



Microservicios con Spring



© JMA 2020. All rights reserved

INTRODUCCIÓN

© JMA 2020. All rights reserved

Introducción

- El término "Microservice Architecture" ha surgido en los últimos años (2011) para describir una forma particular de diseñar aplicaciones de software como conjuntos de servicios de implementación independiente. Si bien no existe una definición precisa de este estilo arquitectónico, existen ciertas características comunes en torno a la organización en torno a la capacidad empresarial, la implementación automatizada, la inteligencia en los puntos finales y el control descentralizado de lenguajes y datos. (Martin Fowler)
- El estilo arquitectónico de microservicio es un enfoque para desarrollar una aplicación única como un conjunto de pequeños servicios, cada uno ejecutándose en su propio proceso y comunicándose con mecanismos ligeros, a menudo una API de recursos HTTP.
- Estos servicios se basan en capacidades empresariales y se pueden desplegar de forma independiente mediante mecanismos de implementación totalmente automatizada.
- Hay un mínimo de administración centralizada de estos servicios, que puede escribirse en diferentes lenguajes de programación y usar diferentes tecnologías de almacenamiento de datos.

© JMA 2020. All rights reserved

Estilos de arquitectura

- **N Niveles:** es la arquitectura tradicional para aplicaciones empresariales.
- **Web-queue-worker:** solución puramente PaaS, la aplicación tiene un front-end web que controla las solicitudes HTTP y un trabajador back-end que realiza tareas de uso intensivo de la CPU u operaciones de larga duración. El front-end se comunica con el trabajador a través de una cola de mensajes asincrónicos.
- **Orientada a servicios (SOA):** término sobre utilizado pero, como denominador común, significa que se estructura descomponiéndola en varios servicios que se pueden clasificar en tipos diferentes, como subsistemas o niveles.
- **Microservicios:** en un sistema que requiere alta escalabilidad y alto rendimiento, la arquitectura de microservicios se descompone en muchos servicios pequeños e independientes.
- **Basadas en eventos:** usa un modelo de publicación-suscripción (pub-sub), en el que los productores publican eventos y los consumidores se suscriben a ellos. Los productores son independientes de los consumidores y estos, a su vez, son independientes entre sí.
- **Big Data:** permite dividir un conjunto de datos muy grande en fragmentos, realizando un procesamiento paralelo en todo el conjunto, con fines de análisis y creación de informes.
- **Big compute:** también denominada informática de alto rendimiento (HPC), realiza cálculos en paralelo en un gran número (miles) de núcleos.

© JMA 2020. All rights reserved

Antecedentes

- El estilo de microservicio surge como alternativa al estilo monolítico.
- Una aplicación monolítica esta construida como una sola unidad. Las aplicaciones empresariales a menudo están integradas en tres partes principales:
 - una interfaz de usuario del lado del cliente (que consta de páginas HTML y javascript que se ejecutan en un navegador en la máquina del usuario)
 - una base de datos (que consta de muchas tablas insertadas en una instancia de bases de datos común y generalmente relacional)
 - una aplicación del lado del servidor que manejará las solicitudes HTTP, ejecutará la lógica del dominio, recuperará y actualizará los datos de la base de datos, y seleccionará y completará las vistas HTML que se enviarán al navegador.
- Esta aplicación del lado del servidor es un monolito, un único ejecutable lógico. Cualquier cambio en el sistema implica crear e implementar una nueva versión de la aplicación del lado del servidor.
- Cuando las aplicaciones escalan y se vuelven muy grandes, una aplicación monolítica construida como una sola unidad presenta serios problemas.

© JMA 2020. All rights reserved

SOA

- El concepto de dividir una aplicación en partes discretas no es nuevo. La idea para microservicios se origina en el patrón SOA de diseño de arquitectura orientado a servicios más amplio. La arquitectura orientada a servicios (SOA) es un método de desarrollo de software que utiliza componentes de software llamados servicios para crear aplicaciones empresariales. Cada uno de estos servicios brinda una capacidad empresarial y, además, pueden comunicarse también con el resto de servicios mediante diferentes plataformas y lenguajes. Los desarrolladores usan SOA para reutilizar servicios en diferentes sistemas o combinar varios servicios independientes para realizar tareas complejas.
- Sin embargo, a diferencia de la arquitectura orientada a servicios, una arquitectura de microservicios (como su nombre indica) debe contener servicios que son explícitamente pequeños y ligeros y que son desplegables de forma independiente. Los objetivos son:
 - Poder utilizar diferentes tecnologías por cada servicio (Java EE, Node, ...)
 - Permitir que cada servicio tenga un ciclo de vida independiente, es decir, versión independiente del resto, inclusive equipos de desarrollo diferentes.
 - Al ser servicios sin dependencia entre sí (especialmente de sesión), poder ejecutar el mismo en varios puertos, colocando un balanceador delante.
 - Poder crear instancias en servidores de diferentes regiones, lo que permitirá crecer (tanto verticalmente como horizontal) sin necesidad de cambiar el código fuente.

© JMA 2020. All rights reserved

La nube

- La nube está cambiando la forma en que se diseñan y protegen las aplicaciones. En lugar de ser monolitos, las aplicaciones se descomponen en servicios menores y descentralizados. Estos servicios se comunican a través de APIs, mediante el uso de eventos o de mensajería asíncrona. Las aplicaciones se escalan horizontalmente, agregando nuevas instancias, tal y como exigen las necesidades.
- Estas tendencias agregan nuevos desafíos:
 - El estado de las aplicaciones se distribuye.
 - Las operaciones se realizan en paralelo y de forma asíncrona.
 - Las aplicaciones deben ser resistentes cuando se produzcan errores.
 - Las aplicaciones son continuamente atacadas por actores malintencionados.
 - Las implementaciones deben estar automatizadas y ser predecibles.
 - La supervisión y la telemetría son fundamentales para obtener una visión general del sistema.

© JMA 2020. All rights reserved

Cambio de paradigma

Local tradicional

- Monolítica
- Diseñada para una escalabilidad predecible
- Base de datos relacional
- Procesamiento síncrono
- Diseño para evitar errores (MTBF)
- Actualizaciones grandes, ocasionales
- Administración manual
- Servidores en copo de nieve

Nube moderna

- Descompuesto
- Diseñado para un escalado elástico
- Persistencia poliglota (combinación de tecnologías de almacenamiento)
- Procesamiento asíncrono
- Diseño resiliente a errores (MTTR)
- Pequeñas actualizaciones, frecuentes
- Administración automatizada
- Infraestructura inmutable

© JMA 2020. All rights reserved

Monolítico: Beneficios

- Simple de desarrollar: el objetivo de las herramientas de desarrollo e IDE actuales es apoyar el desarrollo de aplicaciones monolíticas.
- Fácil de implementar: simplemente necesita implementar el archivo WAR (o jerarquía de directorios) en el tiempo de ejecución adecuado
- Fácil de escalar: puede escalar la aplicación ejecutando varias copias de la aplicación detrás de un balanceador de carga

© JMA 2020. All rights reserved

Monolítico: Inconvenientes

- La gran base de código monolítico intimida a los desarrolladores, especialmente aquellos que son nuevos en el equipo. La aplicación puede ser difícil de entender y modificar. Como resultado, el desarrollo normalmente se ralentiza. Además, la modularidad se descompone con el tiempo. Además, debido a que puede ser difícil entender cómo implementar correctamente un cambio, la calidad del código disminuye con el tiempo. Es una espiral descendente.
- IDE sobrecargado: cuanto mayor sea la base del código, más lento será el IDE y los desarrolladores menos productivos.
- La implementación continua es difícil: una gran aplicación monolítica también es un obstáculo para las implementaciones frecuentes.
 - Para actualizar un componente, se debe volver a desplegar toda la aplicación. Esto interrumpirá los procesos en segundo plano, independientemente de si se ven afectados por el cambio y posiblemente causen problemas.
 - También existe la posibilidad de que los componentes que no se han actualizado no se inicien correctamente. Como resultado, aumenta el riesgo asociado con la redistribución, lo que desalienta las actualizaciones frecuentes. Esto es especialmente un problema para los desarrolladores de interfaces de usuario, ya que por lo general necesitan que sea iterativo y la redistribución rápida.

© JMA 2020. All rights reserved

Monolítico: Inconvenientes

- Contenedor web sobrecargado: cuanto más grande es la aplicación, más tarda en iniciarse. Esto tiene un gran impacto en la productividad del desarrollador debido a la pérdida de tiempo en la espera de que se inicie el contenedor. También afecta el despliegue.
- La ampliación de la aplicación puede ser difícil: solo puede escalar en una dimensión.
 - Por un lado, puede escalar con un volumen creciente de transacciones ejecutando más copias de la aplicación. Algunas nubes pueden incluso ajustar el número de instancias de forma dinámica según la carga. Pero, por otro lado, esta arquitectura no puede escalar con un volumen de datos en aumento. Cada copia de la instancia de la aplicación accederá a todos los datos, lo que hace que el almacenamiento en caché sea menos efectivo y aumenta el consumo de memoria y el tráfico de E/S. Además, los diferentes componentes de la aplicación tienen diferentes requisitos de recursos: uno puede hacer un uso intensivo de la CPU y otro puede requerir mucha memoria. Con una arquitectura monolítica no podemos escalar cada componente independientemente.

