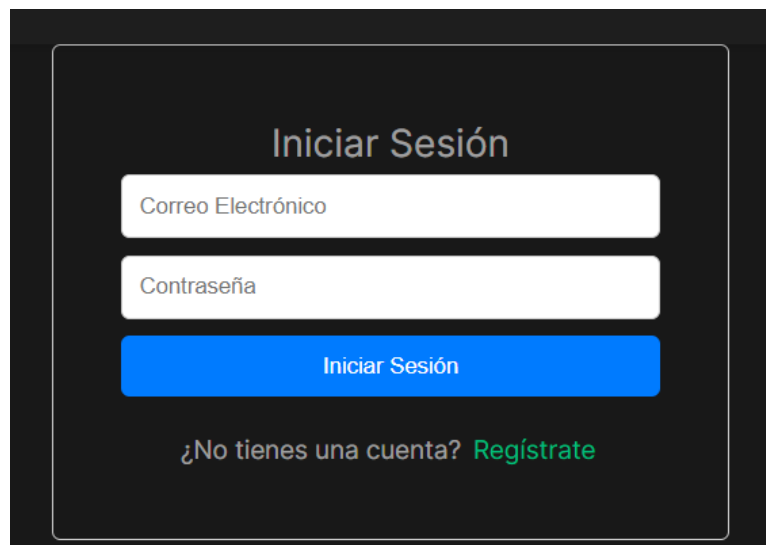


Ilustración XXXVIII: Pantalla de inicio de sesión en la WebApp

A screenshot of a registration form titled "Registro" on a dark background. The form contains several white input fields: "Nombre", "Apellidos", "Correo Electrónico", "Dirección", and an empty field. Below these is a "Género" dropdown menu set to "Masculino" and a "Fecha de nacimiento" field with a date picker icon. Further down are "Contraseña" and "Confirmar Contraseña" fields. A blue "Registrarse" button is at the bottom of the form. Below the button, there is a link that says "¿Ya tienes una cuenta? [Iniciar Sesión](#)".

Ilustración XXXIX: Pantalla de registro en la WebApp

Cada una de estas pantallas realiza operaciones GET, PUT, POST sobre los microservicios anteriormente detallados. La filosofía optada para el diseño de esta webapp fue usar la información que el backend expone, digerirla y presentarla al usuario, todo esto intentando realizar la menor de las peticiones posibles.

5.4.2 Webapp administración

El portal de administración para los clientes también se ha desarrollado usando VueJS. Además, se ha usado otra librería llamada CoreUI que brinda decenas de componentes orientados a la creación de vistas para paneles administrativos.

La estructura es similar a la webapp general: una Single Page Application que permite a los promotores acceder a sus eventos, editarlos o agregar nuevos a su perfil.

Cada una de las páginas realiza operaciones HTTP sobre los microservicios a través del API Gateway. Además, al igual que la anterior webapp, la sesión es persistente mediante el almacenamiento local.

La página de login permite al usuario autenticarse para poder acceder a la aplicación de administración.

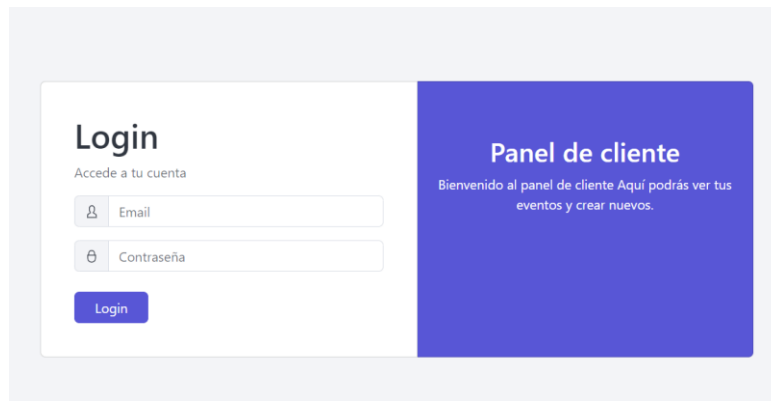


Ilustración XXXX: Pantalla de inicio de sesión

El diseño se enfocó en hacer una interfaz rápida y fluida. La página de inicio ya muestra los eventos que esa cuenta tiene registrados.

