Prueba Técnica 10

La solución a la prueba técnica que voy a describir en este documento ha sido desarrollada utilizando las siguientes herramientas y tecnologías:

- **Node.js** v12.16.1
- **Docker** 18.09.9
  - **MongoDB** mongo:4.2.3 docker image
- **Apache Kafka** 2.12-2.4.0
- **Visual Studio Code** 1.42.1
- **Postman** v7.18.0

En una estación con sistema operativo:

- **Ubuntu** 18.04.3 LTS (bionic)

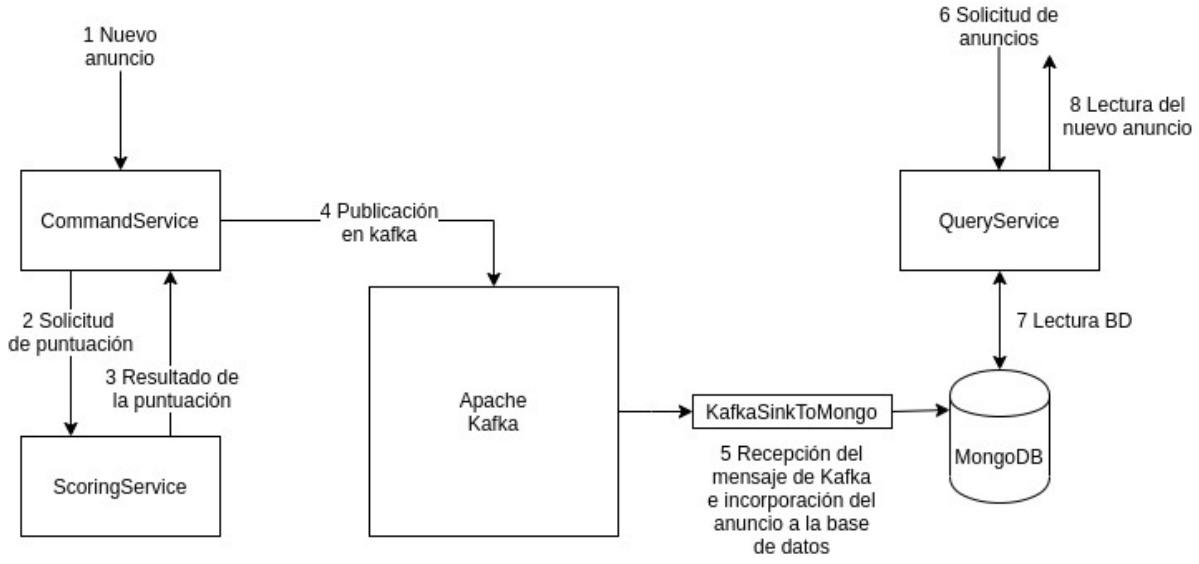
Estructura general

La solución propuesta esta formada por una serie de microservicios que se complementan entre sí para satisfacer los requisitos del portal de anuncios.

Los servicios desarrollados son los siguientes:

- **Servicio de puntuación de anuncios.** Se encarga de calcular la valoración de un anuncio.
- **Servicio de consulta de anuncios.** Su función es proporcionar una API para la extracción de la información de los anuncios del sistema de persistencia.
- **Servicio de inserción de anuncios.** Será utilizada para incorporar nuevos anuncios al sistema.

El flujo que sigue un nuevo anuncio, desde que es creado, hasta que es consultado en el portal, es el siguiente:



1.- El nuevo anuncio es introducido en el sistema a través del servicio de inserción de anuncios (**CommandService**).

2.- **CommandService** solicita una evaluación del anuncio al servicio de puntuación de anuncios (**ScoringService**).

3.- **ScoringService** evaluá el anuncio y la puntuación es comunicada de vuelta a **CommandService**.

4.- **CommandService** realiza una publicación en **Apache Kafka** con la información del nuevo anuncio, incluida su puntuación.

5.- El mensaje de **Kafka** es recogido por un **consumidor** de Kafka y este lo incluye en la base de datos **MongoDB**.

6, 7 y 8.- En la próxima solicitud de anuncios realizada al servicio de consulta de anuncios (**QueryService**), el nuevo anuncio será devuelto al usuario.

Acerca de la arquitectura escogida

Al diseñar esta arquitectura, he tenido siempre en mente tratar de conseguir una alta independencia entre los servicios que la componen.

Es por esto que he utilizado el patrón de diseño arquitectónico **CQRS** (Command-Query-Responsability-Segregation), intentado separar y mantener independientes los servicios de introducción de información (Command), y los de consulta de la misma (Query).

