Documento de Arquitectura

Link al repositorio

Introducción

El Documento de Arquitectura que se presenta a continuación es el resultado de un proceso de análisis, diseño y planificación meticulosos y centrados en la creación de una arquitectura de software robusta, flexible y eficiente. Este documento se centra en justificar y documentar las decisiones arquitectónicas que se han diseñado y llevado a cabo durante la etapa inicial de desarrollo del proyecto.
Incorporamos al cierre del documento una sección denominada "Conclusiones y Lecciones Aprendidas", con el objetivo de resaltar que este trabajo se realizó durante una etapa educativa del curso. Este segmento permitirá reflejar nuestra progresión y los conocimientos adquiridos a lo largo del proceso.

Vista General del Sistema

En el desarrollo del sistema, adoptamos una arquitectura de microservicios. Este enfoque facilitó la extracción de funcionalidades reutilizables a componentes autónomos, contribuyendo a la creación de un sistema más mantenible y adaptable. Así, se logró una desacoplamiento de funciones, distribuyéndolas en diferentes servicios, lo que optimizó la modularidad y la independencia de cada componente.
En cuanto al manejo de los datos, optamos por mantener una base de datos unificada para la lógica de negocio. Esta decisión simplificó considerablemente el proceso de desarrollo del sistema. Al centralizar la información, pudimos garantizar un acceso coherente y eficiente a los datos, favoreciendo la consistencia y la integridad de la información.

Requerimientos

En esta sección pasaremos a resumir los requerimientos solicitados, en caso de necesitarlos de forma detallada, se deja en la carpeta acceso a la letra de requerimientos del obligatorio. Finalmente a lo largo del documento explicaremos cómo hicimos para poder cumplir con estos requerimientos planteados.
Los requerimientos son:

- REQ 1 – Alta de Proveedores
- REQ 2 – Autorizar/Desautorizar un evento
- REQ 3 – Consultar bitácora de la plataforma
- REQ 4 – Crear Eventos
- REQ 5 – Editar
- REQ 7 – Consultar Eventos
- REQ 8 – Comprar un evento
- REQ 9 – Acceder a un evento
- REQ 10 – Consulta de todos los Eventos para Proveedores
- REQ 11 – Consulta de un Evento para Proveedores

- REQ 12 – Consulta de Eventos Activos para Administradores
- REQ 13 – Consulta de todos los Eventos para Administradores
- REQ 14 – Consulta de todos los Eventos para Usuarios
- REQ 15 - Gestión de errores y fallas
- REQ 16 – Protección de datos y accesos a la plataforma
- REQ 17 - Manejo de carga
- REQ 18 - Información de auditoría

Restricciones:

- La implementación del Backend debe desarrollarse en NodeJS utilizando las tecnologías vistas en el curso, o en la tecnología que el equipo docente haya autorizado en casos particulares por motivos académicos. Es opcional y abierta la elección de packages que puedan ayudar al desarrollo, teniendo que justificar la elección en la documentación.
- Las consultas deben poder realizarse en un tiempo de menos de 5 segundos, demostrando que, para un número elevado de usuarios concurrentes, los tiempos se mantienen relativamente constantes en el tiempo.
- Los usuarios deben autenticarse en la plataforma para poder utilizar cualquiera de los servicios que provee. La autenticación se debe realizar a través de servicios de terceros.

Vistas

1. Vista de módulos - VM01
2. Vista de asignación - VA01
3. Vista de componentes y conectores - VCC01

Análisis y Justificación de Arquitectura

A continuación analizaremos algunas de las decisiones de diseño que nos resultan destacables y cómo impactan en nuestra aplicación en materia de los atributos de calidad en juego.
Optamos por que el servicio externo de autenticación sólo nos brinde dentro del Payload el email del usuario, de esta forma tenemos que implementar toda la lógica de roles dentro de la aplicación. Haciéndolo de esta forma no nos acoplamos a un servicio externo, pero conlleva tener que mantener el sistema de roles.
Decidimos utilizar un servicio externo para obtener datos de país, ciudad y moneda, lo cual nos evita la necesidad de mantener y actualizar esta información en nuestra propia base de datos. Esto mejora la mantenibilidad del sistema, al reducir la cantidad de datos que necesitamos gestionar, pero también puede introducir ciertos riesgos en cuanto a la disponibilidad y la confiabilidad, ya que estamos dependiendo de servicios externos para estas funciones críticas.
La elección de mantener una base de datos separada para los servicios de autenticación y la lógica de negocio mejora la escalabilidad del sistema, ya que nos permite mover el servicio de autenticación a otras aplicaciones fácilmente. Sin embargo, también conlleva un coste en términos de mantenibilidad, ya que implica manejar dos bases de datos.
La decisión de manejar el servicio de archivos por separado mejora el rendimiento, especialmente durante los períodos de alta carga de archivos, al no sobrecargar el resto de la aplicación. Sin embargo, esto introduce cierta redundancia en el código y crea múltiples puntos de entrada a la aplicación, lo que puede afectar a la seguridad y la mantenibilidad.

