UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

MÁSTER EN SOFTWARE CRAFTSMANSHIP

PROYECTO FIN DE MÁSTER

Título:

Comparativa de frameworks Java para el desarrollo de aplicaciones con arquitectura de microservicios desplegados en Kubernetes

Alumno:

Jose Luis Ojosnegros Manchón

Tutor(es):

Micael Gallego Carrillo

Fecha

Junio-2019

UNIVERSIDAD POLITÉCNICA DE MADRID

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA DE SISTEMAS INFORMÁTICOS

MÁSTER EN SOFTWARE CRAFTSMANSHIP

PROYECTO FIN DE MÁSTER

Título:

Comparativa de frameworks Java para el desarrollo de aplicaciones con arquitectura de microservicios desplegados en Kubernetes

Autor:
Jose Luis Ojosnegros Manchón
Tutor (es):
Micael Gallego Carrillo

Fecha
Junio 2019

Contenido

1. Introducción

1.1. Objetivos iniciales.

Tal y como se detallaba en el anteproyecto, los objetivos para este trabajo se resumían básicamente en comparar la viabilidad, así como las ventajas y posibles inconvenientes, que los nuevos frameworks de desarrollo de aplicaciones, nacidos específicamente para plataformas cloud, pudieran presentar con respecto a los frameworks ya existentes adaptados para el cloud. En concreto para la realización del trabajo se eligió Springboot 1 como framework nacido antes de la existencia del cloud y adaptado para el y Micronaut 2 como framework "cloud native".

La comparativa pretendía tomar en cuenta tanto aspectos de desarrollo y testing así como métricas de los distintos elementos, para poder comprobar si, como el framework Micronaut promete, la performance de su framework es superior.

1.2. Alcance del trabajo.

Debido a las limitaciones de tiempo ha resultado imposible realizar la composición completa de la aplicación y por tanto tomar el número de medidas de performance que habría sido deseable, más concretamente aquellas relacionadas con el comportamiento dinámico del sistema. Sin embargo el proceso de desarrollo así como las medidas tomadas, han permitido hacerse una idea sólida de la situación actual en la que se encuentra el desarrollo de ambos frameworks, haciendo posible obtener unas conclusiones válidas.

1.3. Marco Tecnológico.

1.3.1. Kubernetes

En las plataformas de cloud se suelen tener múltiples servicios ejecutándose en contenedores en lugar de tener una única aplicación monolítica. Esto implica nuevos problemas y dificultades derivados tanto del número de servicios como de las necesidades de escalado. Para atajar estos problemas surgieron los Orquestadores de Contenedores. Kubernetes es un orquestador de contenedores que nos permite organizar múltiples contenedores de manera que las

aplicaciones que corren dentro de estos estén activas y los contenedores distribuidos en un cluster. Kubernetes tiene las siguientes características:

- Scheduling automático: Permite lanzar contenedores en los nodos del cluster basándose en sus necesidades y requerimientos de recursos.
- "auto-curación": Cuando un nodo muere kubernetes reorganiza los contenedores que se ejecutaban en él. También termina los contenedores que no responden
- rollout & rollback automáticos: Permite realizar cambios en la aplicación o en la configuración de manera "secuencial" para evitar caídas de servicio, y si algo va mal puede hacer rollback del cambio.
- Escalado Horizontal y Balanceo de carga Permite escalar la aplicación con un simple comando o automáticamente basándose en los requisitos de la aplicación.

Arquitectura

En la gráfica podemos ver la arquitectura de k8s.

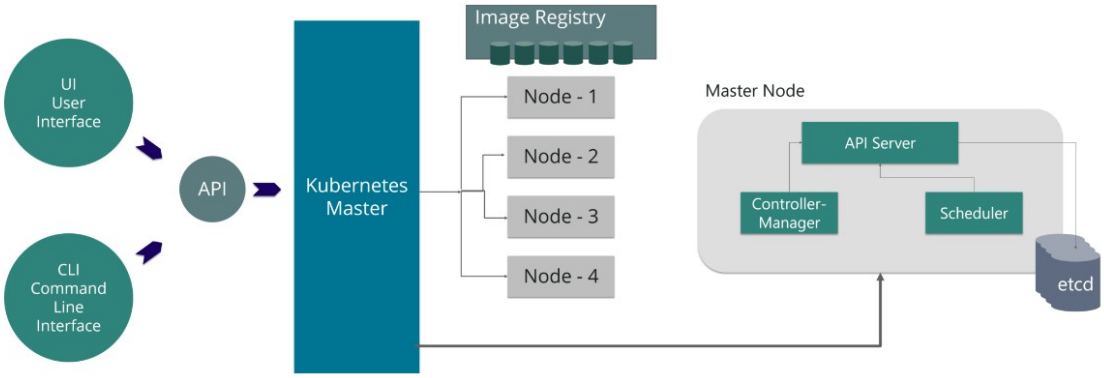

Illustration 1: K8s Arquitectura

También podemos ver aquí una vista detalle de un nodo esclavo:

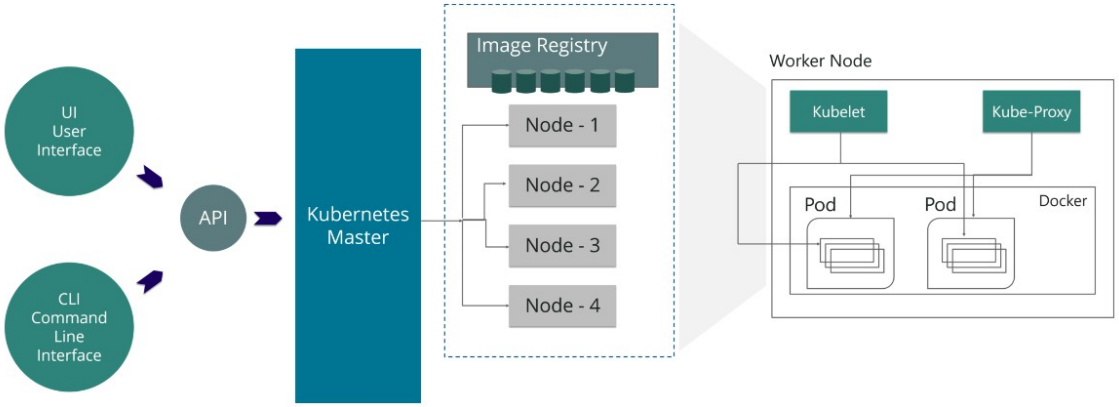

Illustration 2: K8s: Arquitectura (detalle Nodo Esclavo)

- Nodo Maestro: Responsable del manejo del cluster. Es el punto de entrada de todas las tareas administrativas sobre el cluster
- Nodos Esclavos:
  - Docker container: Se ejecuta en cada nodo esclavo.
  - Kubelet: Toma la configuración de un pod del api server y se asegura de que los contenedores adecuados están ejecutándose correctamente.
  - Kube-proxy: es un proxy de red y balanceador de carga
  - Pods: Uno o mas containers que ejecutan contiguos.
- Api Server : Es el punto de entrada de todos los comandos REST para poder controlar el cluster.
- Controller Manager
- Scheduler: Organiza tareas en los nodos esclavos
- ETCD: Es un key-value database distribuida utilizada principalmente para la configuración distribuida y el service discovery.