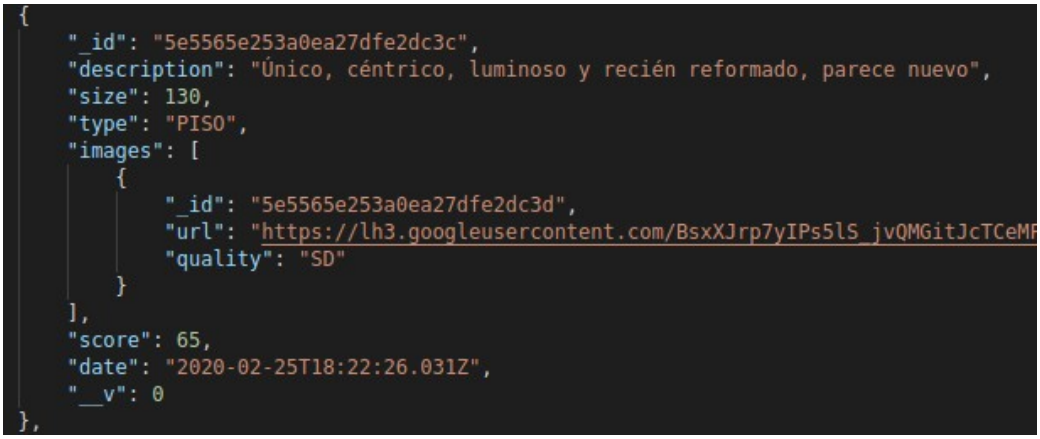
En referencia a esto, la utilización de **Apache Kafka** como mecanismo de intercambio de información entre ambas partes, nos permite tener el servicios de introducción de anuncios completamente independiente del sistema de persistencia (**MongoDB**).

Adicionalmente, quería indicar, que algunas partes del sistema no son todo lo independientes que podrían ser. Es el caso de CommandService y ScoringService. Estos servicios se comunican directamente entre ellos.

Me hubiera gustado profundizar en este asunto e incorporar un componente **Sidecar** a cada servicio para que se encargue de las comunicaciones con el exterior, y aunque seguro que hubiera sido muy interesante poder mostrároslo, el tiempo que me hubiera requerido se escaparía del que dispongo para realizar este ejercicio.

Sobre la estructura de la información en MongoDB

A pesar de obtener la información de prueba segregada en dos objetos diferentes, por un lado los anuncios y por otro las imágenes, he decidido que para los propósitos del sistema y el ámbito de la prueba resulta más conveniente mantenerlos juntos en la persistencia.



La razón de tomar esta medida ha sido evitar complicar en exceso el flujo de la información en el sistema. Mantener dos colecciones en MongoDB, supondría:

- Tener una nueva ruta en el servicio Command para soportar el nuevo tipo de entrada (imágenes).
- Tener un nuevo Topic en Kafka para notificar la nueva imagen.
- El servicio de Command tendría que ir a buscar a algún lugar (Mongo?) las imágenes antes de solicitar la valoración a ScoringService, ya que las imágenes son necesarias para evaluar un anuncio.
- Y por último, aunque es el menor de los males, me hubiera obligado a realizar un join de las dos colecciones en el servicio de Query, para mostrar la información completa al usuario solicitante (O bien tener 2 endpoints para que se soliciten por separado imágenes y anuncios).

Por estas razones, y porque el tiempo de desarrollo debía mantenerse relativamente bajo, he decidido mantener la información de forma compacta en una sola colección de Mongo.

Además, como se puede apreciar en la imagen, he almacenado la puntuación de los anuncios en la persistencia. En un principio, al no ver el campo en la información de prueba proporcionada, pensé en puntuar los anuncios al obtenerlos de la base de datos. Sin embargo, pronto me di cuenta de que este enfoque no era adecuado.

Puntuar los anuncios en cada lectura supondría tener que leer todos los anuncios de la colección cada vez que queremos mostrarlos. Entre los requisitos para el visitante comprador, se encuentra que los mensajes apareciesen ordenados por puntuación. Así que como íbamos a saber que anuncios debían aparecer en primer lugar, si no teníamos la puntuación en la base de datos.

No nos queda otra que guardarla o que calcularla para todos los mensajes en cada query. Me quedo con la primera.

No es oro todo lo que reluce, pues hacer persistir la puntuación también nos trae algún problema. Y es que, ¿que ocurriría si cambiamos los criterios de puntuación? Esto nos obligaría a recalcular la puntuación de todos los anuncios almacenados.

En cualquier caso, el número de solicitudes de anuncios siempre superara muy ampliamente al número de cambios en los criterios en un portal de éxito como el nuestro, así que he tomado la opción de almacenar la puntuación.

Descripción de los servicios que componen el sistema

Antes de comenzar quería indicar que los servicios **Node.js** han sido desarrollados siguiendo la arquitectura **route-controller-service-model**. No obstante no me he ajustado al 100% a la arquitectura, ya que en algunos de servicios he omitido al fichero de **rutas**, por la razón de que tan solo iba a contener una ruta, y tenerla en el fichero principal definida no me parece un gran inconveniente, cuando es una cantidad tan pequeña como una.

Servicio de puntuación de anuncios (ScoringService)

Este servicio recibe en el cuerpo de la petición el anuncio a evaluar y, tras realizar la evaluación, responde con el anuncio, con la puntuación incorporada.