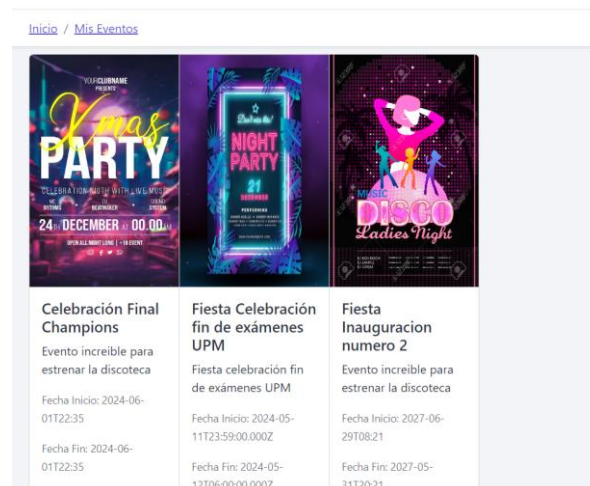


Ilustración XXXXI: Pantalla de eventos disponibles

Una vez accedido a un evento, se muestra información sobre este y también el número de entradas vendidas, junto al ingreso estimado.

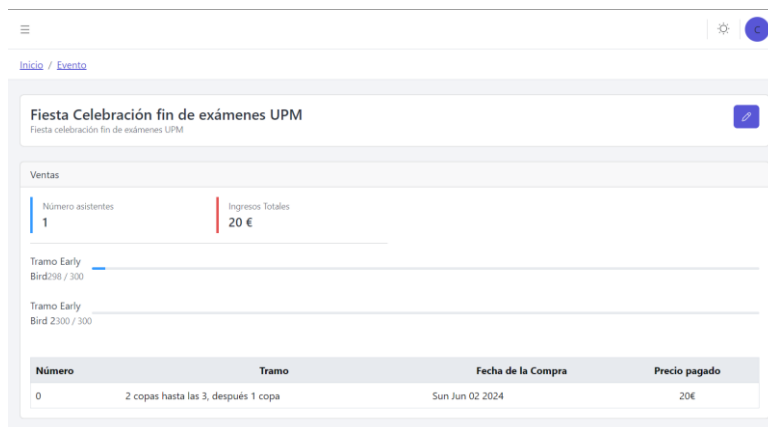


Ilustración XXXXII: Pantalla de información del evento

También es posible editar la información del evento. Eliminar tramos existentes no se permite, ya que haría que las entradas dejarasen de estar asociados a uno, provocando un problema para saber a qué tramo pertenece un ticket. La alternativa que se le ofrece al promotor es deshabilitar el tramo.

Ilustración XXXXIII: Pantalla para editar el evento

Por último, el promotor también puede crear un evento completamente nuevo que se mostrará en su perfil.

Cancelar Crear

Datos generales

Título del evento

Enlace de la imagen del evento

Descripción del evento

Fecha y lugar

Dirección del evento

Hora inicio evento

06/03/2024 03:54 PM

Hora inicio evento

06/03/2024 03:54 PM

Secciones de tickets > Comprar

0 / 1 (1 ítem disponible para la compra)

Ilustración XXXXIV: Pantalla para crear un evento

5.5 Despliegue en AWS

Para conseguir un despliegue escalable y con alta disponibilidad en Amazon Web Services es necesario elegir correctamente los servicios utilizados.

Para desplegar los microservicios se ha optado por Amazon Elastic Beanstalk, que gestiona de forma automática la creación de las instancias EC2, crea un balanceador de carga que distribuya las peticiones entre las instancias generadas y también genera un grupo de autoescalado en función de los parámetros configurados.