1.3.2. Springboot: Standard de facto

Springboot es un framework que toma como base el framework Spring y define una serie de configuraciones y componentes por defecto para hacer que la configuración del framework sea más sencilla y rápida. Spring, y Springboot, es actualmente uno de los frameworks de java más utilizados en el desarrollo de aplicaciones y, con su proyecto "Spring Cloud" ha intentado adaptarse al nuevo

paradigma de ejecución de aplicaciones en la nube, con bastante éxito. Por estas razones se decidió tomar este framework como "referencia de medida" a la hora de valorar los nuevos frameworks. Este framework basa su funcionamiento en la Inyección de dependencias y en la Inversión de control, utilizando para ello "runtime proxies", de manera que toda la manipulación del código se realiza en tiempo de ejecución, con el consiguiente gasto en tiempo y memoria.

En general, podemos decir que Springboot tiene soporte para casi cualquier software de terceros que tenga cierta utilización en el entorno cloud, su documentación es ingente y extensa, y tiene una configuración asequible, aunque un tanto verbosa a veces.

Para la realización de este trabajo se ha utilizado la versión 2.1.4.RELEASE de Springboot, ya que era la última con GA en el momento de comienzo del proyecto.

1.3.3. Micronaut: New kid on the block

Es un nuevo framework para la creación de aplicaciones que nace directamente enfocado hacia la creación de aplicaciones cloud, tanto microservicios como "serverless". Según su propia página web, la intención de Micronaut es crear un framework moderno que facilite el desarrollo de este tipo de "nuevas" aplicaciones, así como el de mejorar tanto los tiempos de arranque como el consumo de memoria tradicionalmente altos de este tipo de frameworks que hacen un uso extensivo e intensivo de la reflexión para implementar la Inyección de dependencias y la Inversión de control. Esta mejora es debida a que Micronaut, al contrario que Spring, no utiliza elementos en tiempo de ejecución para realizar las inyecciones, si no que utiliza un "Ahead-of-Time compiler" para realizar estas tareas en tiempo de compilación, lo que mejora mucho tanto el tiempo de ejecución como el consumo de memoria. También ofrece soporte para crear aplicaciones basadas en la máquina virtual de Graal lo que podría mejorar drásticamente las medidas de performance.

El framework aún está en sus primeras fases, su primer milestone fue en Mayo de 2018, mientras que su primera versión oficial se dio en Octubre de ese mismo año.

Para la realización de este trabajo se ha utilizado la versión 1.1.1

1.4. Desarrollo de los objetivos.

Para poder hacerse una idea de las facilidades e inconvenientes que cada uno de los frameworks podía presentar, así como para poder tomar una serie de medidas suficientemente significativas, se hacía necesario desarrollar una aplicación con una arquitectura de microservicios que fuese a la vez, lo suficientemente sencilla para poder realizar su desarrollo utilizando ambos frameworks en el tiempo disponible y lo suficientemente realista como para que las problemáticas a enfrentar permitiesen hacerse una idea de la situación actual de los frameworks.

2. Aplicación.

La aplicación elegida para ilustrar este trabajo es completamente ficticia, y se basa en un sistema de control simplificado de una plataforma de emisión de películas por streaming.

La aplicación está formada por multiples servicios que colaboran entre ellos para permitir a un usuario que visualice entre un grupo de contenidos multimedia, con la posibilidad de obtener aquellos que se acerquen más a sus gustos personales, y de ordenar su visualización.

Este sistema presenta una interface REST hacia el usuario, suponiendo que existe una aplicación cliente que consumirá dicho interface REST y lo presentará mediante una interface más amigable para un ser humano. También supone la existencia de un sistema de visualización de los streams multimedia con el que se comunicará y al que enviar ordenes que resulten en el envío del streaming de vídeo al usuario final.

La aplicación está pensada para ser utilizada sobre kubernetes.

2.1. Esquema general de la aplicación.

En el siguiente diagrama de contexto podemos ver las interacciones de la aplicación con los sistemas externos antes mencionados:

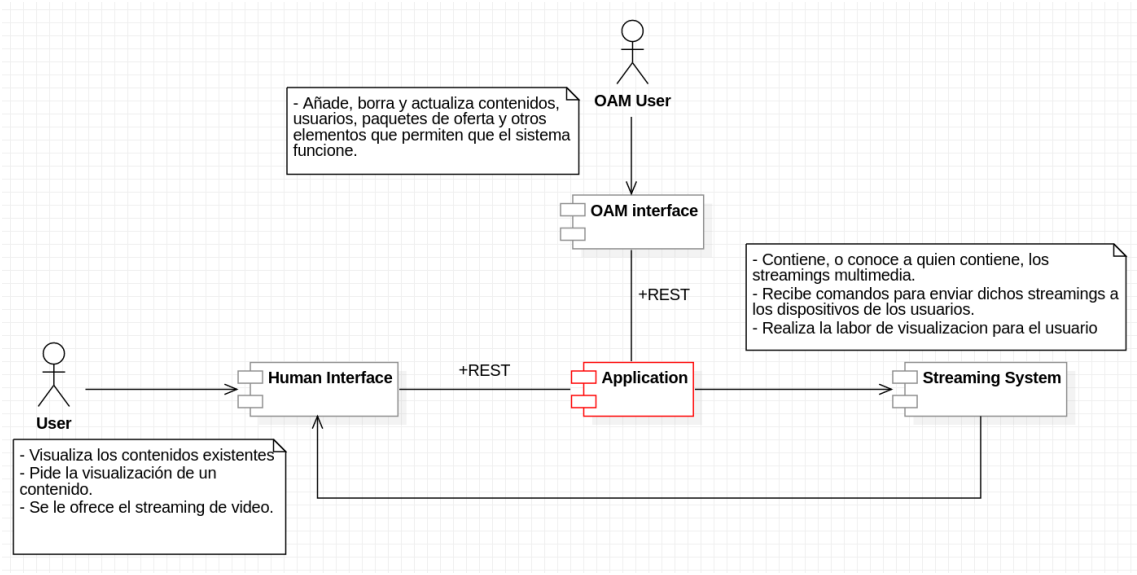

Illustration 3: Aplicación: Diagrama de Contexto del Sistema

2.2. Descripción de los distintos microservicios.

La aplicación está formada por seis microservicios, cada uno de ellos con una responsabilidad única en el sistema. En el siguiente diagrama podemos ver los microservicios y los interfaces que proveen y consumen cada uno de ellos.

