Arquitectura near real time para el análisis de negocio

RODRIGO DE MIGUEL GONZÁLEZ

MASTER EN ARQUITECTURA BIG DATA
KSCHOOL

TRABAJO FIN DE MASTER

Curso 2018-2019

Director: Rubén Casado Tejedor

1. Introducción

Desde hace año y medio trabajo junto con un amigo (y socio) en el desarrollo de una plataforma web para la adopción y gestión de animales procedentes que las protectoras tiene a su cargo a la espera de ser adoptados por sus nuevas familias.

La idea de este TFM es el desarrollo de una arquitectura que pueda proporcionar información útil de forma cercana al tiempo real sobre el uso que se hace de la plataforma tanto de los cliente de la misma (registro de animales nuevos) como de la demanda de los usuarios (búsquedas de los animales) y poder hacer el análisis y toma de decisiones sobre la misma y estrategias de mercado y marketing.

La plataforma web permite que las protectoras aumenten tanto su visibilidad como la de los animales en internet registrándolos en ella para facilitar a sus posibles adoptantes encontrar el compañero ideal que están buscando, con el objetivo de fomentar la adopción frente a la compra.

Por otra parte, de cara a los usuarios, la plataforma expone un potente buscador de los animales registrados, de forma que poder encontrar la mascota ideal sea muy sencillo.

La plataforma está implementada con React+Redux en el front, NodeJS en el Backend , MongoDB como sistema de persistencia principal y AWS S3 como repositorio de objetos.

La arquitectura montada para este TFM permite que el *backend* de la plataforma además de realizar la persistencia en MongoDB pueda enviar los registros de nuevos animales que se realizan por parte de las protectoras y de las búsquedas que se están realizando a un pequeño cluster de analitica hecho con Elasticsearch y Kibana desplegado en Amazon Web Services.

Además permite que los usuarios puedan suscribirse a los filtros de las búsquedas que quieran para poder recibir vía email avisos inmediatos de los animales que entren nuevos mediante el uso de AWS Lambda, AWS Kinesis (junto con Kinesis Firehose) y el módulo de envío de emails implementado en el en el backend.

2. Casos de uso alto nivel

El entorno de analítica lo conforman dos casos de uso:

- Dashboard en Kibana del registro de animales nuevos
- Dashboard en Kibana de las búsquedas realizadas

Esta parte consiste en un dashboard que muestra la información en tiempo real los animales nuevos y búsquedas que se estén realizando en la plataforma. Haciendo especial hincapié en la especie de los mismos (perros o gatos), y sobretodo en la raza, característica que se considera principal a la hora de elegir un animal de compañía.

Se muestra un conteo de registros o búsquedas, la línea de tiempo del conteo para poder ver la evolución histórica, un top 5 de razas tanto en gatos como perros, y lo que a mi parecer aporta mayor valor de negocio, un mapa de puntos calientes en lo que poder visualizar desde donde hay mayor actividad, desde donde realizan las búsquedas los usuarios y poderlo contrastar con dónde tiene mayor presencia la plataforma y poder realizar campañas de marketing dirigido a esas zonas.

Por otro lado se ha implementado la posibilidad de realizar suscripciones a filtros de búsquedas de forma que al registrarse un nuevo animal se chequean las suscripciones activas que cumplan las características de dicho animal y se procede al envío de un email a los usuarios avisando de la nueva entrada con la información del animal y de contacto con la protectora.

3. Arquitectura técnica

La arquitectura realizada, aunque a mi parecer es sencilla, no conteniendo demasiados elementos, pero está diseñada con una finalidad muy clara, por una parte, la obtención de valor para negocio de lo que ocurre al otro lado de la plataforma, en el lado de clientes/usuarios, y por otro, una cota superior en coste y recursos, ajustarla en coste dado que la intención tras este máster es seguir usandola, y, a día de hoy, será pagada con fondos personales.

Como introducción y posicionar la entrada a la arquitectura del TFM, el backend de la aplicación está realizado en **NodeJS**, usando **ExpressJS** como framework web y de procesamiento de la aplicación operacional. La aplicación, a día de hoy, está desplegada en **Heroku** (soportado por Salesforce), es un **PaaS**, una plataforma como servicio para el despliegue de aplicaciones web.

En dicha plataforma, conectada con el repositorio de código **GitHub**, se tiene desplegado los tres entornos más típicos, el entorno de desarrollo se utiliza la máquina local para la aplicación aunque los distintos servicios utilizados si se encuentran desplegados (MongoDB, SendGrid como servicio de emails y AWS S3), el entorno para preproducción con todos los servicios activos y un entorno de producción completo aunque actualmente no productivo.