© JMA 2020. All rights reserved

Monolítico: Inconvenientes

- Obstáculo para el desarrollo escalar. Una vez que la aplicación alcanza un cierto tamaño, es útil dividir a los desarrolladores en equipos que se centran en áreas funcionales específicas. El problema es que impide que los equipos trabajen de forma independiente. Los equipos deben coordinar sus esfuerzos de desarrollo y despliegue. Es mucho más difícil para un equipo hacer un cambio y actualizar la producción.
- Requiere un compromiso a largo plazo con una pila de tecnología: obliga a casarse con una tecnología (y, en algunos casos, con una versión particular de esa tecnología) que se eligió al inicio del desarrollo, puede que hace mucho tiempo. Puede ser difícil adoptar de manera incremental una tecnología más nueva. No permite utilizar otros lenguajes o entornos de desarrollo. Además, si la aplicación utiliza una plataforma que posteriormente se vuelve obsoleta, puede ser un desafío migrar gradualmente la aplicación a un marco más nuevo y mejor.

© JMA 2020. All rights reserved

Microservicios: Beneficios

- Cada microservicio es relativamente pequeño.
 - Más fácil de entender para un desarrollador.
 - El IDE es más rápido haciendo que los desarrolladores sean más productivos.
 - La aplicación se inicia más rápido, lo que hace que los desarrolladores sean más productivos y acelera las implementaciones.
- Permite la entrega y el despliegue continuos de aplicaciones grandes y complejas.
 - Mejor capacidad de prueba: los servicios son más pequeños y más rápidos de probar
 - Mejor implementación: los servicios se pueden implementar de forma independiente
 - Permite organizar el esfuerzo de desarrollo alrededor de múltiples equipos autónomos. Cada equipo (dos pizzas) es propietario y es responsable de uno o más servicios individuales. Cada equipo puede desarrollar, implementar y escalar sus servicios independientemente de todos los otros equipos.
- Aislamiento de defectos mejorado. Por ejemplo, ante una pérdida de memoria en un servicio, solo ese servicio se verá afectado, los otros continuarán manejando las solicitudes. En comparación, un componente que se comporta mal en una arquitectura monolítica puede derribar todo el sistema.
- Elimina cualquier compromiso a largo plazo con una pila de tecnología. Al desarrollar un nuevo servicio, se puede elegir una nueva pila tecnológica. Del mismo modo, cuando realiza cambios importantes en un servicio existente, puede reescribirlo utilizando una nueva pila de tecnología.

© JMA 2020. All rights reserved

Microservicios: Inconvenientes

- Los desarrolladores deben lidiar con la complejidad adicional de crear un sistema distribuido.
 - Las herramientas de desarrollo / IDE están orientadas a crear aplicaciones monolíticas y no proporcionan soporte explícito para desarrollar aplicaciones distribuidas.
 - La prueba es más difícil, requiere un mayor peso en las pruebas de integración
 - Sobrecarga a los desarrolladores, deben implementar el mecanismo de comunicación entre servicios.
 - Implementar casos de uso que abarcan múltiples servicios sin usar transacciones distribuidas es difícil
 - La implementación de casos de uso que abarcan múltiples servicios requiere una coordinación cuidadosa entre los equipos
- La complejidad del despliegue. En producción, también existe la complejidad operativa de implementar y administrar un sistema que comprende muchos tipos componentes y servicios diferentes.
- Mayor consumo de recursos. La arquitectura de microservicio reemplaza n instancias de aplicaciones monolíticas con $n*m$ instancias de servicios. Si cada servicio se ejecuta en su propia JVM (o equivalente), que generalmente es necesario para aislar las instancias, entonces hay una sobrecarga de m veces más tiempo de ejecución de JVM. Además, si cada servicio se ejecuta en su propia VM, como es el caso en Netflix, la sobrecarga es aún mayor.

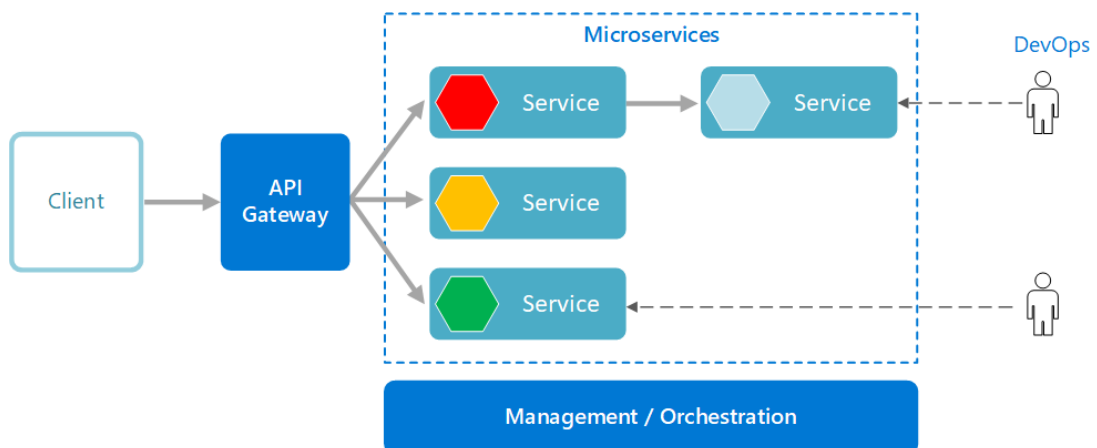
© JMA 2020. All rights reserved

Arquitectura de Microservicios (Lewis/Fowler)

- Componentización a través de Servicios
- Organización de equipos alrededor de Capacidades Empresariales
- Productos no Proyectos
- Gobernanza descentralizada
- Puntos finales inteligentes y conexiones tontas
- Gestión descentralizada de datos
- Automatización de Infraestructura
- Diseño tolerante a fallos
- Diseño Evolutivo

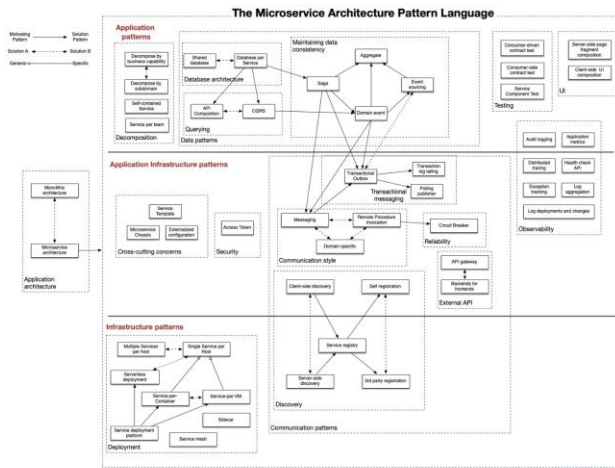
© JMA 2020. All rights reserved

Arquitectura de microservicios



© JMA 2020. All rights reserved

Patrones arquitectónicos y de diseño



- La arquitectura de microservicios no es una panacea. Tiene varios inconvenientes. Además, al utilizar esta arquitectura, existen numerosos problemas que debe abordar.
- El lenguaje de patrones de la arquitectura de microservicios es una colección de patrones para aplicar la arquitectura de microservicios. Tiene dos objetivos:
 - El lenguaje de patrones le permite decidir si los microservicios son adecuados para su aplicación.
 - El lenguaje de patrones le permite usar la arquitectura de microservicios con éxito.
- Referencias:
 - <https://microservices.io/>
 - <https://learn.microsoft.com/es-es/azure/architecture/patterns/>

De Monolito a Microservicios

- Patrones para descomponer monolitos
 - Descomposición por capacidad empresarial
 - Descomposición por subdominio
 - Descomposición por transacciones
 - Descomposición por equipo
- Patrones para migrar de monolito a microservicios
 - Reescritura Big Bang
 - Strangler Fig
 - Branch by Abstraction
 - Anti-corruption Layer
 - Ambassador

Preocupaciones transversales

- En referencia a las APIs, servicios y microservicios, la tendencia natural es a crecer, tanto por nuevas funcionalidades del sistema como por escalado horizontal.
- Todo ello provoca una serie de preocupaciones adicionales:
 - Localización de los servicios.
 - Balanceo de carga.
 - Tolerancia a fallos.
 - Gestión de la configuración.
 - Gestión de logs.
 - Gestión de los despliegues.
 - y otras ...