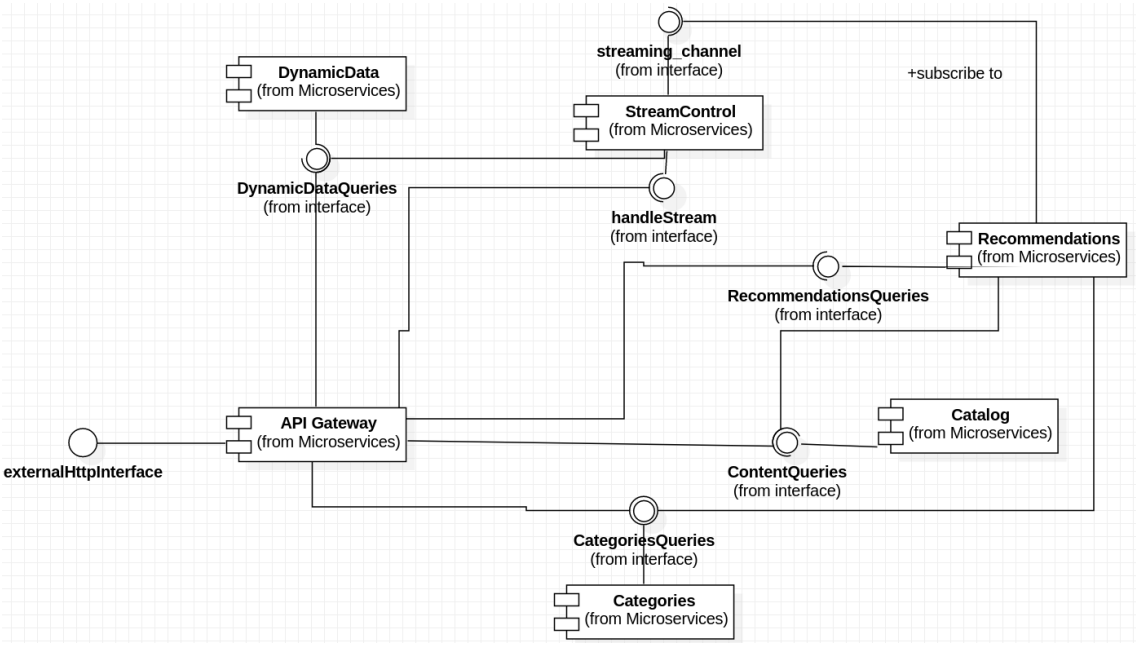

Illustration 4: Aplicación: Esquema de microservicios

Seguidamente se van a detallar los microservicios que forman la aplicación mostrando principalmente la responsabilidad de cada uno de ellos dentro del sistema, así como las distintas tecnologías utilizadas para su realización, dato importante a la hora de valorar más adelante el soporte de que ambos frameworks disponen para relacionarse de manera sencilla con la ingente cantidad de software que puebla el ámbito del cloud.

En el siguiente esquema podemos ver el modelo de datos de la aplicación, sirva como referencia pues se hablará de estos elementos en las descripciones de los microservicios.

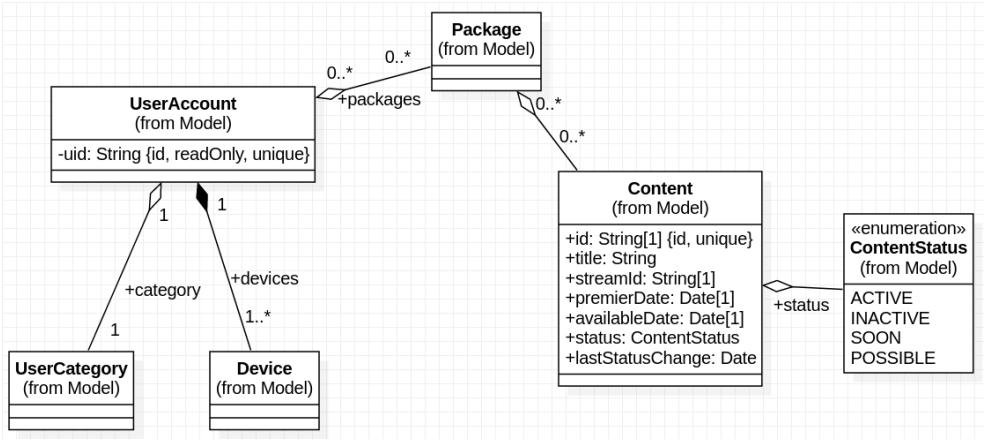

Illustration 5: Aplicación: Modelo

StreamControl.

Es el punto de comunicación de la aplicación con el sistema externo de visualización de contenidos multimedia.

Responsabilidad

Envía comandos a dicho sistema para que los contenidos se vuelquen mediante streaming a los dispositivos de los usuarios. Guarda una relación de qué contenido está visualizando cada usuario en cada uno de sus dispositivos en un momento dado, para ello utilizar una base de datos Redis.

Entradas.

Presenta un interface RPC atendiendo a los comandos PLAY,PAUSE y STOP, para ello utiliza RabbitMq RPC.

Salidas.

A parte de los comandos al sistema externo, este microservicio publica notificaciones sobre cada uno de los comandos ejecutados sobre un stream multimedia, para ello utiliza RabbitMq

Colaboraciones.

Realiza queries al microservicio DynamicData para poder relacionar un identificador de dispositivo (deviceId) activo en ese momento con un usuario determinado.

Catalog

Es la base de datos de contenidos de la aplicación. Contiene una relación de la caracterización de todos los contenidos que existen,

han existido o existirán en el sistema, para lo cual utiliza una base de datos Redis. Cada contenido tiene, entre otras cosas: - Información que relaciona el identificador del contenido (contentId) con el identificador del stream multimedia correspondiente en el sistema externo de streaming. - Una lista de palabras clave, tags, que permiten agrupar los contenidos según distintas características. - Un estado, de manera que los contenidos puedan ser introducidos en el sistema sin que eso los haga accesibles a los usuarios. - Un tag especial que asigna el contenido a una "categoría de usuario", de manera que se pueda filtrar qué contenido es accesible para cada tipo de usuario

Responsabilidad

El microservicio permite realizar búsquedas sobre la base de datos de contenido utilizando como criterio cualquiera de sus elementos característicos.

Salidas

Presenta un interface REST que permite la realización de queries y de comandos.

Categories

Mantiene una relación de las posibles "Categorías" a las que puede pertenecer un determinado usuario, por ejemplo Oro,Plata,Bronce ... Estas categorías permiten definir qué contenidos serán accesibles para el usuario. También dispone de una relación de "Paquetes" que puede contratar un usuario y que le darán acceso a nuevos contenidos definidos por un grupo de etiquetas. Todo usuario tiene que pertenecer a una categoría. Un usuario puede tener cualquier número de paquetes contratados, incluido ninguno.

Responsabilidad

La responsabilidad de este microservicio es mantener la información de qué paquetes tiene contratado un usuario dentro de los disponibles, y a qué categoría pertenece dicho usuario.

Permite filtrar una lista de contenidos para un usuario determinado, teniendo en cuenta la información sobre categoría y paquete que tiene de él.

Recommendations.

Mantiene una relación de los distintos tipos de contenidos visualizados por un usuario determinado de manera ponderada, es decir de los tags de los contenidos que visualiza cada usuario asignándoles un peso. También contiene dicha relación para todo el sistema. Para ambas utiliza una base de datos Redis .

Responsabilidad

Permite obtener información sobre los tipos de contenidos que más visualiza un usuario determinado, así como aquellos mas visualizados en el sistema. Con estos datos pueden realizarse otro tipo de operaciones tanto de presentación de nuevos contenidos al usuario en función de lo que suele ver, como de gestión de contenidos, añadir nuevos contenidos de aquellos tags que mas suelen verse en el sistema.