Cada microservicio se configuró con instancias t2.micro en un grupo de autoescalado con un número mínimo de instancias de 1 y máximo de 6. El parámetro para generar o eliminar estas instancias es el porcentaje de consumo medio de CPU; estableciendo el margen superior en 60% y el inferior en 20%. Con esto se asegura que antes de que las instancias se saturen, se generen nuevas que ayuden a gestionar la carga de trabajo. Las variables de entorno de Elastic Beanstalk permiten editar sin tener que editar el código el valor de los archivos de configuración de cada uno de los microservicios.

Todos los microservicios se ejecutan como WebServer exceptuando ticket-generation, que trabaja como Worker ya que su trabajo se basa en extraer elementos de una cola.

Environment name	Health	Applica...	Platform	Domain	Running v...	Tier na...
Ticketapi-env	Ok	ticket_api	Docker ru...	ticket-api.eu-west-3.elasticbea...	code-pipeline...	WebServer
Ticketgeneration-env	Ok	ticket_gen...	Docker ru...	-	code-pipeline...	Worker
Ticketpayment	Ok	ticket_pay...	Docker ru...	ticket-payment.eu-west-3.elas...	code-pipeline...	WebServer
Ticketsignupservice-env	Ok	ticket_sig...	Docker ru...	Ticketsignupservice-env.eba-m...	code-pipeline...	WebServer

Ilustración XXXXV: Configuración entornos Elastic Beanstalk

Cada una de estas instancias corre una imagen Docker que se encuentra alojada en Amazon Elastic Container Registry las cuales se generan automáticamente a través del CI/CD pipeline que se mencionará adelante. Para reducir los costes de mantener las imágenes, se ha limitado a un máximo de 5 por registro a través de reglas de ciclo de vida.

ticket-api	590183875813.dkr.ecr.eu-west-3.amazonaws.com/ticket-api	June 02, 2024, 18:27:38 (UTC+02)	Disabled	Manual	AES-256
ticket-generation	590183875813.dkr.ecr.eu-west-3.amazonaws.com/ticket-generation	June 02, 2024, 20:31:22 (UTC+02)	Disabled	Manual	AES-256
ticket-payment	590183875813.dkr.ecr.eu-west-3.amazonaws.com/ticket-payment	June 02, 2024, 13:52:47 (UTC+02)	Disabled	Manual	AES-256
ticket-signup-service	590183875813.dkr.ecr.eu-west-3.amazonaws.com/ticket-signup-service	June 02, 2024, 20:30:53 (UTC+02)	Disabled	Manual	AES-256

Ilustración XXXXVI: Registros Elastic Container Registry

Como ya se mencionó en el apartado 5.3.1, para el uso de un servidor caché se optó por desplegar ElastiCache for MemCached en un cluster con dos nodos t3.micro que gestionará las peticiones del microservicio *ticket-api*. Cada nodo está en una zona de disponibilidad distinta para conseguir alta disponibilidad en caso de que una falle. Para dar una capa de seguridad extra, el grupo de seguridad de estos nodos sólo permite conexiones al puerto 11211 (Memcached) a aquellas instancias EC2 que formen parte de los microservicios.

Node name	Status	Created date	Endpoint
0001	Available	April 30, 2024, 17:01:07 (UTC+02:00)	memcachedticketsapi.3yyxvg.0001.euw3.cache.amazonaws.cc
0002	Available	April 30, 2024, 17:01:07 (UTC+02:00)	memcachedticketsapi.3yyxvg.0002.euw3.cache.amazonaws.cc

Ilustración XXXXVII: Nodos Cluster Memcached

Amazon API Gateway es el servicio que se interpone entre los microservicios que ofrecen servicios a internet como APIs y los clientes. Actúa como punto de entrada común y se encarga de distribuir en función del destinatario las peticiones al servicio correcto. Las peticiones del exterior sólo pueden ser

dirigidas a dos servicios: *ticket-signup* o *ticket-api* por lo que para realizar la distribución de forma correcta, se ha desplegado un HTTP API Gateway con dos reglas de integración:

1. Todas las peticiones que empiecen con `/auth/{proxy+}` serán enviadas al servicio *ticket-signup*
2. Aquellas peticiones que no cumplan con el punto 1, serán enviadas al servicio *ticket-api* (\$default)

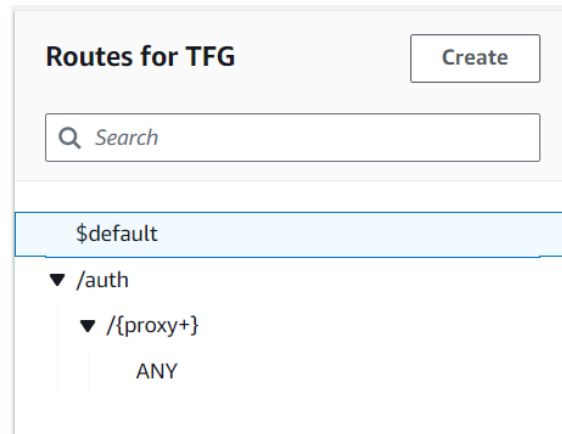


Ilustración XXXXVIII: Rutas del HTTP API Gateway

Para la comunicación asíncrona se crearon los temas en Amazon Simple Notification Service, un modelo PubSub. El tema *ticket-paid* tiene como suscriptor una cola creada en Amazon Simple Queue Service y que recibe los eventos, para luego agregarlos a su cola y que sean consumidos por otros servicios.

Para el registro e inicio de sesión de los usuarios se optó por utilizar Amazon Cognito. Generando una user pool se ha podido establecer los campos deseados a la hora de registrarse un usuario, delegando toda la lógica de almacenamiento y gestión de tokens a este servicio.

Los archivos estáticos necesarios por la aplicación: archivos de entradas, webapp general y webapp de administración están almacenados en un S3 bucket, el cual para evitar ataques que aumente drásticamente el precio de el servicio, se encuentra con acceso bloqueado para cualquier petición excepto de Amazon CloudFront. Este servicio se usa como Content Delivery Network (CDN) que distribuye los contenidos del bucket S3 en puntos estratégicos para obtener menos latencia y aplicar caching a los contenidos.

Para el despliegue se crearon tres CDN, uno para las entradas, y otros dos para cada webapp. Estas dos últimas requieren de dos reglas en el apartado de errores de página para que la SPA funcione correctamente, haciendo que las respuestas 404 se transformen en 200 hacia el archivo `index.html`

Esto sucede porque las SPA actualizan la URL de búsqueda que se gestiona de forma interna por la aplicación. Sin embargo, también hace que se busque dicha

ruta en el servidor, donde no existe y devuelve 404. Por lo que se transforma en 200 y se reenvía al index.html, donde procesará correctamente la petición.

Tanto la CDN como la API Gateway tienen delante Amazon WAF, un pequeño firewall que bloquea peticiones maliciosas y evita que lleguen a los servicios finales.

Para conseguir Integración y Despliegue Continuo (CI/CD), se ha usado Amazon CodePipeline y Amazon CodeBuild. Estas dos herramientas generan flujos de integración y despliegue cuando se produce un commit en el repositorio de los microservicios o las webapps, que están alojados en AWS CodeCommit. Cuando se hace commit a la rama principal, se activan las canalizaciones que llevan a cabo diferentes acciones:

- Para los microservicios, se genera una etapa de CodeBuild que se encarga de compilar la nueva imagen Docker, subirla al repositorio y generar un archivo Dockerrun.aws.json que se le pasará a Elastic Beanstalk para que actualice el entorno correspondiente.
- Para las webapps, se genera otra etapa diferente de CodeBuild que genera los archivos estáticos de la página. Una vez se ha generado la distribución, se suben los archivos al bucket S3 correspondiente para que puedan ser accesibles desde la CDN.

Cada uno de los repositorios contiene un archivo buildspec.yml que define la rutina de compilación que debe usar CodeBuild.