Entrada:


```
{
    "description":"Ocasión!. Nuevo Audi a3",
    "km":"0",
    "fabricant":"Audi",
    "type":"VEHICULO",
    "images":[
      {
        "url":"https://lh3.googleusercontent.com/V3r1ufJ12GoxxqfiS7KOVOKj08Zbc6URkdHUK6N_kSyLAX1RE4z2URDga0w99jo5sKRPuKS70YvXHFDAHfEb6K1HVvsZ8NnOjyFT-Xhw",
        "quality":"SD"
      },
      {
        "url":"https://lh3.googleusercontent.com/jkguSbkaV4tkU3OEvKo5ebAhX9_rAoSwxwU2g3pS5v8winD3Mnl8ZT4WYThAwqOvjm8h2RO_EG57z6Yfbv-Ek_NdWULEdD7OqCkYCdOANQ",
        "quality":"HD"
      },
      {
        "url":"https://lh3.googleusercontent.com/LeSa57xyAqiF_IlnidwRIXoNrN9j-nB0Xoe_L3iCsVsfleMc-q0aKLGzYznYXqRh_hwj8wmnQDQt99IEUGoCTnKhhCvc_R-Dz9Iz00M9BA",
        "quality":"HD"
      }
    ]
}
```


Salida:


```
{
    "status": 200,
    "data": {
        "description": "Ocasión!. Nuevo Audi a3",
        "km": "0",
        "fabricant": "Audi",
        "type": "VEHICULO",
        "images": [
            {
                "url": "https://lh3.googleusercontent.com/V3r1ufJ12GoxxqfiS7KOVOKj08Zbc6URkdHUK6N_kSyLAX1RE4z2URDga0w99jo5sKRPuKS70YvXHFDAHfEb6K1HVvsZ8NnOjyFT-Xhw",
                "quality": "SD"
            },
            {
                "url": "https://lh3.googleusercontent.com/jkguSbkaV4tkU3OEvKo5ebAhX9_rAoSwxwU2g3pS5v8winD3Mnl8ZT4WYThAwqOvjm8h2RO_EG57z6Yfbv-Ek_NdWULEdD7OqCkYCdOANQ",
                "quality": "HD"
            },
            {
                "url": "https://lh3.googleusercontent.com/LeSa57xyAqiF_IlnidwRIXoNrN9j-nB0Xoe_L3iCsVsfleMc-q0aKLGzYznYXqRh_hwj8wmnQDQt99IEUGoCTnKhhCvc_R-Dz9Iz00M9BA",
                "quality": "HD"
            }
        ],
        "score": 65
    },
    "message": "Advert succesfully scored"
}
```

Para desarrollar este servicio, en primer lugar he agrupado en una tabla los diferentes criterios de puntuación de cada tipo.

<table>
<tr><th></th><th>Chalet</th><th>Piso</th><th>Vehículo</th><th>Frigorífico</th></tr>
<tr><td>Sin fotos</td><td colspan="4">-10 puntos</td></tr>
<tr><td>Cada foto HD</td><td colspan="4">+20 puntos</td></tr>
<tr><td>Cada foto no HD</td><td colspan="4">+10 puntos</td></tr>
<tr><td>Con descripción</td><td colspan="4">+5 puntos</td></tr>
<tr><td>Longitud descripción</td><td>> 50 palabras
+20 puntos</td><td>>= 20 palabras
+10 puntos

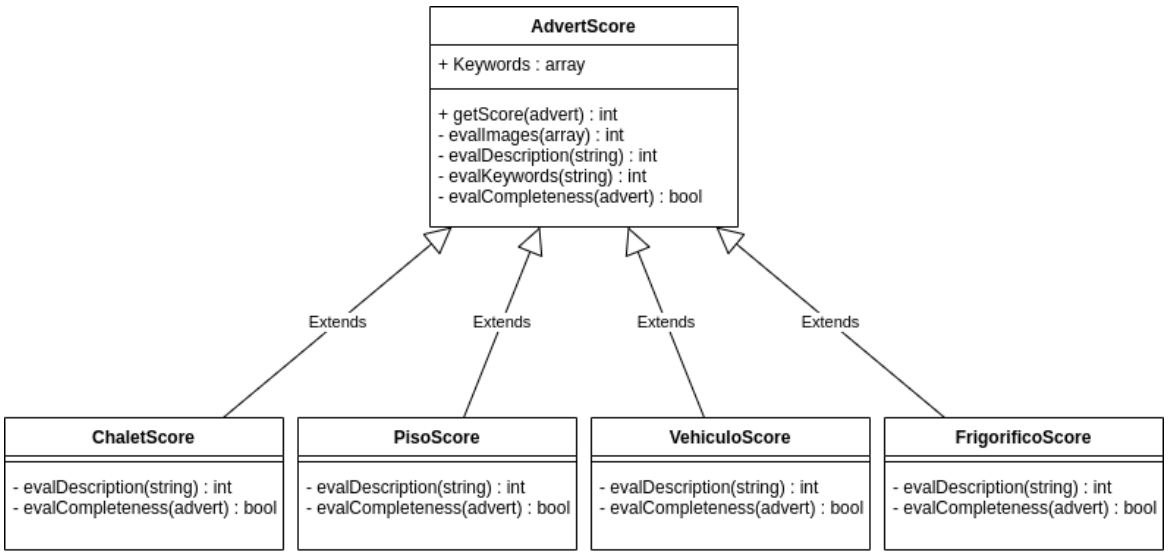
>= 50 palabras
+30 puntos</td><td colspan="2">-</td></tr>
<tr><td>Keywords en descripción</td><td colspan="4">+5 puntos cada una</td></tr>
<tr><td rowspan="4">Anuncio completo</td><td colspan="4">• Al menos una foto</td></tr>
<tr><td colspan="3">• Con descripción</td><td>-</td></tr>
<tr><td colspan="2">• Superficie (m^2)</td><td>• Kilometraje
• Color
• Fabricante</td><td>• Altura (cm)</td></tr>
<tr><td colspan="4">+40 puntos</td></tr>
</table>