Entradas

La información la obtiene de las notificaciones que envía el microservicio StreamControl, para ello utiliza RabbitMq.

Salidas

Presenta un interface REST donde permite la realización de queries para usuarios o para todo el sistema.

DynamicData.

Existen datos que son necesarios mantener mientras los usuarios están utilizando el sistema, pero que no tiene sentido mantener en una persistencia. Este microservicio se encarga de mantener dichos datos.

Responsabilidad

En el caso que nos ocupa, un mismo usuario puede tener al mismo tiempo más de un dispositivo funcionando, y no tienen porque ser siempre los mismos dispositivos. Este microservicio se encarga de mantener la información sobre que dispositivos pertenecen a que usuario. Se basa en la suposición de que el cliente enviará el identificador de dispositivo, cuando este realizando operaciones.

Entradas y Salidas

Presenta un interface REST que permite realizar queries sobre la información de los usuarios y de los dispositivos activos.

ApiGw.

Es la fachada de entrada al sistema. Todas las interacciones de los usuarios pasan por este microservicio quien después distribuye las peticiones necesarias a los demás microservicios para satisfacer los comandos de los usuarios.

Responsabilidad.

Al ser el frontal de entrada a la aplicación es el encargado de autenticar y autorizar a los usuarios. Se utiliza para ello Json Web Token y una base de datos Redis con la información de credenciales de los usuarios. Evidentemente esta no es una solución utilizable en una instalación real por tener una seguridad muy deficiente, pero para el trabajo que nos ocupa y con los recursos disponibles era la mejor opción.

3. Comparativa de características.

Además de la comparación de las características de memoria y velocidad de carga de la aplicación, otro de los puntos importantes a dirimir con este trabajo es si el framework Micronaut resulta, a día de hoy, una alternativa viable para el desarrollo de aplicaciones.

Realizar una comparativa semejante no es sencillo. Después de haber trabajado durante horas con ambos frameworks desarrollando en ambos la misma aplicación, se tiene una serie de sensaciones sobre la facilidad y los inconvenientes de cada uno de los frameworks, pero resulta complicado diseccionar estas sensaciones y apuntalarlas con datos.

3.1. Documentación.

Uno de los pilares para poder trabajar con facilidad con frameworks tan complejos como estos, en términos de cantidad de "magia" que tienen entre bambalinas, es la documentación.

Una buena documentación debería:

- Estar actualizada

- Estar organizada siguiendo las mismas versiones que la propia aplicación.

- Tener un punto único de entrada.

- Ser completa en lo referente a ejemplos de utilización en diversas situaciones

Desgraciadamente ninguno de los dos frameworks a examen cumplen estas características, sin embargo cada uno las incumple de distinta manera.

Springboot, al ser un framework con una larga trayectoria, tiene una extensa documentación, demasiado extensa quizás.

El problema reside en la cantidad de fuentes que existen y en lo poco organizadas que estas están.

Buscar cómo realizar cualquier cosa para este framework es un perfecto ejemplo de "infoxicación". Lo más probable es que

encontremos al menos cinco maneras distintas de solucionarlo, pero sin referencias a cual de ellas podría ser mejor, o para que versiones de Springboot fueron propuestas estas soluciones.

Todo esto convierte en un proceso largo, tedioso el encontrar la manera que tiene el framework de ofrecer solución a un nuevo problema, y la mayoría de las veces termina en frustración, al no tener la seguridad de haber encontrado la mejor manera de solucionarlo.

Micronaut por otro lado, al ser un proyecto bastante joven, carece de una gran cantidad de documentación, aunque la que tiene está bien organizada, bien versionada, escrita de manera sencilla y con ejemplos.

Sin embargo el problema llega por esa "sencillez".

La documentación, en la mayoría de las ocasiones, solo cubre los casos más sencillos, de manera que, en cuanto queremos hacer algo que sea un tanto más complejo, nos encontramos en tierra de nadie y solos ante el peligro.

La aplicación que ha servido para ilustrar este trabajo no es tan complicada como lo podría ser una aplicación profesional real, sin embargo se han dado múltiples situaciones a lo largo del desarrollo, en los que la documentación no cubría en absoluto los casos de uso, teniendo en esos casos que recurrir a la investigación del código del framework.

En definitiva, ninguno de los dos frameworks brilla en el apartado de documentación. Con Springboot resulta más probable encontrar una solución al problema, pero es difícil decir si esta es la más adecuada, y es muy probable que no esté bien explicada, lo que acerca muchas veces a la sensación de estar programando con piezas de Lego.

Por otro lado Micronaut, al cubrir solo los casos más sencillos, deja muchas veces al desarrollador en una situación complicada, en la que solo puede buscar por si mismo si es posible realizar lo que necesita mirando el código del framework, o buscar un camino alternativo. Esto puede achacarse a la juventud del proyecto, por lo que si continúan manteniendo la documentación bien organizada y la amplían adecuadamente puede que en un futuro esto se subsane.

3.2. Comunidad

Otro de los salvavidas cuando se buscan soluciones en el proceso de desarrollo es preguntar a la comunidad existente de desarrolladores.

En este caso nos encontramos en una situación parecida a la comentada en el apartado de documentacion: La comunidad de usuarios de SpringBoot es mayor y por tanto existe mejores posibilidades de obtener ayuda, mientras que la base de usuarios de Micronaut es aún limitada.

Una pequeña ventaja a comentar con respecto a Micronaut es el establecimiento de un canal de comunicación, mediante la herramienta "gitter", donde poder plantear las preguntas directamente a los desarrolladores principales del framework así como a otros usuarios. Es una buena iniciativa que además provee de un punto de entrada para las preguntas, sin embargo por ahora existen muchas mas personas planteando preguntas que personas resolviéndolas, por lo que por ahora no es una herramienta útil.

3.3. Soporte para 3pp.

En un entorno de ejecución distribuido como es el cloud se plantean muchos nuevos retos que no existían en los entornos tradicionales cuando toda la aplicación consistía en un único monolito corriendo en un servidor localizado.

Para dar solución a estos nuevos problemas de una manera sencilla y uniforme, de manera que los desarrolladores puedan centrarse en el desarrollo de la "lógica de negocio" de su aplicación, que es lo que realmente aporta valor, se han adoptado muchos elementos software de terceros, de manera que una parte importante del desarrollo de una aplicación consiste en integrar esta con los elementos proveedores de dichas soluciones.

Para que esto sea más sencillo el framework que utilicemos tiene que proporcionarnos una manera fácil y directa de integrarlos, a la vez que permite que tengamos el control sobre como deseamos que estos elementos se comporten dentro de sus especificaciones.

SpringBoot ofrece una integración con casi cualquiera de los elementos software 3PP más utilizados actualmente en el desarrollo de aplicaciones para el cloud. Además el soporte ofrecido es, en su

mayor parte, completo, pudiendo trabajar en casi cualquier configuración que el elemento permita.

Micronaut por otro lado tiene muchas más limitaciones a la hora de ofrecer soporte para elementos de terceros, y en muchos de los caso el soporte que ofrece es básico y limitado solo a un tipo de operaciones o configuraciones de dicho elemento.

Como contrapartida decir que, en los casos en los que Micronaut si ofrece soporte para un elemento de terceros, la configuración de este suele ser menos verbosa y mas directa que su contrapartida en Springboot.