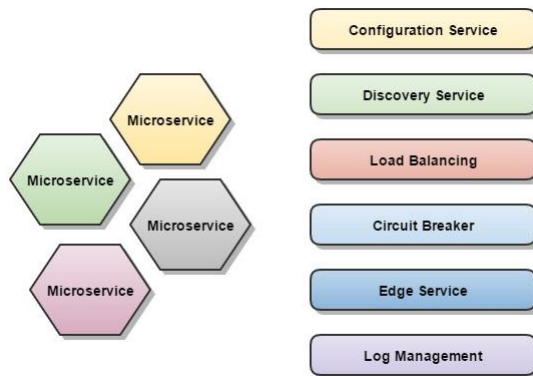
© JMA 2020. All rights reserved

Implantación

- Para la implantación de una arquitectura basada en APIs hemos tener en cuenta 3 aspectos principalmente:
 - Un modelo de referencia en el que definir las necesidades de una arquitectura de las APIs.
 - Un modelo de implementación en el que decidir y concretar la implementación de los componentes vistos en el modelo de referencia.
 - Un modelo de despliegue donde definir cómo se van a desplegar los distintos componentes de la arquitectura en los diferentes entornos.

© JMA 2020. All rights reserved

Modelo de referencia



© JMA 2020. All rights reserved

Modelo de referencia

- Servidor perimetral / exposición de servicios (Edge server)
 - Será un gateway en el que se expondrán los servicios a consumir.
- Servicio de registro / descubrimiento
 - Este servicio centralizado será el encargado de proveer los endpoints de los servicios para su consumo. Todo microservicio, en su proceso de arranque, se registrará automáticamente en él.
- Balanceo de carga (Load balancer)
 - Este patrón de implementación permite el balanceo entre distintas instancias de forma transparente a la hora de consumir un servicio.
- Tolerancia a fallos (Circuit breaker)
 - Mediante este patrón conseguiremos que cuando se produzca un fallo, este no se propague en cascada por todo el pipe de llamadas, y poder gestionar el error de forma controlada a nivel local del servicio donde se produjo.
- Mensajería:
 - Las invocaciones siempre serán síncronas (REST, SOAP, ...) o también llamadas asíncronas (AMQP).

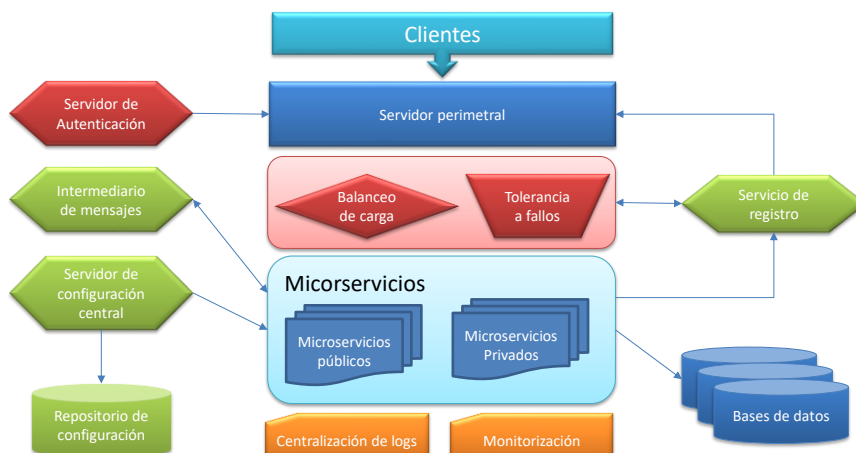
© JMA 2020. All rights reserved

Modelo de referencia

- **Servidor de configuración central**
 - Este componente se encargará de centralizar y proveer remotamente la configuración a cada API. Esta configuración se mantiene convencionalmente en un repositorio Git, lo que nos permitirá gestionar su propio ciclo de vida y versionado.
- **Servidor de autenticación / autorización**
 - Para implementar la capa de seguridad (recomendable en la capa de servicios API)
- **Centralización de logs**
 - Se hace necesario un mecanismo para centralizar la gestión de logs. Pues sería inviable la consulta de cada log individual de cada uno de los microservicios.
- **Monitorización**
 - Para poder disponer de mecanismos y dashboard para monitorizar aspectos de los nodos como, salud, carga de trabajo...

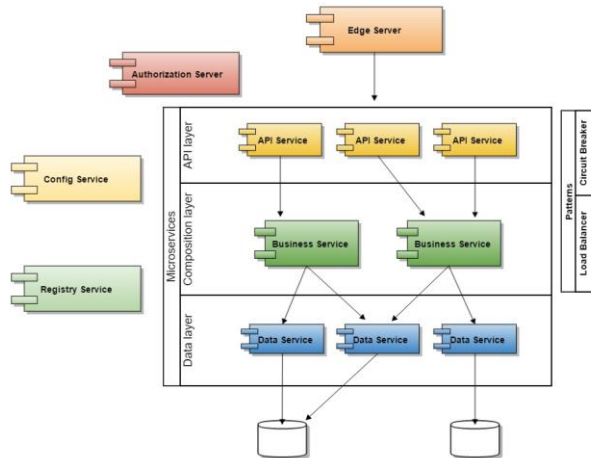
© JMA 2020. All rights reserved

Modelo de referencia



© JMA 2020. All rights reserved

Modelo de referencia



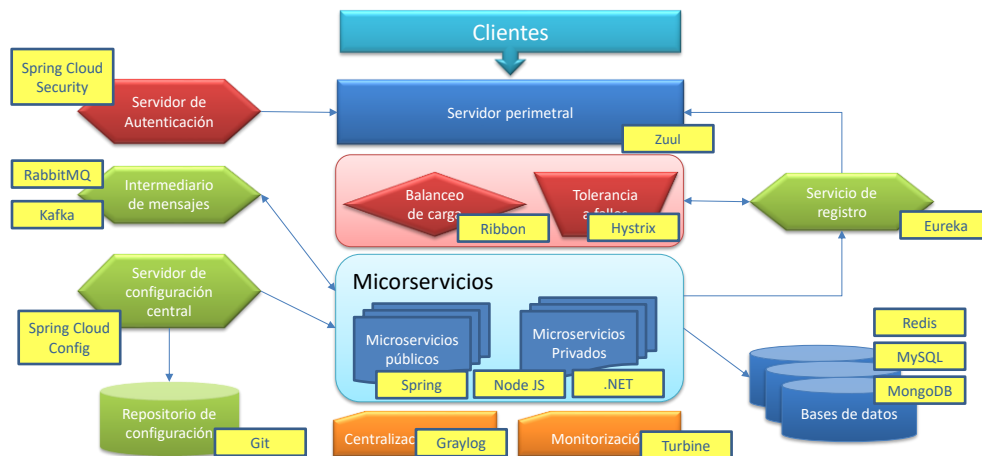
© JMA 2020. All rights reserved

Modelo de implementación (Netflix OSS)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Spring Cloud y Netflix OSS:
 - Microservicios propiamente dichos: Serán aplicaciones Spring Boot con controladores Spring MVC. Se puede utilizar Swagger para documentar y definir nuestra API.
 - Config Server: microservicio basado en Spring Cloud Config y se utilizará Git como repositorio de configuración.
 - Registry / Discovery Service: microservicio basado en Eureka de Netflix OSS.
 - Load Balancer: se puede utilizar Ribbon de Netflix OSS que ya viene integrado en REST-template de Spring.
 - Circuit breaker: se puede utilizar Hystrix de Netflix OSS.
 - Gestión de Logs: se puede utilizar Graylog
 - Servidor perimetral: se puede utilizar Zuul de Netflix OSS.
 - Servidor de autenticación / autorización: se puede utilizar el servicio con Spring Cloud Security.
 - Agregador de métricas: se puede utilizar el servicio Turbine.
 - Intermediario de mensajes: se puede utilizar AMQP con RabbitMQ o Kafka.