Finalmente, el uso de Redis como caché para las consultas solicitadas mejora considerablemente la performance del sistema al reducir los tiempos de respuesta para recuperar datos. Sin embargo, esta implementación puede ser más compleja, lo que afecta la mantenibilidad. Además, al tener que mantener una copia en caché de los datos, se puede introducir cierta complejidad adicional en términos de consistencia de los datos.

Mapeo Entre Arquitectura y Requerimientos

A continuación realizaremos un mapeo de los principales casos de uso o restricciones y las soluciones arquitectónicas planteadas.

Caso de Uso	Solución de Arquitectura y Notas
Autorizar/Desautorizar un evento	Nos pedían varios chequeos por parte del administrador al momento de realizar la autorización y desautorización de un evento, además de que en el futuro sean sencillos de modificar o agregar nuevos, por lo que lo hicimos teniendo en cuenta una implementación de Pipes & Filters, cada chequeo se sumaba o no al Pipe según correspondiera, en caso de existir nuevos en el futuro basta con sólo agregarlo a la lista de filtros. También se pedía que informará a los proveedores, por lo que lo establecido durante la autorización se envía a una cola de mensajes, donde un consumidor lo obtiene y es quien envía el correo electrónico. En este caso hicimos la configuración para el envío por medio de GMail, pero al situar un adaptador que funciona de intermediario, dicho método se puede reemplazar sin necesidad de tocar la lógica del servicio. Finalmente se nos pedía que se informe si un evento no había sido autorizado y faltaban 168 hs para su inicio, al no poseer MySQL eventos programados que envíen correos o disparen una llamada a un servicio externo, lo realizamos a través de un servicio CRON de un servidor, lo dejamos documentado en el README del repositorio. Dicho artefacto corre todos los días y dispara una llamada a nuestro servicio de eventos, el cual tiene un endpoint dedicado a dicha función. También es posible acceder desde el Gateway autenticado como administrador. La lista de administradores se consigue de otro servicio y a todos ellos les envía, en caso de querer modificarlo se puede hacer a través del Gateway que tiene todo el ABM de usuarios expuesto para administradores.
Consultar bitácora de la plataforma	Para implementar esta funcionalidad, implementamos el patrón productor-consumidor, siendo el productor el servicio de eventos, que envía a una cola de mensajes las acciones correspondientes, y por el otro lado un servicio de bitácora consume de dicha cola y las agrega a una base de datos MongoDB. Decidimos utilizar MongoDB ya que en caso de querer modificar estas tablas que no son de negocio, es más sencillo en una base de datos no relacional.
Crear Eventos	La creación de eventos decidimos dividirla en dos, una con los archivos y otra para los otros campos. Esto lo decidimos por el