En el caso de la aplicación que ilustra este trabajo, la elección de los elementos de terceros estuvo siempre dirigida por las limitaciones de Micronaut, ya que desde un principio se vio que este framework daba soporte a menos elementos.

Como ejemplo mencionar que, en la fecha de desarrollo de este trabajo, Micronaut no ofrece soporte para la carga de configuración mediante los "configmaps" de kubernetes, y solo ofrece un soporte limitado al servicio de "service discovery" de dicha plataforma, por lo que, a pesar de desarrollar una aplicación para una plataforma que dispone de semejantes servicios de manera nativa, hubo de utilizarse Consul tanto para "service discovery" como para la carga de configuración distribuida.

3.4. Soporte para cloud.

Micronaut dice en su página web oficial ser "NATIVELY CLOUD NATIVE", intentando así distinguirse de los frameworks como SpringBoot que nacieron antes de la existencia del cloud y que se han ido adaptando a él.

La realidad es que, a parte del soporte para 3PP comentado anteriormente, si que hay ciertos elementos nativos en el framework que parecen ciertamente orientados a la creación y ejecución de aplicaciones en el cloud como podría ser: cliente http declarativo, anotaciones nativas para "overload protection" y "circuti breaking", etc. Sin embargo todo esto es conseguible en SpringBoot, por lo que tampoco resulta en un elemento decisivo a favor de Micronaut

3.5. Facilidad para el desarrollo/testing.

A parte de los datos que se han ido aportando en los apartados anteriores, existe también una sensación subjetiva de dificultad que se da a la hora de desarrollar una aplicación en un framework determinado. En el caso de la aplicación que ilustra este trabajo la sensaciones fueron que, siempre que el soporte de Micronaut no se interponía, el desarrollo era más sencillo y menos verboso que su contrapartida en Springboot, es decir hay que implementar menos código para obtener el mismo resultado.

Para ilustrar esto se puede ver como ejemplo el código necesario para configurar una conexión a Redis en ambos frameworks:

- En **Micronaut** no es necesario añadir código alguno, pues en el momento en el que encuentra en la configuración una conexión con la base de datos genera todo lo necesario.
- En **Springboot** la configuración tiene que realizarse por parte del desarrollador. Ejemplo:

```
@Configuration
public class RedisConfig {

  @Bean
  public RedisStandaloneConfiguration redisStandaloneConfiguration() {
    RedisStandaloneConfiguration redisStandaloneConfiguration = new RedisStandaloneConfiguration();

    redisStandaloneConfiguration.setHostName("recommendations-db");
    redisStandaloneConfiguration.setPort(6379);

    redisStandaloneConfiguration.setPassword(RedisPassword.of("recommendations-passwd"));
    return redisStandaloneConfiguration;
  }

  @Bean
  public LettuceConnectionFactory jedisConnectionFactory(RedisStandaloneConfiguration configuration) {
    return new LettuceConnectionFactory(configuration);

  }


  @Bean
  @ConditionalOnMissingBean(name = "redisTemplate")
  @Primary
  public RedisTemplate<String,Object> redisTemplate(RedisConnectionFactory factory) {

    final RedisTemplate<String,Object> template = new RedisTemplate<>();
    template.setConnectionFactory(factory);
    template.setValueSerializer(new GenericJackson2JsonRedisSerializer());
    return template;
  }
}
```

Illustration 6: SpringBoot Redis Configuration

Como contrapartida cuando estos mecanismos por defecto no se ajustan a las necesidades de la aplicación a desarrollar, los mecanismos de control para cambiar el comportamiento son en ocasiones insuficientes y/o complicados de implementar.

Como ejemplo ilustrativo de este hecho tenemos el manejo del servidor embebido de Redis para testing.

- **Springboot** relega en el desarrollador la tarea de lanzarlo cuando considere oportuno, cosa bastante sencilla aunque requiere cierto código.

```
private static int redisEmbeddedServerPort = 6379;

private RedisServer embeddedRedis = new RedisServer(redisEmbeddedServerPort);


@Autowired
private IRecommendationsRepository recommendationsRepository;

private static final Logger LOG = LoggerFactory.getLogger(ModelTest.class);

@Before
public void setup() {
  embeddedRedis.start();
  String redisContainerIpAddress = "localhost";
  int redisFirstMappedPort = redisEmbeddedServerPort;
  LOG.info("-=* Redis Container running on: " + redisContainerIpAddress + ":" + redisFirstMappedPort);
}

@After
public void tearDown() {
  embeddedRedis.stop();
}
```

Illustration 7: Springboot Embedded Redis Server

- **Micronaut** por otro lado lo hace de manera automática. Si detecta que se esta ejecutando en modo test, que el servidor embebido está configurado en el proyecto y que no hay ninguna conexión a base de datos creada para una conexión configurada, lanza el servidor embebido. Esto, que en un principio parece más cómodo tiene varios problemas:
  - No es posible controlar el comportamiento del servidor de manera manual, pararlo o arrancarlo.
  - Este comportamiento no está claramente documentado.

Con todo esto, la solución sería reimplementar y substituir la clase del framework que controla este servidor embebido y su comportamiento. Si bien el hecho de substituir la clase resulta sencillo, ya que el propio framework aporta anotaciones que permiten hacerlo, encontrar la clase que hay que substituir dentro del framework, no resulta sencillo por la falta de documentación al respecto.

```
@Requires(classes = RedisServer.class)
@Requires(beans = AbstractRedisConfiguration.class)
@Replaces(io.micronaut.configuration.lettuce.test.EmbeddedRedisServer.class)
@Requires(env = "test")
@Factory
public class EmbeddedRedisServer implements BeanCreatedEventListener<AbstractRedisConfiguration>, Closeable {
```

Illustration 8: Micronaut: Replacing framework class...

```
public void start() {
  if (redisServer != null) {
    if (!redisServer.isActive()) {
      redisServer.start();
    }
  }
}

public void stop() {
  if (redisServer != null) {
    redisServer.stop();
  }
}
```

Illustration 9: ... just to add some control code.

3.6. Resumen

En definitiva, la sensación general es que el framework Micronaut aún presenta muchas carencias como para ser considerado una alternativa seria a SpringBoot, sin embargo, si eliminamos aquellas fallas achacables a su falta de madurez, la situación mejora considerablemente, por lo que, si continúan realizando un buen trabajo, podría cambiar este balance en un futuro cercano

4. GraalVm

4.1. ¿Qué es GraalVM?