© JMA 2020. All rights reserved

Modelo de implementación (Netflix OSS)



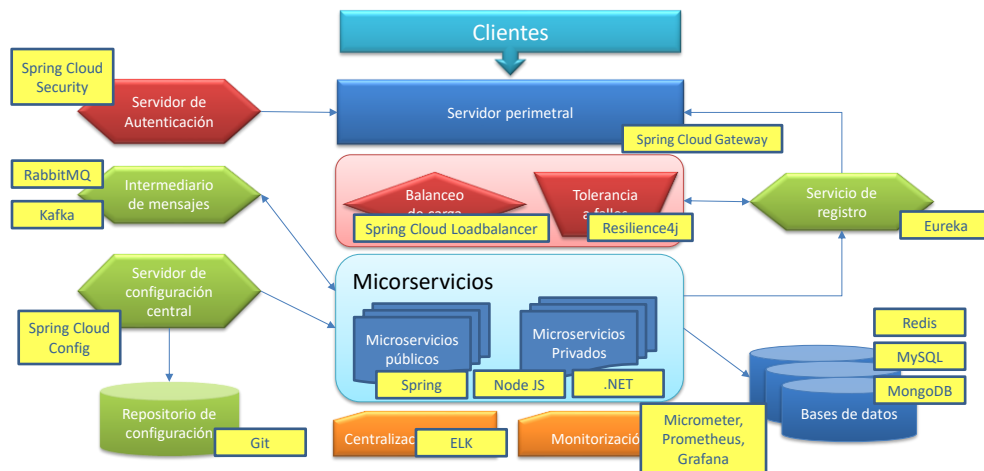
© JMA 2020. All rights reserved

Modelo de implementación (Spring Cloud)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Spring Cloud y Netflix OSS:
 - Microservicios propiamente dichos: Serán aplicaciones Spring Boot con controladores Spring MVC. Se puede utilizar Swagger para documentar y definir nuestra API.
 - Config Server: microservicio basado en Spring Cloud Config y se utilizará Git como repositorio de configuración.
 - Registry / Discovery Service: microservicio basado en Eureka de Netflix OSS.
 - Load Balancer: se puede utilizar Spring Cloud Loadbalancer que ya viene integrado en REST-template de Spring.
 - Circuit breaker: se puede utilizar Spring Cloud Circuit Breaker con Resilience4j.
 - Gestión de Logs: se puede utilizar Elasticsearch, Logstash y Kibana
 - Servidor perimetral: se puede utilizar Spring Cloud Gateway.
 - Servidor de autenticación / autorización: se puede utilizar el servicio con Spring Cloud Security.
 - Agregador de métricas: se puede utilizar el servicio Turbine.
 - Intermediario de mensajes: se puede utilizar AMQP con RabbitMQ o Kafka.

© JMA 2020. All rights reserved

Modelo de implementación (Spring Cloud)



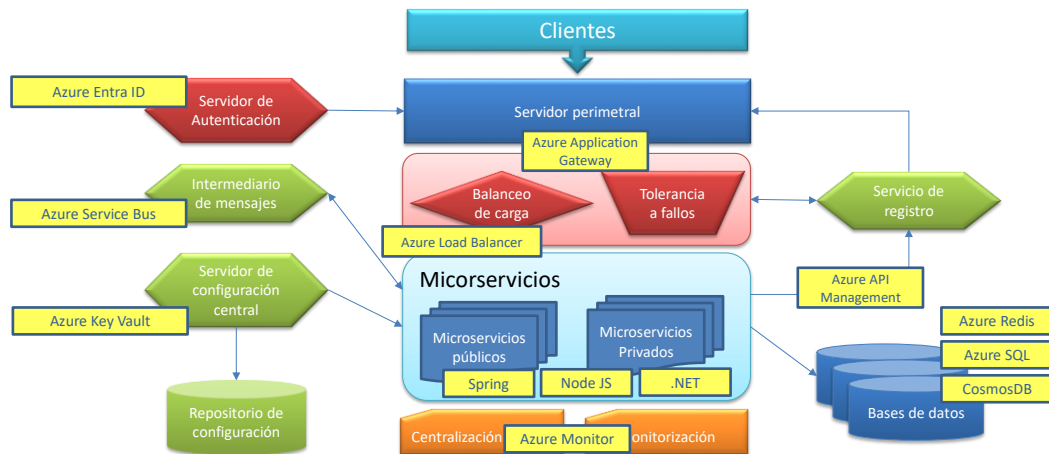
© JMA 2020. All rights reserved

Modelo de implementación (Azure)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de suministrado por Azure:
 - Microservicios propiamente dichos: Serán aplicaciones ASP.NET Core con WebApi. Se puede utilizar OpenAPI para documentar y definir nuestra API.
 - Azure Key Vault: Se puede utilizar para almacenar de forma segura y controlar de manera estricta el acceso a los tokens, contraseñas, certificados, claves de API y otros secretos.
 - Azure API Management: es una solución completa para publicar API para clientes externos e internos.
 - Servidor perimetral, Registry / Discovery Service, Load Balancer (con Azure Application Gateway), Circuit breaker.
 - Servidor de autorización: Azure Entra ID (Azure Active Directory) es un servicio de administración de identidades y acceso basado en la nube de Microsoft.
 - Azure Monitor: ayuda a maximizar la disponibilidad y el rendimiento de las aplicaciones y los servicios.
 - Agregador de métricas: Detección y diagnóstico de problemas en aplicaciones y dependencias con Application Insights.
 - Gestión de Logs: Profundización en sus datos de supervisión con Log Analytics para la solución de problemas y diagnósticos profundos.
 - Intermediario de mensajes: se puede utilizar AMQP con Azure Service Bus.

© JMA 2020. All rights reserved

Modelo de implementación (Azure)



© JMA 2020. All rights reserved

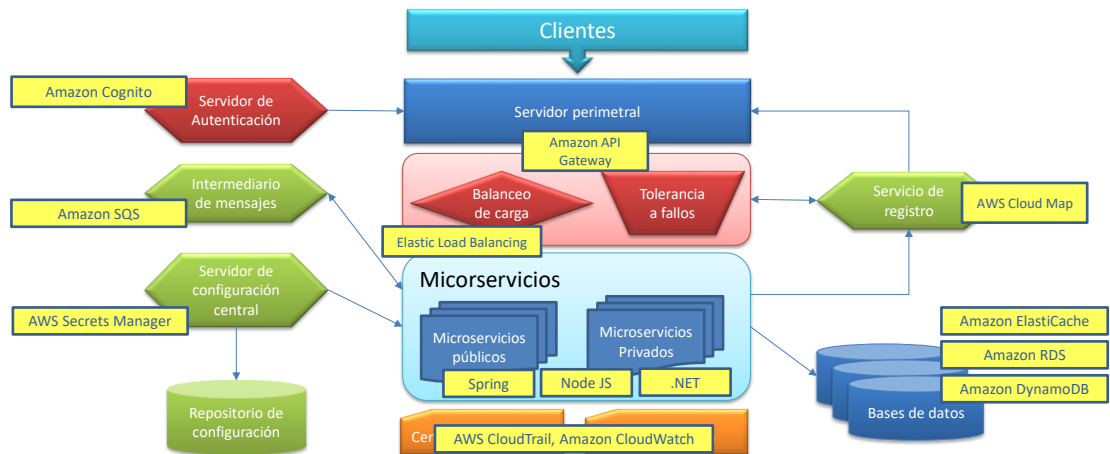
Modelo de implementación (AWS)

- Basándonos en el modelo de referencia, vamos a definir un modelo de implementación para cada uno de los componentes descritos. Para ello podemos hacer uso del stack tecnológico de Amazon Web Services:
 - Amazon API Gateway: Proxy de la API
 - Elastic Load Balancing: Balanceador de carga de aplicaciones
 - AWS Cloud Map: Detección de servicios
 - Amazon RDS: Bases de datos relacionales
 - Amazon DynamoDB: Bases de datos NoSQL
 - Amazon ElastiCache: Almacenamiento en caché
 - Amazon Simple Queue Service (Amazon SQS): Colas de mensajes
 - AWS CloudTrail, Amazon CloudWatch: Monitorización de API
 - Amazon Cognito, AWS IAM: Registro, inicio de sesión y control de acceso de usuarios
 - AWS Secrets Manager, AWS KMS: Datos confidenciales de configuración

<https://aws.amazon.com/es/microservices/>

© JMA 2020. All rights reserved

Modelo de implementación (AWS)



© JMA 2020. All rights reserved

Modelo de despliegue

- El modelo de despliegue hace referencia al modo en que vamos a organizar y gestionar los despliegues de los microservicios, así como a las tecnologías que podemos usar para tal fin.
- El despliegue de los microservicios es una parte primordial de esta arquitectura. Muchas de las ventajas que aportan, como la escalabilidad, son posibles gracias al sistema de despliegue.
- Existen convencionalmente varios patrones en este sentido a la hora de encapsular microservicios:
 - Máquinas virtuales.
 - Contenedores.
 - Sin servidor: FaaS (Functions-as-a-Service)
- Los microservicios están íntimamente ligados al concepto de contenedores (una especie de máquinas virtuales ligeras que corren de forma independiente, pero utilizando directamente los recursos del host en lugar de un SO completo). Hablar de contenedores es hablar de Docker. Con este software se pueden crear las imágenes de los contenedores para después crear instancias a demanda.