La base de datos principal es **MongoDB**, alojada en mLab, un **DBaaS**, una base de datos como servicio. El entorno de trabajo y preproducción por ser más que suficiente para el desarrollo actual usa la versión gratuita de entorno, que consta de un cluster de un único nodo compartido con 0,5GB de almacenamiento y RAM variable. El entorno de producción que se empezará a utilizar será el básico, aunque aún no activo por no ser usado, consta de un cluster de tres nodos, un primario, un secundario y uno de arbitraje, parte desde 1GB de almacenamiento ampliable hasta 8, RAM variable, aunque en un estado de poco uso ronda los 150MB. Además tiene un backup diario a **S3** de la BD con la que poder hacer los respaldos de datos en caso de perdidas.

Por otra parte todo el alojamiento de imágenes se realiza en AWS S3 desde el backend para su uso en el *frontend.*

El backend, una vez realizada la búsqueda o la inserción del nuevo animal en la BBDD envía el registro a los **Streams** de **AWS Kinesis**, desde ahí **Kinesis Firehose** los envía al cluster de **ElasticSearch** para su explotación.

Los streams están configurados con un único shard por el bajo volumen de información, eso sí, su reescalado es trivial pudiéndose automatizar el proceso.

Cada shard de **Kinesis** permite la escritura de 1MB/s o 1000 registros/segundo, y la lectura de 2MB/s, siendo el escalado lineal. La persistencia de los registros es de 24 horas por defecto, aunque pudiéndose ampliar hasta una semana.

Al igual que el **Kinesis**, el cluster de **Elasticsearch** está configurado actualmente con el mínimo de máquinas, una instancia de datos con una t2.small.elasticsearch, una instancia EC2 t2.small customizada para **ELK**. Obviamente esta configuración de cluster no está nada recomendada para un entorno productivo, para ello AWS probé la posibilidad de tener hasta tres zonas de disponibilidad de las instancias de datos, el reescalado del número de instancias, poder añadir instancias maestras dedicadas, y mejorar las máquinas EC2.

En la parte el almacenamiento es una configuración común para cada instancia, tipo de almacenamiento **EBS**, el estándar de AWS, con SSDs de 10GB por nodo del cluster. Para la parte de seguridad se pueden configurar el cifrado entre máquinas, backups, y la autenticación en **Kibana** mediante Amazon Cognito, que para el proceso no se tiene activo.

Para el *pipeline* montado para la suscripción de los filtros consta de 3 partes, la suscripción propiamente dicha, y la búsqueda de los filtros que coincidan con el animal y el envío de los emails.

Para la primera parte, una vez que el usuario ha registrado su email en el front, el backend envía el registro a un stream de Kinesis que mediante Kinesis Firehose se guarda en **Elasticsearch**.

Para la segunda parte, una vez que un animal se se registra se envía al Stream de registro de animales ahí se lanza una **AWS Lambda** para cada registro, la Lambda, codificada en **Java 8**, primero construye una query para **Elasticsearch**, una vez recupera todos los emails suscritos para cada uno los va enviando a otro Stream con el email y el MongoID del animal.

Tras este stream, usado para desacoplar el proceso de envío de email, que es algo costoso, de todo el conjunto de registros que se puedan encontrar en Elasticsearch, se encuentra otra **Lambda**, ésta escrita en **NodeJS**, que se encarga de realizar una petición HTTPS, de nuevo al *backend* a un endpoint que contiene el módulo de envío de emails.

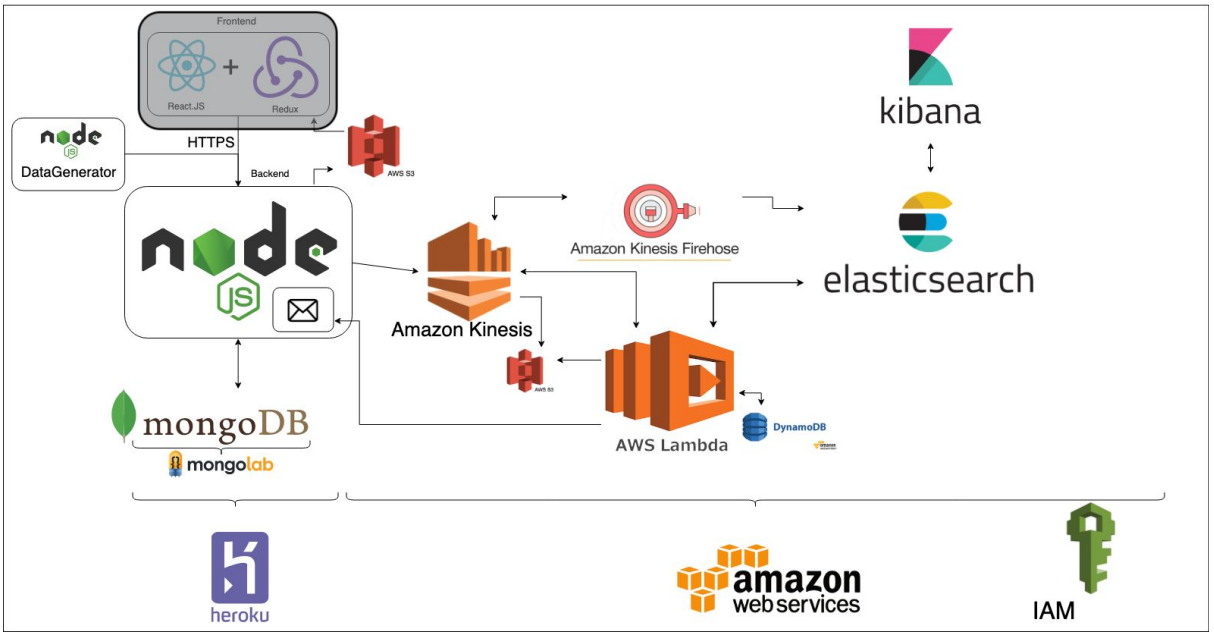
Este módulo, puramente software, recibe el los datos de envío del email, los valida, realiza una query usando el framework **Mongoose ODM** contra **MongoDB** para obtener el animal y los datos de su protectora mediante un *populate***.** Tras esto se renderiza una plantilla .hjs (Hogan, un módulo de NodeJS para este fin) con los datos y los envía a SendGrid, el servicio de emails contratado para la plataforma.