Hasta ahora HotStpotJVM era la principal máquina virtual de java mantenida por Oracle. Esta máquina virtual se encarga, principalmente, de interpretar y ejecutar código en "java bytecode" (.class), y analiza continuamente el código en ejecución para identificar aquellas partes que se ejecutan muy a menudo y compilarlas a código nativo para mejorar así el performance. En la siguiente figura podemos ver un workflow simplificado de como funciona esta "java VM"

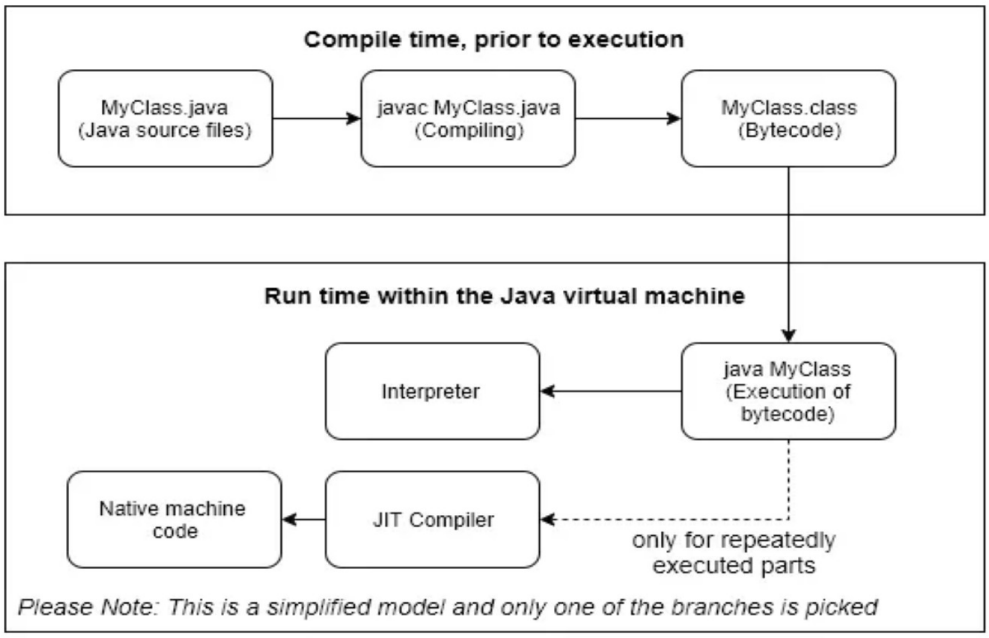

Illustration 10: HotSpotJVM: Workflow

La conversión del código a nativo se realiza en "runtime" de manera que no es viable hacerlo con todo el código pues esto impactaría a la ejecución de la aplicación. La JVM decide que elementos son compilados. El problema de esta aproximación es que, el JIT puede tardar mucho en "calentarse" y puede que determinadas partes del código nunca se lleguen a compilar a nativo.

Con la aparición del Java 9 se creó un "ahead-of-time compiler" en el JDK. A este proyecto se le dio el nombre de Graal. Este AOT-compiler permitía la generación de código nativo antes de la ejecución,

haciendo innecesarios los JIT. Esto derivó en una mejora del tiempo de ejecución, ya que el JIT no tenía que interceptar la ejecución de la aplicación en ningún momento. En la siguiente figura podemos ver un workflow simplificado del funcionamiento del código compilado mediante este AOT compiler.

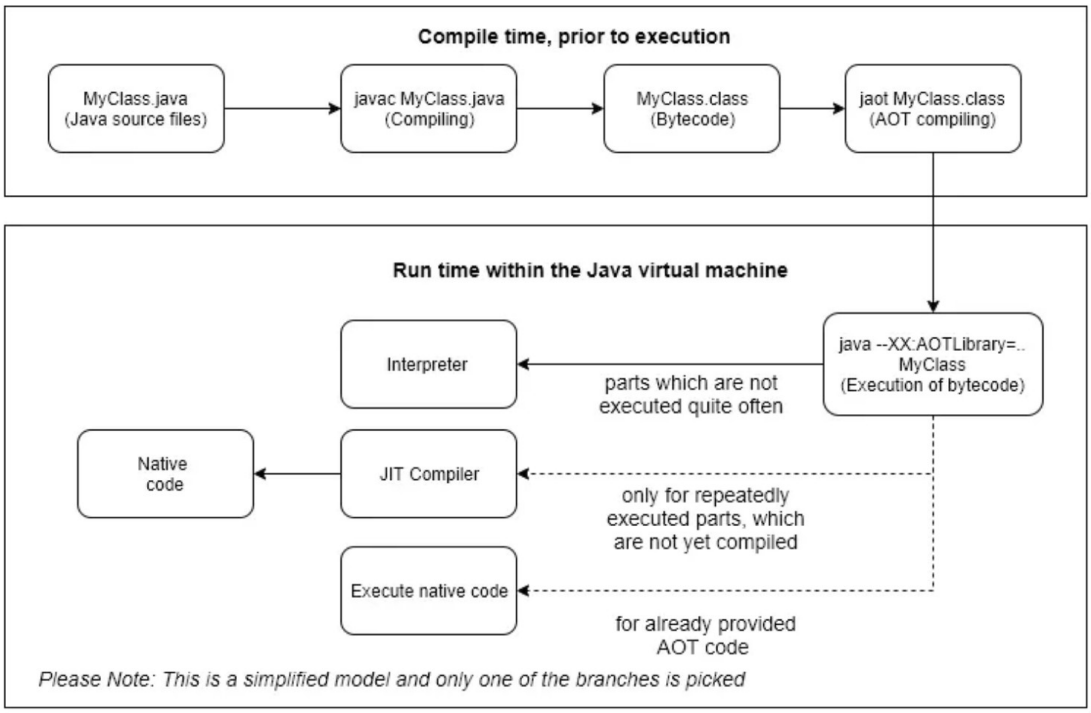

Illustration 11: Ahead of Time Compilation

GraalVM es un proyecto cuyo objetivo principal es mejorar la performance de los lenguajes interpretados en la JVM hasta acercarlo a los lenguajes compilados a binario nativo.

Incluye los componentes normales de una java virtual machine, como puede ser el JDK, y componentes adicionales que le permiten construir una “native-image”.
Uno de esos componentes es el Graal compiler, un “ahead of time compiler” que substituye a los actualmente utilizados.

Native Image

Es el resultado de utilizar un ahead-of-time-compiler, y produce binarios nativos de los ficheros .class.

Este ejecutable contiene todas las dependencias necesarias, de manera que no corre sobre una java virtual machine y además el tiempo de arranque se ve mejorado.

4.2. Compilación nativa usando GraalVM

La documentación de Micronaut sobre como compilar una aplicación para GraalVM es más bien sucinta. En este apartado de su documentación hay una pequeña guía que ilustra como "preparar" una sencilla aplicación para ser compilada usando la GraalVM. Desgraciadamente esta documentación, como casi toda la de este framework, peca de exceso de sencillez, por lo que no sirve más que para aplicaciones muy sencillas.

Si bien Micronaut dice ser compatible con GraalVM, deja bien claro que cualquiera de los elementos 3PP que se utilicen deben ser compatible con esta tecnología por su propia cuenta. Como resulta casi imposible no utilizar elementos de terceros para casi cualquier proyecto en el cloud que sea mas complejo que un simple "hola mundo", y la compatibilidad con la máquina GraalVM no es algo demasiado extendido por el momento, resulta complicado conseguir generar una máquina nativa de cualquier microservicio.

En el caso que nos ocupa resultó imposible realizar una imagen nativa, sin ningún error, de los microservicios utilizados, teniendo que recurrir a la "fallback image" para poder tomar algunas lecturas de datos.