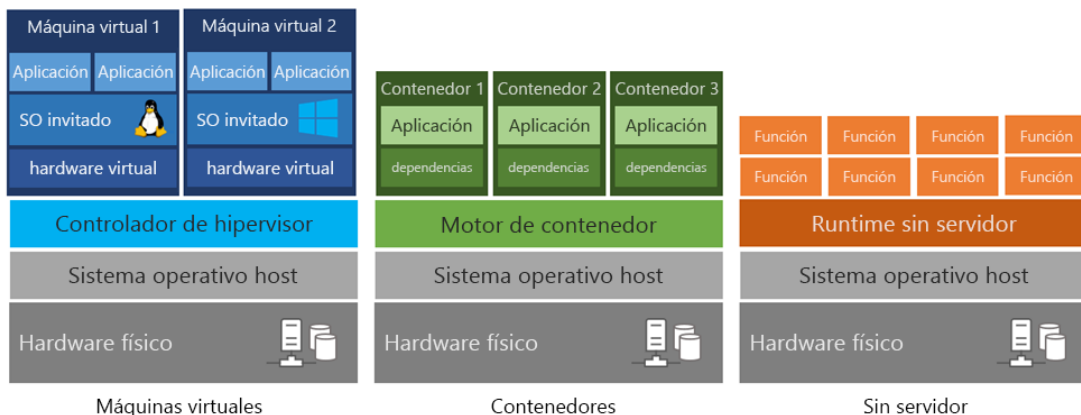
© JMA 2020. All rights reserved

Modelo de despliegue

- Las imágenes Docker son como plantillas. Constan de un conjunto de capas y cada una aporta un conjunto de software a lo anterior, hasta construir una imagen completa.
- Por ejemplo, podríamos tener una imagen con una capa Ubuntu y otra capa con un servidor LAMP. De esta forma tendríamos una imagen para ejecutar como servidor PHP.
- Las capas suelen ser bastante ligeras. La capa de Ubuntu, por ejemplo, contiene algunos los ficheros del SO y otros, como el Kernel, los toma del host.
- Los contenedores toman una imagen y la ejecutan, añadiendo una capa de lectura/escritura, ya que las imágenes son de sólo lectura.
- Dada su naturaleza volátil (el contenedor puede parar en cualquier momento y volver a arrancarse otra instancia), para el almacenamiento se usan volúmenes, que están fuera de los contenedores.

© JMA 2020. All rights reserved

Contenedores



© JMA 2020. All rights reserved

Modelo de despliegue

- Sin embargo, esto no es suficiente para dotar a nuestro sistema de una buena escalabilidad. El siguiente paso será pensar en la automatización y orquestación de los despliegues siguiendo el paradigma cloud. Se necesita una plataforma que gestione los contenedores, y para ello existen soluciones como Kubernetes.
- Kubernetes permite gestionar grandes cantidades de contenedores, agrupándolos en pods. También se encarga de gestionar servicios que estos necesitan, como conexiones de red y almacenamiento, entre otros. Además, proporciona también esta parte de despliegue automático, que puede utilizarse con sus componentes o con componentes de otras tecnologías como Spring Cloud+Netflix OSS.
- Todavía se puede dar una vuelta de tuerca más, incluyendo otra capa por encima de Docker y Kubernetes: Openshift. En este caso estamos hablando de un PaaS que, utilizando Docker y Kubernetes, realiza una gestión más completa y amigable de nuestro sistema de microservicios. Por ejemplo, nos evita interactuar con la interfaz CLI de Kubernetes y simplifica algunos procesos. Además, nos provee de más herramientas para una gestión más completa del ciclo de vida, como construcción, test y creación de imágenes. Incluye los despliegues automáticos como parte de sus servicios y, en sus últimas versiones, el escalado automático.
- Openshift también proporciona sus propios componentes, que de nuevo pueden mezclarse con los de otras tecnologías.

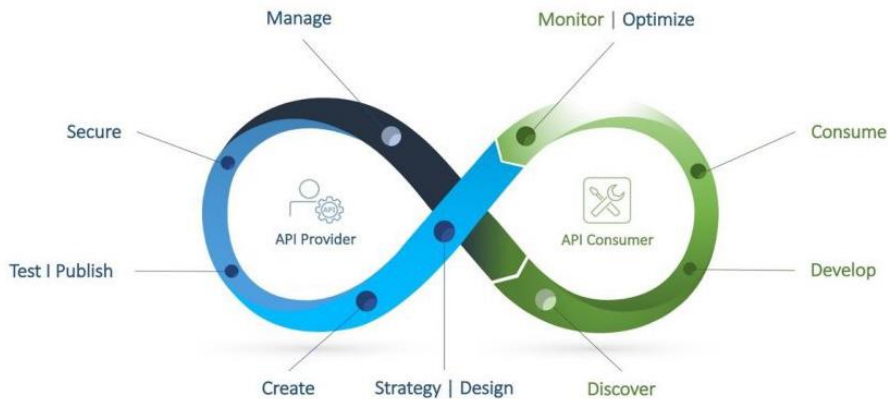
© JMA 2020. All rights reserved

FaaS (Functions-as-a-Service)

- El auge de la informática sin servidor es una de las innovaciones más importantes de la actualidad. Las tecnologías sin servidor, como Azure Functions, AWS Lambda o Google Cloud Functions, permiten a los desarrolladores centrarse por completo en escribir código. Toda la infraestructura informática de la que dependen (máquinas virtuales (VM), compatibilidad con la escalabilidad y demás) se administra por ellos. Debido a esto, la creación de aplicaciones se vuelve más rápida y sencilla. Ejecutar dichas aplicaciones a menudo resulta más barato, porque solo se le cobra por los recursos informáticos que realmente usa el código.
- La arquitectura serverless habilita la ejecución de una aplicación mediante contenedores efímeros y sin estado; estos son creados en el momento en el que se produce un evento que dispare dicha aplicación. Contrariamente a lo que nos sugiere el término, serverless no significa «sin servidor», sino que éstos se usan como un elemento anónimo más de la infraestructura, apoyándose en las ventajas del cloud computing.
- La tecnología sin servidor apareció por primera vez en lo que se conoce como tecnologías de plataforma de aplicaciones como servicio (aPaaS), actualmente como FaaS (Functions-as-a-Service).

© JMA 2020. All rights reserved

Ciclo de vida



© JMA 2020. All rights reserved

Ciclo de vida

- **Estrategia:** que camino se va a seguir y como se planifica
- **Creación:** una vez se tenga una estrategia y un plan sólidos, es hora de crear las APIs.
- **Pruebas:** antes de publicar, es importante completar las pruebas de API para garantizar que cumplan con las expectativas de rendimiento, funcionalidad y seguridad.
- **Publicación:** una vez probado, es hora de publicar la API para que estén disponibles para los desarrolladores.
- **Protección:** los riesgos y las preocupaciones de seguridad son un problema común en la actualidad.
- **Administración:** una vez publicadas, los creadores deben administrar y mantener las APIs para asegurarse de que estén actualizadas y que la integridad de sus APIs no se vea comprometida.
- **Integración:** cuando se ofrece las APIs para consumo público o privado, la documentación es un componente importante para que los desarrolladores comprendan las capacidades clave.
- **Monitorización:** una vez las APIs están activas, es necesario supervisarlas y analizar los datos para detectar anomalías o detectar nuevas necesidades.
- **Promoción:** hay varias formas de comercializar las APIs, incluida su inclusión en un mercado de APIs.
- **Monetización:** se puede optar por ofrecer las APIs de forma gratuita o, cuando existe la oportunidad, se puede monetizar las APIs y generar ingresos adicionales para el negocio.
- **Retirada:** Retirar las APIs es la última etapa del ciclo de vida de una API y ocurre por una variedad de razones, incluidos cambios tecnológicos y preocupaciones de seguridad.

© JMA 2020. All rights reserved

Ciclo de vida



© JMA 2020. All rights reserved

¿Qué es una API?

- API es el acrónimo de Application Programming Interface, que es un intermediario de software que permite que dos aplicaciones se comuniquen entre sí.
- A lo largo de los años, lo que es una API a menudo se ha descrito como cualquier tipo de interfaz de conectividad genérica para una aplicación. Más recientemente, sin embargo, una API moderna ha adquirido algunas características que las hacen extraordinariamente valiosas y útiles:
 - Las API modernas se adhieren a los estándares (generalmente HTTP y REST), que son amigables para los desarrolladores, de fácil acceso y comprensibles ampliamente
 - Se tratan más como productos que como código. Están diseñados para el consumo de audiencias específicas, están documentados y están versionados de manera que los usuarios puedan tener ciertas expectativas sobre su mantenimiento y ciclo de vida.
 - Debido a que están mucho más estandarizados, tienen una disciplina mucho más sólida para la seguridad y la gobernanza, además de monitorear y administrar el rendimiento y la escalabilidad.
 - Como cualquier otra pieza de software producido, una API moderna tiene su propio ciclo de vida de desarrollo de software (SDLC) de diseño, prueba, construcción, administración y control de versiones.