Tanto las **Lambdas** como **Kinesis Firehose** envían todos los logs a **AWS CloudWatch**, donde se pueden ven y depurar los procesos. Adicionalmente, como en casi todo AWS, estos dos servicios tiene un respaldo en **S3** de aquellos registros que hayan ocasionado un fallo en la ejecución. También se puede tener de todo lo que pase, pero decidí no tenerlo activo. Además aquellos procesos que sean consumidores de Kinesis necesitan acceder a

DynamoDB para poder guardar el offset de lectura por el que van, eso es transparente para el programador, salvo por la parte de permisos de acceso.

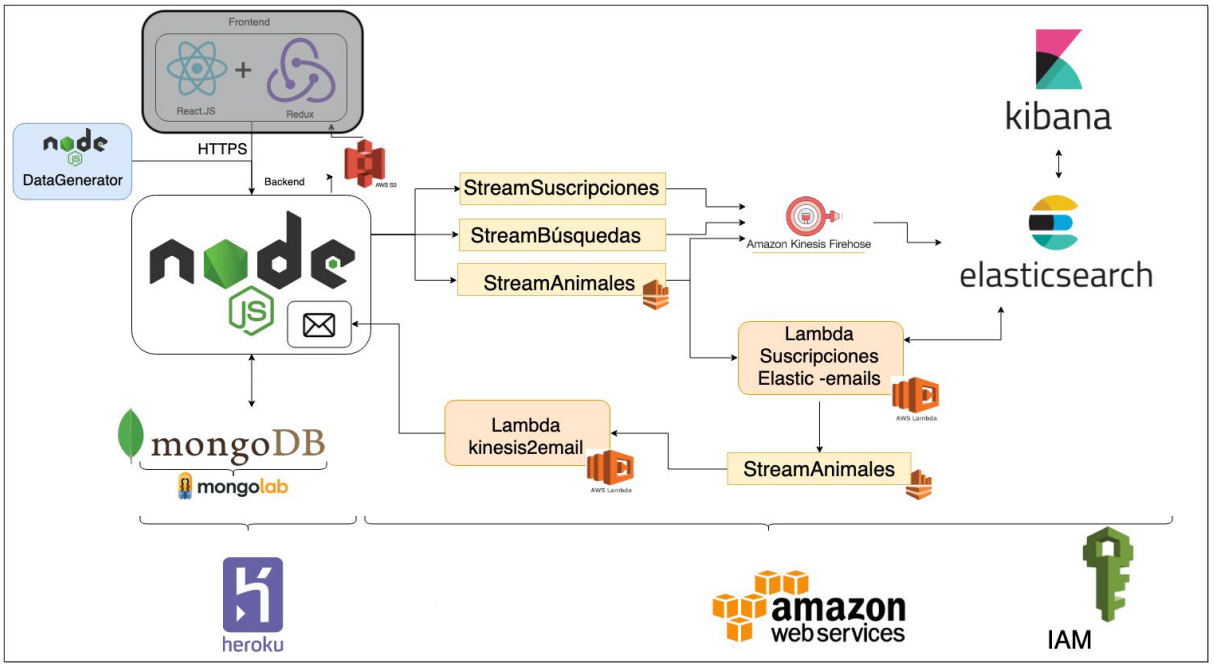
Por último, y no por ello menos importante, aunque sí lo más tedioso de todo, es la gestión de los permisos, las políticas, los roles y los usuarios. Todo ello gestionado con **AWS IAM** (Identity and Access Management), es el gestor de claves y permisos, cada servicio que provee Amazon tienes “acciones” (p.e.: putRecord, DescribeStream, HTTP GET, ...) y se pueden gestionar agrupándolos en “políticas”, (p.e.: política de publicación en el StreamAnimales, que incluye las acciones necesarias para poder hacerlo) y estas a su vez se agrupan en roles, que ya son a nivel de aplicación (p.e.: Rol de Lambda-suscripcion-elastic-email) que contiene los roles necesarios para poder consultar Elasticsearch por un lado y otro para poder publicar en el Stream de emails.