El error que se producía daba a entender algún problema no relacionado directamente con la aplicación, por lo que, después de acudir al soporte de Micronaut, se abrió un ticket en GraalVM

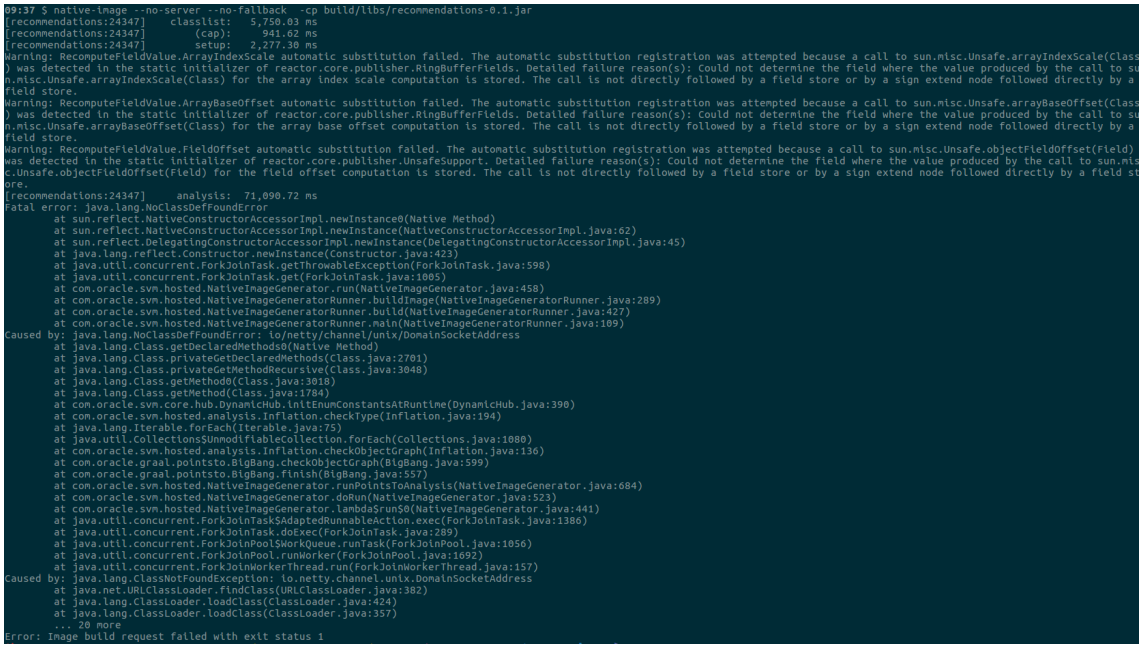

Illustration 12: Error while creating GraalVM

5. Comparativa en performance.

5.1. Memoria

En las siguientes gráficas podemos ver las diferencias de consumo de memoria de la máquina virtual de java en las distintas plataformas.

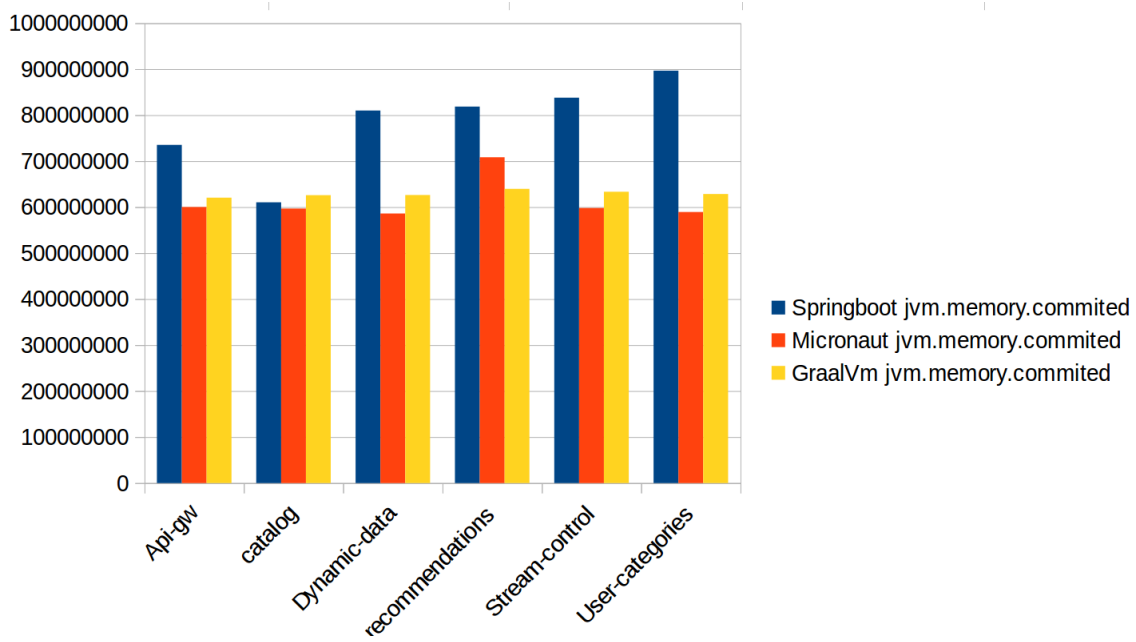

Illustration 13: Performance: Memory committed

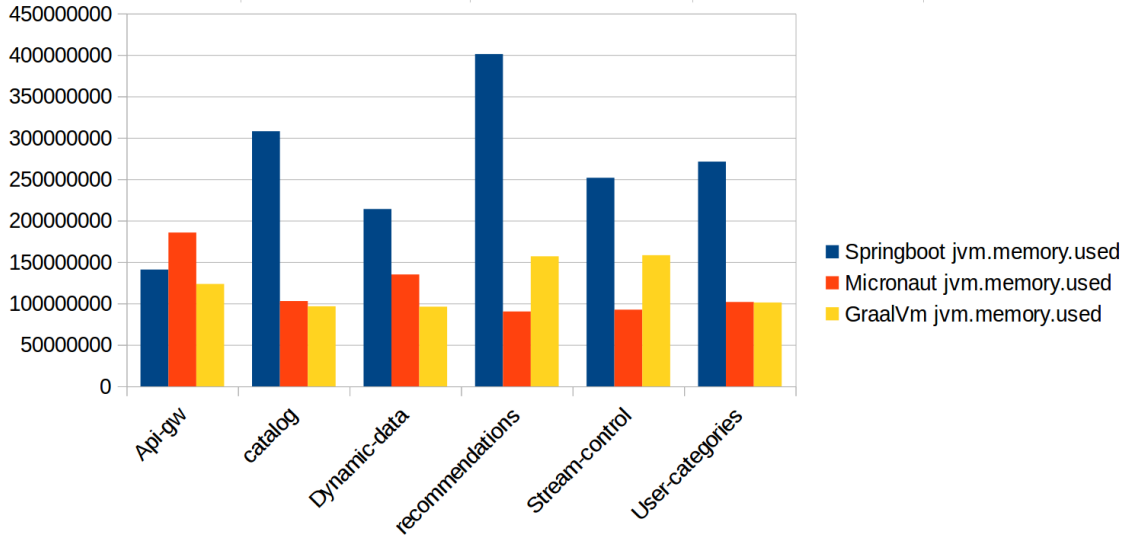

Illustration 14: Performance: Memory Used

Como se puede ver claramente en las gráficas, si exceptuamos el curioso caso del api-gw, las implementaciones de todos los microservicios hechas utilizando Springboot consumen sensiblemente más memoria que su contrapartida realizada utilizando Micronaut, llegando en ocasiones a una relación de 4:1.