Observo que hay ciertos criterios que son comunes para todos los tipos de anuncio, sin embargo existen algunos específicos para cada uno de ellos.

Podríamos tener toda está lógica de negocio dentro de un solo fichero. Pero al tener la necesidad de utilizar diferentes criterios para puntuar unos tipos u otros, sería interesante lograr separar esta lógica en diferentes ficheros. Esto nos permitirá modificar de forma más sencilla los criterios de un tipo e incluso añadir otros tipos de anuncios al sistema sin demasiadas dificultades.

Por otra parte, hay ciertos criterios comunes a todos los tipos. Por lo que necesitamos una solución que nos permita definir los criterios específicos de cada tipo, sin repetir la parte común en cada uno de los tipos.

Esto lo conseguiremos aplicando el patrón de diseño **Template.** Definiré una clase base donde situaré la lógica de puntuación común a todos los tipos. Los tipos concretos heredarán de este clase base y sobrescribirán los métodos que contienen lógica específica de puntuación para el tipo concreto.



Cada una de las clases concretas tiene la responsabilidad de calcular la puntuación para el tipo concreto. Sin embargo, la lógica de puntuación común a todos los tipos, como la evaluación de las keywords, se encuentra en AdvertScore.

AdvertScore se ha definido simulando ser una clase abstracta. Esto es, no puede ser instanciada. La razón es que no nos interesa que esta instanciación pueda producirse, porque las clases concretas son siempre las que deberían ser utilizadas para evaluar un anuncio.

Con esta arquitectura del programa:

- Es muy sencillo alterar un criterio común a todos los tipos. Tan solo debemos modificar la clase AdvertScore.
- Es muy sencillo alterar un criterio específico de un tipo. Tan solo debemos modificar la clase concreta de ese tipo.
- Es muy fácil añadir un nuevo tipo de anuncio. Tan solo debemos crear una nueva clase que extienda AdvertScore y definir los métodos de evaluación específicos del nuevo tipo.

Adicionalmente, he utilizado el patrón de diseño **Factory** para encapsular la lógica de decisión entre elegir una clase evaluadora u otra.

Durante el desarrollo de este servicio, debido a la naturaleza del problema, me ha sido de gran utilidad la realización de **test unitarios** con la metodología **TDD**. Me ha permitido poder refactorizar el código continuamente con la confianza de no romper algo que ya funcionaba.

```
koski@koski-Inspiron-5570:~/workspace/PT10/ScoringService$ npm run test advert-score.test.js

> scoringservice@1.0.0 test /home/koski/workspace/PT10/ScoringService
> jest "advert-score.test.js"

 PASS  ./advert-score.test.js
  ✓ AdvertScore can't be instantiated because it's abstract (5ms)
  Image scoring tests
    ✓ Adverts with no images score -10
    ✓ Adverts score 20 per HD image & 10 per other images (1ms)
  Description scoring tests
    ✓ Adverts score 0 if they have no description (1ms)
    ✓ Adverts score 5 if they have description
    ✓ evalDescriptionLength is called in concrete classes (1ms)
    ✓ evalDescriptionLength return value is added
    ✓ evalKeywords gets 5 points per keyword (1ms)
    ✓ evalKeywords doesn't get points for keyword repetitions
    ✓ Letter case is ignored when searching for keywords
    ✓ But accent mark is not. You should write propertly
    ✓ Keywords preffixes or suffixes invalidate the match (1ms)
  Completeness tests
    ✓ evalCompleteness is true when there is one image at least (1ms)
  Main function getScore tests
    ✓ Parcial scoring functions are called (5ms)
    ✓ Parcial scores are added and returned (1ms)
    ✓ If advert is complete, then 40 points are added

Test Suites: 1 passed, 1 total
Tests:       16 passed, 16 total
Snapshots:   0 total
Time:        0.694s, estimated 1s
Ran all test suites matching /advert-score.test.js/i.
koski@koski-Inspiron-5570:~/workspace/PT10/ScoringService$
```

Endpoint: http://localhost:9010/score

Servicio de inserción de anuncios (CommandService)

Este servicio recibirá los nuevos anuncios que deben ser introducidos en el portal. La información recibida es enviada al servicio de puntuación previamente descrito, y tras recibir la respuesta, el anuncio junto con su puntuación es entonces publicada en Apache Kafka.