5.6 Estimación de costes

Amazon Web Services es un servicio *pay as you go*, por lo que se paga sólo por aquellos recursos de los que se ha hecho uso. Aún así, en cualquier planificación se vuelve esencial conocer el coste de mantener cierta infraestructura, por lo que es necesario hacer una estimación de cuanto costaría mantener de forma mensual todos los servicios usados en AWS.

La elección de los servicios adecuados también influye en el precio, pues elegir una solución u otra puede tener un impacto significativo en el precio. Un ejemplo es el acceso a una web alojada en S3 comparado con acceder a través de una CDN. Para transferir 1TB de datos y procesar 1 millón de solicitudes GET en un mes, Cloudfront cobraría 88€, mientras que en S3 ascendería hasta 93€.

Para estimar los costes debemos aproximar algunos datos teniendo en cuenta que al ser un proyecto de fin de grado, el tráfico será reducido.

- **Elastic Container Registry:** 4 repositorios con 5 imágenes cada uno. De media cada imagen pesa 430MB, haciendo un total de 8,2GB. Se estima que cada mes, cada registro generará 5 imágenes como consecuencia de updates.

- **EC2:** 4 servicios con 1 instancias de media cada uno hacen 4 instancias EC2 t2.micro en total
- **Application Load Balancer:** 4 balanceadores de carga, uno por cada entorno de Elastic Beanstalk. Como la aplicación no tiene mucho tráfico, estimamos que como mucho experimentará 1 conexión por segundo cada uno.
- **DynamoDB:** 1KB cada elemento de media, siendo un 95% de las peticiones no transaccionales.
- **Cognito:** 10 usuarios activos mensualmente.
- **HTTP API Gateway:** 2000 peticiones mensuales con un tamaño medio de cada una de 1KB.
- **Amazon SNS:** Menos de 1 millón de peticiones al mes y menos de 1 millón de peticiones a SQS por mes
- **Amazon SQS:** Menos de 1 millón por mes
- **S3:** 1GB de espacio cada mes
- **Cloudfront:** Menos de un TB de entrada y salida al mes
- **AWF:** No hace falta estimaciones
- **CodeBuild:** 130 builds cada mes de 5 minutos de media en instancias arm1.small
- **Pipeline:** 6 canalizaciones activas al mes.

Usando estas estimaciones sobre la calculadora de precios de AWS[17], se estima que cada mes mantener la infraestructura costaría sobre 150€.

6 Resultados y conclusiones

Los objetivos planteados al inicio del trabajo se han alcanzado: diseñar e implementar una plataforma web que permita a usuarios comprar entradas a eventos de forma online. Todo desplegado en Amazon Web Services.

Diseñar una plataforma de este calibre implica muchos cambios durante el proceso, toma de decisiones fundamentales que pueden tener repercusión en la forma de desarrollar todo el proyecto. Todas estas decisiones se tomaron siempre analizando diversos factores, investigando sobre aquello que no se tenía conocimiento y, sobre todo, pensando a visión de futuro, para permitir que este proyecto pueda seguir escalando con el tiempo.

A pesar de todo, la arquitectura de microservicios fue diseñada e implementada con éxito, desacoplando la lógica en diferentes componentes independientes. Además, se realizaron las dos Single Page Application que se plantearon en los objetivos iniciales de forma completamente funcional.

El despliegue fue una tarea compleja, pues son muchos los servicios presentes que interactúan entre sí para poder conseguir la infraestructura deseada, pero no deja de ser apasionante. Un ingeniero debe ser capaz de adaptarse a nuevos entornos para alcanzar los objetivos propuestos.

Una vez llegado a este punto, se puede decir que este TFG ha sentado las bases para tener un prototipo funcional de lo que pueda ser como se planteaba al inicio de este proyecto, un próximo servicio software que permita a promotores de todas partes del mundo vender entradas de una forma cómoda y sencilla; y a su vez a usuarios tener sus eventos favoritos más cerca.

Los siguientes pasos para alcanzar este nuevo objetivo pueden ser varios. Refinar las interfaces web para hacerlas más atractivas siguiendo los estándares más novedosos en UI/UX o implementar mejores métricas en el apartado de los promotores, para que puedan tener un mejor conocimiento sobre su público. Por otro lado, también se podrían implementar mejoras del lado de la pasarela de pago, diseñando e implementando una página propia que no obligue al usuario a dejar la aplicación hasta finalmente desarrollar un procesador de pagos propietario.