Mención a parte merecen las medidas de memoria tomadas utilizando las imágenes nativas de GraalVM. En estos casos, el consumo de memoria no parece significativamente inferior, como debería ser, pero esto puede ser debido al hecho, comentado con anterioridad, de no haber podido obtener una imagen "limpia" de ninguno de los microservicios, por lo que estas medidas deberían de ser tomadas con cierto escepticismo.

A la vista de esta información podemos comprobar que las afirmaciones de que el framework Micronaut realiza un mejor uso de los recursos podrían estar fundados.

5.2. Velocidad de Arranque.

En la siguiente gráfica se puede ver los tiempos de arranque de los distintos microservicios en las diferentes plataformas.

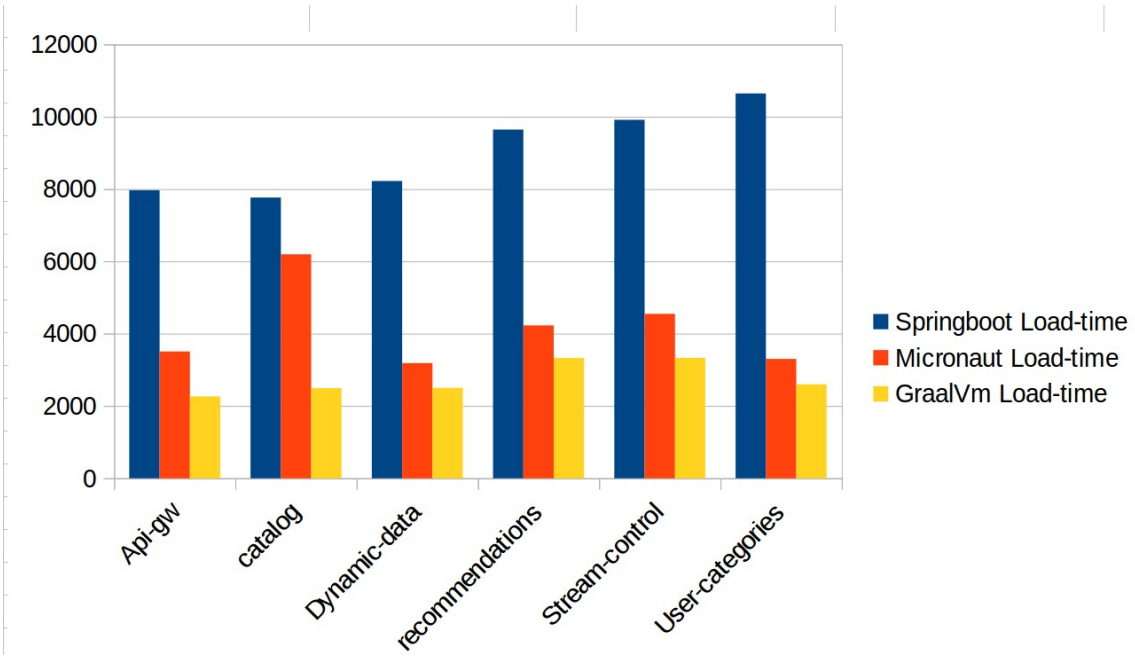

Illustration 15: Performance: Load time

En estas gráficas salta a la vista que el tiempo de arranque de las implementaciones de los microservicios realizadas con Springboot es netamente superior a las de las implementaciones realizadas con Micronaut, si bien en este caso la relación no parece ser tan grande como lo era en la gráfica de memoria.

De nuevo, las medidas realizadas para las imágenes nativas de GraalVM, si bien en este caso parecen más coherentes con lo que deberían ser, han de ser tomadas con cierto escepticismo por las razones ya mencionadas.

6. Conclusiones y siguientes pasos.

El principal objetivo de este trabajo consistía en tomar el pulso al nuevo framework de desarrollo de aplicaciones Micronaut y comprender si este era, a día de hoy una alternativa razonable al establecido Springboot, y si presentaba mejoras sobre a la hora de realizar aplicaciones basadas en el cloud, este por haber sido concebido de manera nativa para este tipo de aplicaciones.

¿Es una alternativa actualmente? ¿Y en un futuro próximo?

Aunando toda la información obtenida a lo largo del tiempo de desarrollo de este proyecto la sensación es que Micronaut aún no es una alternativa viable salvo para proyectos muy sencillos que solo requieran una integración básica con algunas de las herramientas para las que este framework presenta soporte, ya que en cuanto se intenta realizar una utilización un poco mas exigente de alguna de ellas, el framework comienza a presentar vacíos y fallas complicadas de subsanar.

Ahora bien, este framework presenta una buena utilización de recursos, una documentación ordenada, versionada y localizada en un solo punto y una "facilidad de uso" para el desarrollador bastante interesantes, por lo que, si el bajo número de desarrolladores activos no lo impide, podría ser una buena alternativa en un tiempo razonablemente corto.

¿Existen otras posibilidades a tener en cuenta?

Quarkus es otro framework de similares características a Micronaut y que se presenta a sí mismo como una alternativa rápida, tanto en ejecución como en desarrollo, a Springboot. A pesar de no haber podido probar este framework, es un proyecto auspiciado por la empresa RedHat por lo que seguramente deba ser tomado en cuenta a la hora de decidirse por un framework de desarrollo.

¿Cómo esta afrontando esto SpringBoot?

Las aplicaciones Spring, frawework en el que se basa Springboot, no permitían crear una imagen nativa utilizando GraalVM, sin embargo Spring está incorporando soporte experimental para esto es sus nuevas versiones, por lo que seguramente en sus próximas versiones esto sea una realidad y no sea necesario utilizar uno de estos nuevos frameworks para poder tener las ventajas que ofrece GraalVM. ¿Qué

puede significar esto para los frameworks como Micronaut y Quarkus? Pues seguramente una pérdida de inercia en su adopción, ya que una de los ganchos que podrían hacer que fuesen considerados como plataforma para nuevos desarrollos por encima de Springboot podría disiparse.

7. Enlaces y contenido del proyecto.

Todo el código desarrollado para este trabajo se encuentra en el repositorio https://github.com/jlojosnegros/master_upm_tfm.

En dicho repositorio hay tres directorios, cada uno de ellos con la implementación de los microservicios en una de las plataformas.

8. Bibliografía

- Spring Framework: https://spring.io/projects/spring-framework
- SpringBoot: https://spring.io/projects/spring-boot
- Micronaut:
  - Homepage: https://micronaut.io/
  - Gitter support: https://gitter.im/micronautfw/
  - Guia: https://docs.micronaut.io/1.1.2/guide/index.html
- GraalVM: https://github.com/spring-projects/spring-framework/wiki/GraalVM-native-image-support
- Consul: https://www.consul.io/
- Kubernetes: https://kubernetes.io/
- Zipkin: https://zipkin.apache.org/
- Redis: https://redis.io/
- Spring support for GraalVM : https://github.com/spring-projects/spring-framework/wiki/GraalVM-native-image-support