	hecho de separar la lógica de manejo de archivos del resto de la aplicación. Debido a que la lógica de archivos conlleva un mayor flujo de datos, lo extraemos a otro servicio para que el resto del sistema no se sobrecargue en el momento de streaming de los eventos, además nos brinda la capacidad de poder escalar horizontalmente ese servicio en particular. Dentro de la implementación del servicio de archivos, tuvimos consideración de encapsular el método de guardar u obtener un archivo, a efecto de poder implementarlo de manera distinta en el futuro sin modificar la lógica de la aplicación.
Consulta de todos los Eventos enriquecida con datos en menos de 5 segundos	La solución de este requerimiento conlleva la interacción de varios servicios, el de eventos, el de logs y en ocasiones el de bitácora. Los tiempos de demora de las ejecuciones de tomaron de los logs del Gateway, el mismo tiene un middleware que chequea entre otros datos la duración de la llamada. Al implementarlo contemplamos que a medida que los logs aumentaban, también lo hacía el tiempo de respuesta. Para mantener el tiempo constante implementamos el uso de Redis para mantener las estadísticas en memoria e irlas actualizando a medida que se reciben los logs. En caso de pérdida de datos de Redis, se implementa que al inicio del servicio de logs, cargue las estadísticas necesarias y luego continúe trabajando desde allí, de esta forma, con es sistema cargado, obtuvimos un período de tiempo de respuesta constante que se mantiene en el órden de los milisegundos.
Gestión de errores y fallas	Se implementó un servicio de logueo en el sistema general. Cada servicio loguea a una cola de mensajes, y a medida que se van obteniendo un consumidor los obtiene de la cola de mensajes y lo almacena en MongoDB. Además, cada servicio loguea todo lo sucedido de manera local en una dirección configurable desde el archivo de configuración.
Protección de datos y accesos a la plataforma	Se implementó el acceso a la aplicación a través de un servicio de Gateway, él mismo conoce a todos los otros microservicios y rutea la llamada según corresponda. Además, verifica la autenticación y realiza acciones de logueo, seguridad y sanitiza las peticiones. Debido a que las rutas pueden cambiar o agregar nuevas, estas se cargan de forma dinámica desde archivos JSON, que poseen la información necesaria de ruteo y autenticación, para que al momento de reinicio del servicio sean cargados y se modifiquen sin necesidad de modificar el código principal. El otro acceso a la aplicación es el servicio de Archivos, el cual tiene repetida la lógica de autenticación y seguridad que implementa el Gateway.
Manejo de carga	Se hicieron pruebas de carga, tanto de las consultas como de los videos y se verificó que los servicios más afectados eran el de eventos y el de archivos, lo cual es lógico, ya que uno tiene toda la lógica de la aplicación y el otro el manejo de cargas pesadas. Al realizar el despliegue con PM2 pudimos distribuir de mejor forma la carga y llegamos a que si con 25 hilos, durante 30 segundos, se realizan consultas, es conveniente subir a 2 las instancias de evento y archivo, acción que se debe tomar en cuenta en

	momentos de carga del sistema.
Información de auditoría	Al igual que la información de bitácora, implementamos el patrón productor-consumidor para llevar la auditoría de eventos del sistema, en los gateway y sistemas verificamos que si se realizaban acciones de acceso a datos no autorizadas, estas se envían a una cola de mensajes que es procesada por el servicio de auditoría.
La autenticación se debe realizar a través de servicios de terceros	Decidimos utilizar el patrón de Identidad Federada para realizar la acción de autenticación. Los usuarios se autentican contra una API de terceros que genera un Token, que en su Payload tiene el email de la persona, y lo firma con una clave privada. Luego la persona presenta dicho token a nuestro servicio de autenticación y éste, que tiene una clave pública del servicio anterior, verifica que el token es válido y puede recuperar el mail de la persona. Además, al momento de codificarlo, tuvimos en cuenta realizar Defer Binding, para contemplar la posible implementación de nuevos métodos de manera polimórfica.

Conclusiones y Lecciones Aprendidas

A lo largo del desarrollo de este proyecto, fuimos testigos de la relevancia de contar con una estructura sólida y bien fundamentada desde las etapas iniciales antes de iniciar la programación. Al seguir este enfoque, pudimos anticipar situaciones que debíamos considerar desde el comienzo, tales como la implementación de un sistema de logueo con tiempo de solicitudes y el empleo de colas para algunos servicios que podían ser asíncronos. Aunque fue necesario realizar ciertos ajustes durante la fase de integración, la planificación y estructuración previa aseguró el cumplimiento en su mayoría de los requerimientos planteados.

Con el progreso del proyecto, identificamos que algunos servicios, como el svc_admin y svc_evento, compartían lógica de negocio y base de datos, por lo que su unificación hubiera simplificado el proceso. Reconocimos que el servicio de eventos resultó ser más extenso de lo ideal, y si hubiésemos dispuesto de más tiempo, lo hubiésemos optimizado. Observamos también que la forma en que se implementó el logueo podía generar confusiones al seguir el flujo del código, debido a que extendía las funciones ocupando múltiples líneas, complicando su lectura. Este aspecto podría corregirse en una futura instancia de mejora.

Por último, descubrimos una alternativa para la implementación del Gateway, que a pesar de tener una estructura similar, funciona como proxy en lugar de ser quien efectúa las llamadas y las redirige hacia los servicios. Esta alternativa promete un menor consumo de memoria y mayor eficiencia. Sin embargo, dado que el Gateway actual no presentó problemas en instancias de alta carga, decidimos no invertir esfuerzo en modificarlo por ahora.