Lo interesante de este esquema, es que ante una posible caída del sistema de persistencia (MongoDB), seguiría siendo posible físicamente la "inserción" de nuevos anuncios. Estos serían publicados en Kafka, y cuando todo volviese a la normalidad, la persistencia sería actualizada, cuando los mensajes fuesen capturados por el consumidor adyacente a esta.

```
require('dotenv').config('.env')
const kafka = require('kafka-node')

exports.postAdvert = async function (advert) {
  const { KAFKA_HOST: host, KAFKA_PORT: port, KAFKA_TOPIC: topic } = process.env
  const kafkaOptions = { kafkaHost: `${host}:${port}` }

  const client = new kafka.KafkaClient(kafkaOptions)
  const producer = new kafka.Producer(client)

  console.log(advert)
  const payloads = [
    {
      topic: topic,
      messages: JSON.stringify(advert)
    }
  ]

  producer.send(payloads, (err, data) => {
    if (err) {
      console.error(err)
      console.error('[kafkaProducer -> ' + topic + ']: broker update failed')
    } else {
      console.log('[kafkaProducer -> ' + topic + ']: broker update success')
    }
  })

  producer.on('error', function (err) {
    console.error(err)
    console.error('[kafkaProducer -> ' + topic + ']: connection errored')
    throw err
  })
}

```

Kafka-Producer @ CommandService

Endpoint: http://localhost:9040/advert

Servicio de consulta de anuncios (QueryService)

Este servicio proporciona una API al usuario para la consulta de anuncios.

Por defecto se incluyen un máximo de 10 anuncios en la respuesta, pero esto puede ser modificado mediante el uso de ciertos parámetros:

- **limit**: Establece el máximo número de anuncios en la respuesta del servicio. Valor por defecto: 10.
- **page**: Establece la página de los resultados que deseas obtener. Valor por defecto: 0.

Es decir, con **page=0&limit=10** veríamos tan solo los primeros 10 anuncios. Si quisiéramos obtener los siguientes 10 tendríamos que indicar los valores **page=1&limit=10**.

Adicionalmente a estos 2 parámetros para limitar el tamaño de la respuesta, se han creado algunos para tratar de filtrar el tipo de anuncios que buscamos.

Es el caso de los parámetros **minScore** y **maxScore** que nos proporcionan el mecanismo necesario para resolver el requisito por parte de los visitantes compradores de no ver los anuncios irrelevantes (minScore=40) y de verlos por parte de los supervisores (maxScore=39).

Finalmente, se ha implementado el parámetro **sortBy**, que puede ser utilizado con el valor "**-score"** para mostrar los anuncios al visitante comprador ordenados de mejor a peor, tal como fue solicitado.

API call de visitante comprador y respuesta:

http://localhost:9020/adverts?sortBy=-score&minScore=40


```
{
    "status": 200,
    "data": [
        {
            "_id": "5e5565ea53a0ea27dfe2dc3e",
            "description": "Pisazo",
            "size": 115,
            "type": "PISO",
            "images": [
                {
                    "_id": "5e5565ea53a0ea27dfe2dc3f",
                    "url": "https://lh3.googleusercontent.com/r
                    "quality": "SD"
                },
                {
                    "_id": "5e5565ea53a0ea27dfe2dc40",
                    "url": "https://lh3.googleusercontent.com/L
                    "quality": "HD"
                }
            ],
            "score": 75,
            "date": "2020-02-25T18:22:34.076Z",
            "__v": 0
        },
        {
            "_id": "5e5565e253a0ea27dfe2dc3c",
            "description": "Único, céntrico, luminoso y recién
            "size": 130,
            "type": "PISO",
            "images": [
                {
                    "_id": "5e5565e253a0ea27dfe2dc3d",
                    "url": "https://lh3.googleusercontent.com/B
                    "quality": "SD"
                }
            ],
            "score": 65,
            "date": "2020-02-25T18:22:26.031Z",
```

Otros componentes del sistema

Apache Kafka

La arquitectura diseñada emplea Apache Kafka para comunicar ambas partes del sistema (Command & Query).

Durante el desarrollo he podido emplear varias configuraciones para el cluster de Apache Kafka.