7 Análisis de Impacto

La Agenda 2030 para el Desarrollo Sostenible es un plan adoptado por la Asamblea General de las ONU que establece 17 Objetivos con 169 metas[18] que abarcan la esfera económica, social y ambiental con el objetivo de reducir la pobreza, desigualdad, así como combatir contra el cambio climático y el deterioro ambiental. Establecen que para alcanzar estos objetivos y metas, todo el mundo debe contribuir, desde gobiernos, empresas privadas pasando por las personas individuales.

En la introducción de esta memoria se hacía referencia a los beneficios de reducir grandes colas para comprar una entrada física por plataformas web donde con un par de clicks poder comprar una entrada a un evento. La reducción de entradas físicas contribuye enormemente a los siguientes objetivos:

- **Objetivo número 12, “Producción y Consumo Responsables”:** La reducción del uso de papel derivado de la transición hacia plataformas online, permita a su vez reducir los residuos derivados, contribuyendo a un consumo responsable.
- **Objetivo número 15, “Vida de Ecosistemas Terrestres”:** La deforestación es un gran problema en la actualidad que afecta de forma severa a la biodiversidad. Este objetivo pretende conservar la vida de los ecosistemas terrestres. Con la reducción del consumo de papel derivado del uso de plataformas online, se contribuye a la conservación de bosques y de la biodiversidad terrestre.

Bibliografía

- [1] Amazon Web Services, «¿Qué es el software como servicio?». 2024. [En Línea]. Available: <https://aws.amazon.com/es/what-is/saas/>
- [2] Amazon Web Services, «Amazon Web Services». 2024. [En Línea]. Available: <https://aws.amazon.com/>
- [3] Red Hat, «What is infrastructure as code?». 2022. [En Línea]. Available: <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>
- [4] Developer Associate
- [5] Fourvenues, «Fourvenues». 2024. [En Línea]. Available: <https://www.fourvenues.com/en>
- [6] AWS, «Precio de los productos de SaaS». 2024. [En Línea]. Available: https://docs.aws.amazon.com/es_es/marketplace/latest/userguide/saas-pricing-models.html
- [7] Chris Richardson, «What are microservices?». 2024. [En Línea]. Available: <https://microservices.io/>
- [8] NodeJS, «NodeJS». 2024. [En Línea]. Available: <https://nodejs.org>
- [9] Swagger.io, «Swagger.io». 2024. [En Línea]. Available: <https://swagger.io/specification/>
- [10] AWS, «¿Qué es una API RESTful?». 2024. [En Línea]. Available: <https://aws.amazon.com/es/what-is/restful-api/>
- [11] VueJS, «VueJS». 2024. [En Línea]. Available: <https://vuejs.org/>
- [12] Trello, «Trello». 2024. [En Línea]. Available: <https://trello.com>
- [13] Git, «Git». 2024. [En Línea]. Available: <https://www.git-scm.com/>
- [14] Stripe, «Stripe Documentation». 2024. [En Línea]. Available: <https://docs.stripe.com/>
- [15] Martin Fowler, «Test Driven Development». 2023. [En Línea]. Available: <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
- [16] W3C, «Use el formato de fecha internacional ISO 8601». 2003. [En Línea]. Available: <https://www.w3.org/QA/Tips/iso-date.html.es>
- [17] AWS, «AWS pricing calculator». 2024. [En Línea]. Available: <https://calculator.aws/#/>
- [18] Naciones Unidas, «Objetivos de Desarrollo Sostenible». 2024. [En Línea]. Available: <https://www.un.org/sustainabledevelopment/es/objetivos-de-desarrollo-sostenible/>

Este documento esta firmado por



Firmante	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
Fecha/Hora	Mon Jun 03 23:54:40 CEST 2024
Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
Numero de Serie	561
Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)