© JMA 2020. All rights reserved

API Strategy

- Una empresa debe desarrollar una estrategia de API que consista en APIs tanto públicas como privadas. Cuando una empresa lanza APIs públicas que potencian las aplicaciones orientadas al consumidor, habilita nuevas formas de interactuar y conectarse con sus clientes a través de aplicaciones web, móviles y sociales. Al desarrollar APIs privadas, las empresas pueden ofrecer a sus empleados y socios nuevas herramientas que les ayuden a agilizar las operaciones y servir a los clientes aún mejor. En este entorno dinámico, a medida que más y más empresas crean e incorporan APIs, es cada vez más crítico que las empresas innovadoras desarrollen y ejecuten estrategias API de éxito.
- Como ejemplo de los beneficios que una API Strategy puede aportar a una organización, alrededor de 2002, Jeffrey Preston Bezos, director ejecutivo de Amazon, envió un correo a sus empleados con los siguientes puntos:
 - Todos los equipos expondrán sus datos y funcionalidad a través de interfaces de servicios.
 - Los equipos deben comunicarse entre sí a través de estas interfaces.
 - No se permitirá otra forma de comunicación: ni vinculación directa, ni acceso directo a bases de datos de otros equipos, ni memoria compartida ni utilización de ningún tipo de puerta trasera. Sólo se permitirán comunicaciones a través de llamadas que utilicen interfaces de red.
 - La tecnología empleada por cada equipo no debe ser un problema.
 - Todas las interfaces de los servicios, sin excepción, deben ser diseñadas con el objetivo de ser externalizables. Esto es, el equipo debe planear y diseñar sus interfaces para los desarrolladores del resto del mundo. Sin excepciones.
- El correo finalizaba de la siguiente manera: “Todo aquel que no siga las directrices será despedido. Gracias, ¡pasad un buen día!”. Desde hace ya varios años Amazon es el primer proveedor IaaS mundial distanciado significativamente de sus competidores.

© JMA 2020. All rights reserved

API Economy

- El ecosistema de APIs especifica de qué manera el uso de estas micro aplicaciones por terceros puede beneficiar económicamente a una organización, bien por reducción de costes o bien por alquiler o venta de sus propios desarrollos:
 - API as a Service: Obtención de beneficios mediante la exposición de APIs de servicios que son valiosos para terceros y están dispuestos a pagar por su uso.
 - API Products: Desarrollo de herramientas encargadas de facilitar la exposición e integración de aplicaciones a través de sus APIs.
- Una API Economy es, en definitiva, un servicio basado en API que demuestra algún tipo de rentabilidad al negocio, ya sea económica o estratégicamente. Es fundamental pensar en la API como un producto. La economía está cambiando gracias a que las APIs abren nuevos canales, tanto de ingresos como de innovación.
- En general existen muchos servicios APIs de terceros que permiten a un negocio escalar rápidamente para crear productos finales con una inversión y riesgo mínimo.
- Una empresa puede cambiar su estrategia de ventas a la comercialización como proveedor de servicios APIs a terceros: los recursos aquí son sus datos y servicios. La API Economy es un facilitador para convertir una empresa u organización en una plataforma.

© JMA 2020. All rights reserved

Tipos de API por propósito

- Es raro que una organización decida que necesita una API de la nada; la mayoría de las veces, las organizaciones comienzan con una idea, aplicación, innovación o caso de uso que requiere conectividad a otros sistemas o conjuntos de datos. Las APIs entran en escena como un medio para permitir la conectividad entre los sistemas y los conjuntos de datos que deben integrarse.
- Las organizaciones pueden implementar APIs para muchos propósitos: desde exponer internamente la funcionalidad de un sistema central hasta habilitar una aplicación móvil orientada al cliente. El marco de conectividad incluye:
 - APIs del sistema: las APIs del sistema desbloquean datos de los sistemas centrales de registro dentro de una organización. Los ejemplos de sistemas críticos de los que las API podrían desbloquear datos incluyen ERP, sistemas de facturación, CRM y bases de datos.
 - APIs de proceso: las APIs de proceso interactúan y dan forma a los datos dentro de un solo sistema o entre sistemas, rompiendo los silos de datos. Las APIs de proceso proporcionan un medio para combinar datos y organizar varias APIs del sistema para un propósito comercial específico. Algunos ejemplos de esto incluyen la creación de una vista de 360 grados del cliente, el cumplimiento del pedido y el estado del envío.
 - APIs de experiencia: las APIs de experiencia proporcionan un contexto empresarial para los datos y procesos que se desbloquearon y establecieron con las APIs de proceso y sistema. Las APIs de experiencia exponen los datos para que los consuma su público objetivo; esto funciona en un amplio conjunto de canales en una variedad de formas. Algunos ejemplos son las aplicaciones móviles, los portales internos para los datos del cliente o un sistema de cara al cliente que rastrea las entregas.

© JMA 2020. All rights reserved

Tipos de API por estrategias de gestión

- Una vez que se haya determinado el caso de uso de las APIs en la organización, es hora de determinar quién accederá a estas APIs. La mayoría de las veces, el caso de uso y el usuario previsto van de la mano; por ejemplo, es posible que desee mostrar los datos del cliente para sus agentes de ventas y servicios internos; el usuario final previsto, en este caso, son los empleados internos.
- Los tres tipos de APIs según cómo se administran y quién accede a ellas son:
 - APIs externas: Los terceros, que son externos a la organización, pueden acceder a las APIs externas. A menudo, hacen que los datos y servicios de una organización sean fácilmente accesibles en autoservicio por desarrolladores de todo el mundo que buscan crear aplicaciones e integraciones innovadoras.
 - APIs internas: Las APIs internas son lo opuesto a las APIs abiertas, ya que no son accesibles para los consumidores externos y solo están disponibles para los desarrolladores internos de una organización. Las APIs internas pueden permitir iniciativas en toda la empresa, desde la adopción de DevOps y arquitecturas de microservicios hasta la modernización heredada y la transformación digital. El uso y la reutilización de estas APIs pueden mejorar la productividad, la eficiencia y la agilidad de una organización.
 - APIs de socios: Las APIs de socios se encuentran en algún lugar entre las APIs internas y externas. Son APIs a las que acceden otras personas ajenas a la organización con permisos exclusivos. Por lo general, este acceso especial se otorga a terceros específicos para facilitar una asociación comercial estratégica. Un caso de uso común de una API de socio es cuando dos organizaciones desean compartir datos entre sí, se configuraría una API de socio para que cada organización tenga acceso a los datos necesarios con el conjunto correcto de credenciales y permisos.

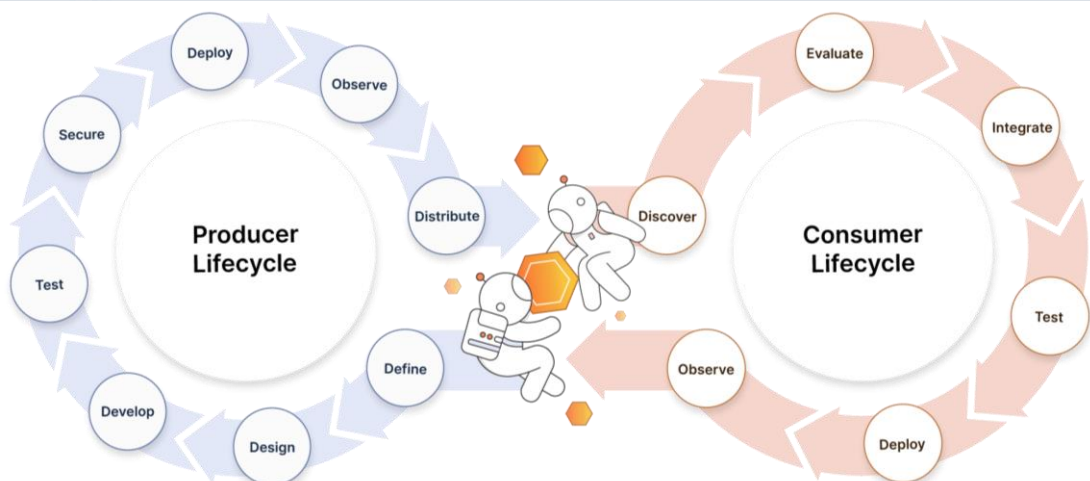
© JMA 2020. All rights reserved

API First

- El enfoque basado en API First significa que, para cualquier proyecto de desarrollo dado, las APIs se tratan como "ciudadanos de primera clase": que todo sobre un proyecto gira en torno a la idea de que el producto final es un conjunto de APIs consumido por las aplicaciones del cliente.
- El enfoque de API First implica que los desarrollos de APIs sean consistentes y reutilizables, lo que se puede lograr mediante el uso de un lenguaje formal de descripción de APIs para establecer un contrato sobre cómo se supone que se comportará la API. Establecer un contrato implica pasar más tiempo pensando en el diseño de una API.
- A menudo también implica una planificación y colaboración adicionales con las partes interesadas, proporcionando retroalimentación de los consumidores sobre el diseño de una API antes de escribir cualquier código evitando costosos errores.
- Entre sus ventajas se encuentran:
 - Los equipos de desarrollo pueden trabajar en paralelo.
 - Reduce el coste de desarrollar aplicaciones
 - Aumenta la velocidad de desarrollo.
 - Asegura buenas experiencias de desarrollador
 - Reduce el riesgo de fallos
 - Proporcionar un sólido perímetro de seguridad