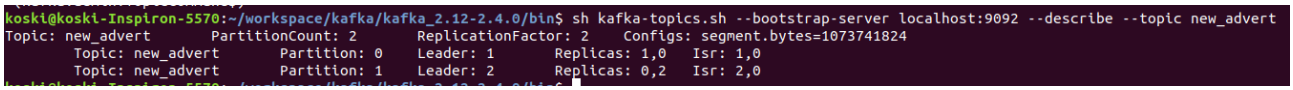


*Imagen tomada durante el desarrollo, en el que se emplea un cluster con **1 zookeeper** y **3 brokers**. La información que se muestra en la imagen corresponde al tópic empleado en el sistema. Los mensajes están dividen en 2 particiones con factor de replicación 2.*

La partición 0 tiene presencia en el broker 0 y 1, siendo líder de la partición el broker 1.

La partición 1 tiene presencia en el broker 0 y 2, siendo líder de la partición el broker 2.

La elección de esta configuración está basada en tratar de dotar al cluster con un cierto grado de robustez frente a caídas o desconexiones.

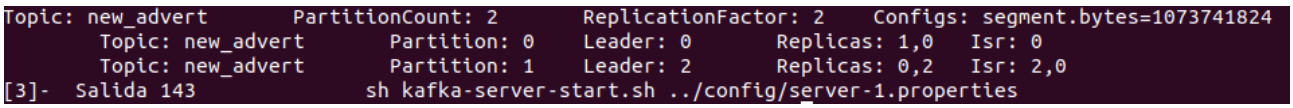
Con la caída de uno de los brokers, el sistema podría seguir funcionando, fuese cual fuese el nodo caído.

Si dos brokers son desconectados, dependiendo de cuales fuesen los nodos eliminados, el sistema podría seguir trabajando o no. Si los brokers eliminados fuesen el 1 y el 2, como ambas particiones tienen una replica en el broker 0, todo seguiría funcionando. Sin embargo, si perdiéramos el broker 0 junto con cualquier de los otros dos, nos faltaría la información contenida en una de las particiones del tópic.

No obstante, con la configuración actual, si el nodo caído es el zookeeper, entonces el sistema dejaría de funcionar, ya que de este servicio no tenemos nada más que una instancia. Sin embargo no es un obstáculo insalvable, pues al igual que podemos tener más de un broker, también podemos levantar varios zookeepers para que gestionen conjuntamente nuestro cluster.

Por supuesto, para conseguir esta resistencia frente a las adversidades, los elementos deberían estar distribuidos en diferentes localizaciones, y no todas en una sola máquina como estoy haciendo durante el desarrollo.

A modo de ejemplo voy a eliminar del sistema al broker 1, que actualmente es líder de la partición 0. Este es el resultado después de hacerlo:



Podemos observar como el líder de la partición 0 ha pasado a ser el broker 0. Además vemos que la partición 0 ya no se encuentra replicada en el broker 1 (ISR).

KafkaSinkToMongo

Este módulo es un pequeño programa, pero no por ello menos importante, cuyo único propósito, es la de alimentar la base de datos Mongo con los nuevos anuncios que son publicados en nuestro cluster Kafka.

Al iniciar, conecta con Kafka como consumidor del tópic 'new_advert' y se mantiene a la escucha de mensajes (e incluso obtiene aquellos que se puedan haber producido cuando no se encontraba activo).

Estos anuncios, que ya llegan junto con su puntuación, son incorporados directamente a la colección en MongoBD.

¿Como cambiaría esta solución si dispusiera de una mayor infraestructura?

Para beneficiarnos de una arquitectura basada en microservicios, es fundamental disponer de una infraestructura distribuida y conseguir bajo acoplamiento entre los servicios.

Por ello, en una infraestructura superior, sería deseable que los elementos que conforman la solución estuviesen distribuidos en diferentes servidores. Preferiblemente en contenedores, que serían orquestados con una herramienta como Kubernetes, que nos proporcionaría monitorización, escalado y balanceo de carga entre ellos.

Al existir escalado de servicios, estos no pueden referirse a los demás de forma directa. Necesitaríamos un servicio de descubrimiento de los mismos, y la comunicación entre ellos se haría a través de un módulo sidecar, que sería común en todos los servicios, y que sería el único elemento de los servicios capaz de comunicarse con el exterior.

Apache Kafka dispondría de más zookeepers, más brokers y un mayor factor de replicación de los tópics, para lograr la robustez que nuestro exitoso portal de anuncios necesita. Además los esquemas utilizados en los tópics (tan solo tenemos 1 tópic, pero en un futuro podrían ser más) deberían estar registrados, para conseguir la homogeneidad dentro de los mensajes de un mismo tópic.