A continuación se puede ver un diagrama que muestra las distintas tecnologías y cómo están conectadas unas con otras. Hay que destacar que la capa frontend queda totalmente fuera del desarrollo de este TFM, aunque se sitúa para tener una visión global. Para ello se ha desarrollado un módulo que genera búsquedas y nuevos animales y los envía por HTTPS al *backend* para simular actividad del *frontend*.



4. Arquitectura de componentes

Despliegue de los distintos componentes desarrollados y los flujos de datos que generan entre ellos.



5. Módulos desarrollados

Data Generator

El generador de datos es una aplicación en NodeJS que levanta dos procesos y a cada uno le asigna generar animales y búsquedas con datos aleatorios siguiendo el modelo de la plataforma. Posteriormente se envían los registros generados mediante REST a la API para simular la actividad del *frontend*.

Backend y envío a Kinesis

Dado que la plataforma web es parte de un desarrollo empresarial junto con mi socio, decidimos que no se publicase por razones entendibles, por ello solo se ha entregado los retazos de código pertenecientes al desarrollo del productor de Kinesis.

El productor se puede encontrar en el fichero
`/□negocio□/□analytics□/sendToAnalytics.js`

y su uso se puede encontrar en el ejemplo
`/□negocio□/□Serv_aplicacion/SA_Busquedas.js`

en el que se envían los registros para las suscripciones, además de dos funciones para el filtrado/normalización de los datos.

`/□negocio□/□modelos/*` están los esquemas de *mongoose* con el modelo de datos usado para el TFM.

Lambda Suscripciones-Elastic-emails

Esta lambda se dispara cada vez que cae un registro en el StreamAnimales, para cada registro decodifica el json, obtiene los campos del filtro de suscripciones (especie, raza, sexo, edad y tamaño), después llama al conector con Elastic que construye una query, la ejecuta y obtiene los emails del resultado, aplicando un *distict* para no enviar varios emails iguales. Tras esto llama al conector de Kinesis y usando la librería KPL (Kinesis Producer Library, AWS probé otra con el SDK, que funciona bastante peor), este serializa el JSON y lo envía al StreamEmails.

En esta lambda me resultó bastante curioso las buenas prácticas que recomienda Amazon para las Lambdas, por que varias de ellas se desaconsejan normalmente en la programación en Java, la principal es que recomienda que cuantas más cosas *static* haya casi mejor. Ciertamente al ser conceptualmente una función la idea es que su ejecución sea lo más liviana posible, por lo que los dos conectores siguen el patrón *Singleton*, de esta forma la primera tarda más por tener que instanciar el entorno y los conectores, pero las siguientes es abrumadoramente rápido, como dato pasá de tardar la ejecución completa de 9 segundo a 150ms.

Lambda kinesis2email

Esta Lambda, escrita en NodeJS (de forma que pueda tener ejemplos en ambos lenguajes) se encarga de recoger cada uno de los JSON de pares (email e id de animal) en *stream* anterior y para cada uno envía una petición HTTP a la API usando la librería Axios para el envío de emails. Esta lambda puede parecer no tener mucho sentido al ser tan sencilla, pero al tener los procesos separados por stream se evitan cuellos de botella en los procesos, haciéndolo asíncrono y dado que la compilación de la template del email y el envio es algo costosos y podrían acumularse muchos envíos de golpe por las consultas a elastic creo que era la mejor forma de hacerlo.

Backend envío de emails

Este módulo desarrollado en el backend de la plataforma sigue el siguiente flujo:

El router pasa la petición al servicio de aplicación de emails), este valida los datos, y enriquece la información recibida consultando la base de datos operacional, MongoDB usando el ODM Mongoose.

Una vez enriquecido se pasa al controlador de emails que se encarga de enviarlo. Una vez enviado se realiza una contingencia del email en mongo como respaldo.

En controlador de email valida que los datos necesarios para el envío son correctos y pasa a procesar el email, compilando la plantilla .hjs con los datos del modelo de animal y de la protectora para generar un HTML. Este se pasa al *callback* que construye la petición y la manda al servicio SendGrid