© JMA 2020. All rights reserved

Ciclo de vida de las API



© JMA 2020. All rights reserved

<https://www.postman.com/api-first/>

CONSULTAS ENTRE SERVICIOS

© JMA 2020. All rights reserved

Eureka

- Eureka permite registrar y localizar microservicios existentes, informar de su localización, su estado y datos relevantes de cada uno de ellos. Además, permite el balanceo de carga y tolerancia a fallos.
 - Eureka dispone de un módulo servidor que permite crear un servidor de registro de servicios y un módulo cliente que permite el auto registro y descubrimiento de microservicios.
 - Cuando un microservicio arranca, se comunicará con el servidor Eureka para notificarle que está disponible para ser consumido. El servidor Eureka mantendrá la información de todos los microservicios registrados y su estado. Cada microservicio le notificará, cada 30 segundos, su estado mediante heartbeats.
 - Si pasados tres periodos heartbeats no recibe ninguna notificación del microservicio, lo eliminará de su registro. Si después de sacarlo del registro recibe tres notificaciones, entenderá que ese microservicio vuelve a estar disponible.
 - Cada cliente o microservicio puede recuperar el registro de otros microservicios registrados y quedará cacheado en dicho cliente.
 - Para los servicios que no están basados en Java, hay disponibles clientes Eureka para otros lenguaje y el servidor Eureka expone todas sus operaciones a través de un [API REST](#) que permiten la creación de clientes personalizados.
-

© JMA 2020. All rights reserved

Eureka Server

- Añadir al proyecto:
 - Spring Boot + Cloud Discovery: Eureka Server + Core: Cloud Bootstrap
- Anotar aplicación:
@EnableEurekaServer
@SpringBootApplication
public class MsEurekaServiceDiscoveryApplication {
- Configurar:
#Servidor Eureka Discovery Server
eureka.instance.hostname: localhost
eureka.client.registerWithEureka: false
eureka.client.fetchRegistry: false
server.port: \${PORT:8761}
- Arrancar servidor
- Acceder al dashboard de Eureka: <http://localhost:8761/>

© JMA 2020. All rights reserved

Auto registro de servicios

- Añadir al proyecto:
 - Eureka Discovery, Cloud Bootstrap
- Anotar aplicación (ahora es opcional):
~~@EnableEurekaClient~~ @EnableDiscoveryClient
@SpringBootApplication
public class MsEurekaServiceDiscoveryApplication {
- Configurar:
Service registers under this name
spring.application.name=educado-service
Discovery Server Access
eureka.client.serviceUrl.defaultZone=\${DISCOVERY_URL:http://localhost:8761}/eureka/
- Arrancar microservicio y refrescar dashboard de Eureka:
 - <http://localhost:8761/>
- Se puede usar EurekaClient o DiscoveryClient para descubrir las instancias de un servicio:
@Autowired
private EurekaClient discoveryClient;

InstanceInfo instance = discoveryClient.getNextServerFromEureka(nombre, false);
return instance.getHomePageUrl();

© JMA 2020. All rights reserved

HttpClient nativo

- El HttpClient nativo se introdujo como un módulo incubador en Java 9 y de forma definitiva en Java 11 como parte de JEP 321 .
- HttpClient reemplaza la clase heredada HttpURLConnection presente en el JDK desde las primeras versiones de Java.
- Algunas de sus características incluyen:
 - Soporte para HTTP/1.1, HTTP/2 y Web Socket.
 - Soporte para modelos de programación sincrónicos y asincrónicos.
 - Manejo de cuerpos de solicitud y respuesta como flujos reactivos.
 - Soporte para cookies.

© JMA 2020. All rights reserved

HttpClient nativo

```
HttpClient client = HttpClient.newBuilder()
    .version(Version.HTTP_2)
    .followRedirects(Redirect.NORMAL)
    .build();

HttpRequest request = HttpRequest.newBuilder()
    .uri(new URI("https://picsum.photos/v2/list?limit=10"))
    .GET()
    .header("Accept", "application/json")
    .timeout(Duration.ofSeconds(10))
    .build();

client.sendAsync(request, BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println)
    .join();
```

© JMA 2020. All rights reserved

RestTemplate

- La RestTemplate proporciona un API de nivel superior sobre las bibliotecas de cliente HTTP y facilita la invocación de los endpoint REST en una sola línea. Para incorporarlo en Maven:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
</dependency>
<dependency>
  <groupId>com.fasterxml.jackson.core</groupId>
  <artifactId>jackson-databind</artifactId>
</dependency>
```
- Para poder inyectar la dependencia:

```
@Bean public RestTemplate restTemplate(RestTemplateBuilder builder) {
    return builder.build();
}
@Autowired RestTemplate srvRest;
```

© JMA 2020. All rights reserved

RestTemplate

Grupo de métodos	Descripción
getForObject	Recupera una representación a través de GET.
getForEntity	Recupera un ResponseEntity(es decir, estado, encabezados y cuerpo) utilizando GET.
headForHeaders	Recupera todos los encabezados de un recurso utilizando HEAD.
postForLocation	Crea un nuevo recurso utilizando POST y devuelve el encabezado Location de la respuesta.
postForObject	Crea un nuevo recurso utilizando POST y devuelve la representación del objeto de la respuesta.
postForEntity	Crea un nuevo recurso utilizando POST y devuelve la representación de la respuesta.
put	Crea o actualiza un recurso utilizando PUT.

© JMA 2020. All rights reserved

RestTemplate

Grupo de métodos	Descripción
patchForObject	Actualiza un recurso utilizando PATCH y devuelve la representación de la respuesta.
delete	Elimina los recursos en el URI especificado utilizando DELETE.
optionsForAllow	Recupera los métodos HTTP permitidos para un recurso utilizando ALLOW.
exchange	Versión más generalizada (y menos crítica) de los métodos anteriores que proporciona flexibilidad adicional cuando es necesario. Acepta a RequestEntity (incluido el método HTTP, URL, encabezados y cuerpo como entrada) y devuelve un ResponseEntity.
execute	La forma más generalizada de realizar una solicitud, con control total sobre la preparación de la solicitud y la extracción de respuesta a través de interfaces de devolución de llamada.

© JMA 2020. All rights reserved

RestTemplate

- Para recuperar uno:

```
PersonaDTO rsIt = srvRest.getForObject("http://localhost:8080/api/personas/{id}", PersonaDTO.class, 1);
```
- Para recuperar todos (si no se dispone de una implementación de List<PersonaDTO>):

```
ResponseEntity<List<PersonaDTO>> response =  
    srvRest.exchange("http://localhost:8080/api/personas",  
        HttpMethod.GET,  
        HttpEntity.EMPTY, new  
        ParameterizedTypeReference<List<PersonaDTO>>() {  
        });  
List<PersonaDTO> rsIt = response.getBody();
```

© JMA 2020. All rights reserved

RestTemplate

- Para crear o modificar un recurso:

```
ResponseEntity<PersonaDTO> httpRslt = srvRest.postForEntity(  
    "http://localhost:8080/api/personas", new PersonaDTO("pepito",  
    "grillo")), PersonaDTO.class);
```

- Para crear o modificar un recurso con identificador:

```
srvRest.put("http://localhost:8080/api/personas/{id}", new  
    PersonaDTO(new Persona("Pepito", "Grillo"))), 111);
```

- Para borrar un recurso con identificador:

```
srvRest.delete("http://localhost:8080/api/personas/{id}", 111);
```

© JMA 2020. All rights reserved

RestTemplate

- De forma predeterminada, RestTemplate lanzará una de estas excepciones en caso de un error de HTTP:

- HttpClientErrorException: en estados HTTP 4xx
- HttpServerErrorException: en estados HTTP 5xx
- UnknownHttpStatusException: en caso de un estado HTTP desconocido.

- Para vigilar las excepciones:

```
} catch (HttpClientErrorException e) {  
    switch (e.getStatusCode()) {  
        case BAD_REQUEST:  
        case NOT_FOUND:  
            // ...  
            break;
```

© JMA 2020. All rights reserved

LinkDiscoverers

- Cuando se trabaja con representaciones habilitadas para hipermedia, una tarea común es encontrar un enlace con un tipo de relación particular en ellas.
- Spring HATEOAS proporciona implementaciones basadas en JSONPath de la interfaz LinkDiscoverer.