Otras decisiones no arquitectónicas

A parte de las decisiones arquitectónicas ya justificadas, se han tomado algunas decisiones de otro tipo que también creo conveniente explicar:

- Cada uno de los elementos que han sido desarrollados se ha hecho utilizando un repositorio github diferente. La razón de esto es la de, ante una posible secuencia de acciones derivadas de un commit, como pueden ser la ejecución de tests, cálculo de métricas del código, publicación, etc., poder hacerlo sobre una parte pequeña del código. Aquella sobre la que se realiza el commit, en lugar de todo el sistema.
- Los servicios incluyen un fichero .env en el que puede alterarse algunos parámetros: puerto de escucha, endpoints de los otros servicios, etc. Este documento, la guía de instalación y el conjunto de request de Postman han sido desarrollados teniendo en cuenta la configuración por defecto de los servicios.
- Las funciones del módulo de KafkaSink deberían de ser implementadas utilizando el conector proporcionado por confluent: https://www.confluent.io/hub/mongodb/kafka-connect-mongodb. Ciertamente me hubiera gustado hacerlo, pero por razones de tiempo de desarrollo y por facilitar el despliegue de la aplicación de cara a que puedan probarla, he creído conveniente hacer este pequeño programa en Node.js.

Conclusión

He disfrutado mucho de cada momento que he dedicado a esta prueba. Node.js y Apache Kafka son ambas tecnologías apasionantes y me gustaría seguir profundizando más y seguir mejorando en ellas.

Soy consciente de que el sistema desarrollado no es perfecto. Siempre cabe alguna mejora. Pero pienso que he conseguido un buen equilibrio entre lo que era solicitado en la prueba y el tiempo de desarrollo que tenía disponible.

A pesar de esto, estoy muy contento con el resultado alcanzado. Me gustaría pensar que ustedes también van a creer que he hecho un buen trabajo, y que puedo tener una oportunidad con vosotros. Estoy convencido de que todo iría bien.

Sin más. Un saludo.

Pdt: Quizás para próximas revisiones pueda incorporar alguna de las mejoras y seguir haciendo crecer nuestro portal de anuncios.

Update 01/03/2020

Incluyo este apartado algunos días después de hacer entrega de la prueba, con permiso de Leo (“el update quita puntos”).

He estado pensando en un cambio que pienso que mejoraría el sistema. Es un cambio arquitectónico, y desarrollarlo supondría cambiar varios de los servicios ya entregados, así como esta documentación, la guía de instalación e incluso la colección de endpoints Postman.

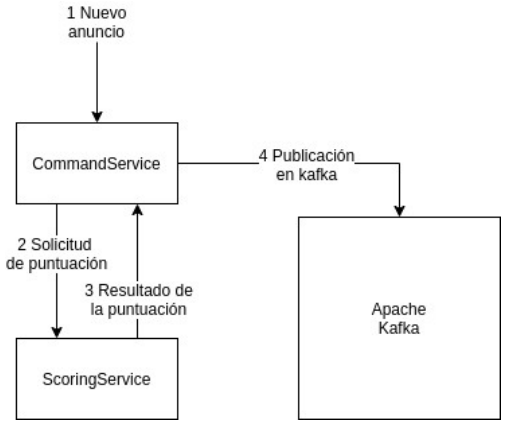
Es por esto que he creído suficiente con simplemente tratar de exponer la mejora en este documento, con la idea de que entendáis los beneficios que aportaría.

Me gustaría que no siguieran leyendo, y pasaran a probar la solución tal cual es ahora. Es entonces, una vez probada, cuando quisiera que vuelvan aquí, a leer sobre la mejora arquitectónica que propongo para la solución, a partir de la próxima página.

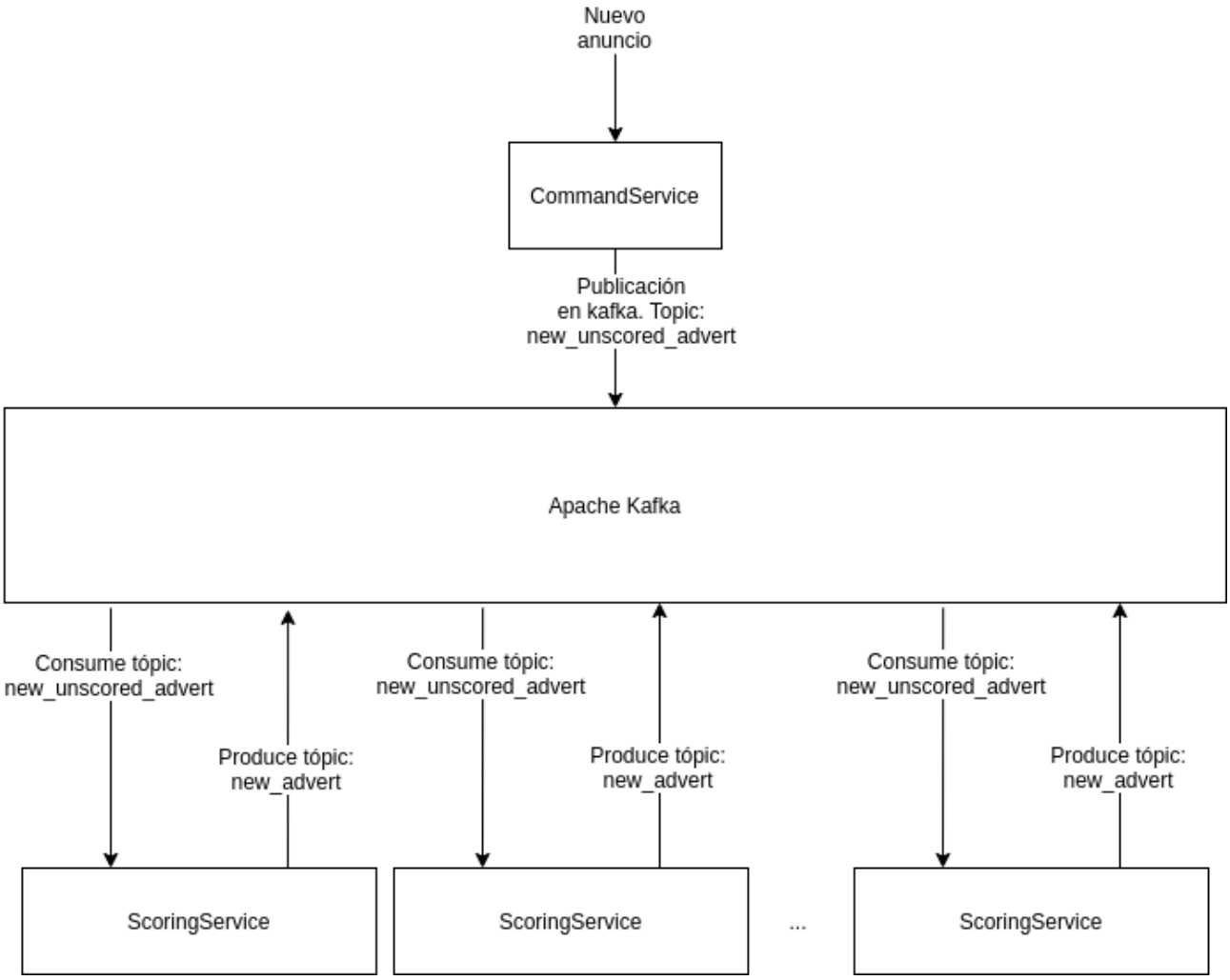
Ya está probada? Bien.

Se trata de modificar la arquitectura de la parte de inserción y puntuación de anuncios del sistema.

Actualmente tenemos esto:



Y vamos a transformarlo en esto otro:



La idea es desacoplar **CommandService** de **ScoringService**. Apoyándonos de nuevo en Apache Kafka.

Los anuncios entrantes serían publicados en Kafka en un **nuevo topic: new_unscored_advert**.

ScoringService dejaría de estar escuchando en un puerto para estar haciéndolo en el nuevo tópic. Sería un consumidor de este tópic.

Tras la puntuación, el anuncio sería publicado, junto con su puntuación, en el topic **“new_advert”** que ya teníamos en funcionamiento. No haría falta hacer ningún cambio en el resto de servicios del sistema, ya que **KafkaSingToMongo sigue trabajando con el tópic ya existente “new_advert”. Para él y QueryService el cambio sería transparente.**

Este cambio nos proporciona algunos beneficios interesantes:

- Tenemos un punto donde almacenamos los anuncios sin puntuar: Apache Kafka, en el nuevo tópic. Esto nos permitiría, si tuviéramos la necesidad, reprocesar todos los mensajes fácilmente (siempre que configuremos el tópic para que los mensajes no tengan caducidad) si por ejemplo queremos cambiar los criterios de puntuación o simplemente se han perdido las puntuaciones por cualquier razón.
- Un punto muy interesante es que esta arquitectura nos permite desplegar, como he sugerido en el esquema, **varias instancias de ScoringService** sin apenas dificultades:
    - La idea sería que **ScoringService** estuviera configurado como **consumidor en un grupo de consumo**, de forma que el trabajo de puntuar los anuncios, que sería mucho, porque nuestro portal va a triunfar, fuese distribuido entre las distintas instancias de ScoringService que despleguemos.
    - Para lograr esto, el tópic **“new_unscored_advert”** debería ser configurado para distribuir en **varias particiones** sus mensajes, ya que a los miembros del grupo de consumo se les asignan las particiones 1 a 1 (Si tengo más consumidores en el grupo que particiones, habría consumidores sin trabajo).

He grabado una prueba de esto último y quería enseñároslo funcionando: https://www.youtube.com/watch?v=SAaO2fp3qt4

Finalmente, quería indicar, que este enfoque también facilita una posible introducción de anuncios de forma masiva, cuando compremos wallapop. Tan “solo” deberíamos incluir todos los anuncios de la plataforma en forma de mensajes de Kafka y todos los servicios de puntuación se pondrían manos a la obra, a puntuar y a introducir estos anuncios ya evaluados en el sistema.

Y eso es todo. Un saludo!