```
<dependency>  
  <groupId>com.jayway.jsonpath</groupId>  
  <artifactId>json-path</artifactId>  
</dependency>
```
- Para acceder a un enlace:

```
String resp = srvRest.getForObject("http://localhost:8080/personas/1", String.class);  
LinkDiscoverer discoverer = new HalLinkDiscoverer();  
Link link = discoverer.findLinkWithRel("direcciones", resp);  
if(link != null)  
    direccionesURL = link.getHref();
```

© JMA 2020. All rights reserved

Feign

- Feign es un cliente declarativo de servicios web.
- Facilita la escritura de clientes de servicios web (proxies) mediante la creación de una interfaz anotada.
- Tiene soporte de anotación conectable que incluye anotaciones Feign y JAX-RS.
- Feign también soporta codificadores y decodificadores enchufables.
- Spring Cloud agrega soporte para las anotaciones de Spring MVC y para usar el mismo HttpMessageConverters usado de forma predeterminada en Spring Web.
- Spring Cloud integra Ribbon y Eureka para proporcionar un cliente http con equilibrio de carga cuando se usa Feign.
- Dispone de un amplio juego de configuraciones.

© JMA 2020. All rights reserved

Feign

- Dependencia: Spring Cloud Routing > OpenFeign
- Anotar la clase principal con:
`@EnableFeignClients("com.example.proxies")`
- Crear un interfaz por servicio:
`@FeignClient(name = "personas", url = "http://localhost:8002")
// @FeignClient(name = "personas-service") // Eureka
public interface PersonaProxy {
 @GetMapping("/personas")
 List<PersonaDTO> getAll();
 @GetMapping("/personas/{id}")
 PersonaDTO getOne(@PathVariable int id);
 @PutMapping(value = "/personas/{id}", consumes = "application/json")
 PersonaDTO update(@PathVariable("id") id, PersonaDTO persona);
}`
- Inyectar la dependencia:
`@Autowired
PersonaProxy srvRest;`

© JMA 2020. All rights reserved

@HttpExchange (v.6)

- En Spring, una interfaz de servicio HTTP es una interfaz Java con métodos `@HttpExchange`. El método anotado se trata como un punto final HTTP, y los detalles se definen estáticamente a través de atributos de anotación, así como a través de los tipos de argumentos del método de entrada.
- `@HttpExchange` es la anotación genérica para especificar un punto final HTTP. Cuando se utiliza a nivel de interfaz, se aplica a todos los métodos. Está disponible como `@GetExchange`, `@PostExchange`, `@PutExchange`, `@PatchExchange`, `@DeleteExchange`.
- `@HttpExchange` permite definir a nivel de interfaz url (ruta base), method, accept y contentType.
- Los métodos de intercambio admiten los siguientes parámetros en la firma del método:
 - `@PathVariable`: sustituye un marcador de posición por un valor en la URL de la solicitud.
 - `@RequestBody`: proporciona el cuerpo de la solicitud.
 - `@RequestParam`: añade los parámetros de la petición. Cuando content-type está configurado como application/x-www-form-urlencoded, los parámetros de la petición se codifican en el cuerpo de la petición. En caso contrario, se añaden como parámetros de consulta de la URL.
 - `@RequestHeader`: añade los nombres y valores de las cabeceras de la petición.
 - `@RequestPart`: se puede utilizar para añadir una parte de la petición (campo de formulario, recurso o HttpEntity).
 - `@CookieValue`: añade cookies a la petición.

© JMA 2020. All rights reserved

@HttpExchange (v.6)

```
@HttpExchange(url = "/actores/v1", accept = "application/json", contentType = "application/json")
public interface ActoresProxy {
    public record ActorShort(int id, String nombre) {}
    public record ActorEdit(int id, String nombre, String apellidos) {}

    @GetExchange
    List<ActorShort> getAll();
    @GetExchange("/{id}")
    ActorEdit getOne(@PathVariable int id);
    @PostExchange
    ResponseEntity<ActorEdit> add(@RequestBody ActorEdit item);
    @PutExchange("/{id}")
    void change(@PathVariable int id, @RequestBody ActorEdit item);
    @DeleteExchange("/{id}")
    void delete(@PathVariable int id);
}
```

© JMA 2020. All rights reserved

@HttpExchange (v.6)

- Un método de intercambio HTTP puede devolver:
 - Clases (modo bloqueante) o clases reactivas (Mono/Flux).
 - ResponseEntity<T> que contiene el estado, los encabezados y el cuerpo deserializado
 - void si el método se trata como sólo de ejecución
- HttpServiceProxyFactory es una fábrica para crear un proxy de cliente a partir de una interfaz de servicio HTTP.

```
@Bean
WebClient webClient() {
    return WebClient.builder().baseUrl("http://localhost:8010/").build();
}

@Bean
ActoresProxy actoresProxy(WebClient webClient) {
    HttpServiceProxyFactory httpServiceProxyFactory = HttpServiceProxyFactory
        .builder(WebClientAdapter.forClient(webClient)).build();
    return httpServiceProxyFactory.createClient(ActoresProxy.class);
}
```

© JMA 2020. All rights reserved

Spring Cloud LoadBalancer

- Un balanceador o equilibrador de carga fundamentalmente es un dispositivo de hardware o software que se interpone al frente de un conjunto de servidores que atienden una aplicación y, tal como su nombre lo indica, asigna o reparte las solicitudes que llegan de los clientes a los servidores usando algún algoritmo (desde un simple round-robin hasta algoritmos más sofisticados).
- Spring Cloud proporciona su propia abstracción e implementación del equilibrador de carga del lado del cliente. Para el mecanismo de equilibrio de carga, `ReactiveLoadBalancer`, se ha agregado una interfaz y se le han proporcionado implementaciones basadas en Round-Robin y Random.
- El balanceo de carga se basa en el descubrimiento de servicios que utiliza el cliente de descubrimiento disponible en la ruta de clases, como Spring Cloud Netflix Eureka, Spring Cloud Consul Discovery o Spring Cloud Zookeeper Discovery.
- Spring Cloud LoadBalancer puede integrarse con:
 - Spring RestTemplate como cliente de equilibrador de carga
 - Spring WebClient como cliente de equilibrador de carga
 - Spring OpenFeign como cliente de equilibrador de carga
 - Spring WebFlux WebClient con `ReactorLoadBalancerExchangeFilterFunction`
- Dependencia: Spring Cloud Routing > Cloud LoadBalancer

© JMA 2020. All rights reserved

Spring Cloud LoadBalancer

- Spring Cloud LoadBalance permite:
 - Cambiar entre los algoritmos de equilibrio de carga
 - Almacenamiento en caché
 - Equilibrio de carga basado en zonas
 - Comprobación del estado de las instancias (HealthCheck)
 - Establecer preferencia de Misma instancia, Sesión fija basada en solicitudes, basado en sugerencias.
 - Transformar la solicitud HTTP en el proceso de equilibrio de carga antes de enviarla.

© JMA 2020. All rights reserved

Cientes Load Balancer

- La anotación `@LoadBalanced` configura los diferentes clientes para que utilicen el balanceo de carga:

```
@LoadBalanced
@Bean RestTemplate restTemplate(RestTemplateBuilder builder) { return builder.build(); }
@LoadBalanced
@Bean public WebClient.Builder webClientBuilder() { return WebClient.builder(); }
```
- Las peticiones sustituyen en la URL el nombre del dominio por el nombre registrado en el servidor de descubrimiento para que utilicen el balanceo de carga:

```
return restTemplate.getForObject("lb://personas-service/resource", String.class);
return webClientBuilder.build().get().uri("lb://personas-service/resource")
    .retrieve().bodyToMono(String.class);

@FeignClient(name = "personas-service")
public interface PersonaProxy {
```

© JMA 2020. All rights reserved

Spring Cloud Gateway

- Spring Cloud Gateway se puede definir como un proxy inverso o edge service (fachada) que va a permitir tanto enrutar y filtrar las peticiones de manera dinámica, así como monitorizar, balancear y securizar las mismas.
- Este componente actúa como un punto de entrada a los servicios públicos, es decir, se encarga de solicitar una instancia de un microservicio concreto a Eureka y de su enrutamiento hacia el servicio que se desea consumir.
- Las peticiones pasarán de manera individual por cada uno de los filtros que componen la configuración de Spring Cloud Gateway. Estos filtros harán que la petición sea rechazada por determinados motivos de seguridad en función de sus características, sea dirigida a la instancia del servicio apropiada, que sea etiquetada y registrada con la intención de ser monitorizada.

© JMA 2020. All rights reserved

Spring Cloud Gateway

- Añadir proyecto:
 - Spring Cloud Routing Gateway, Eureka Discovery Client
- Anotar la aplicación:
 - @EnableDiscoveryClient
 - @EnableEurekaClient
- Configurar:
 - eureka:
 - client:
 - fetchRegistry: true
 - registerWithEureka: false
 - serviceUrl:
 - defaultZone: \${DISCOVERY_URL:http://localhost:8761}/eureka/
 - instance:
 - appname: apigateway-server
 - server:
 - port: \${PORT:8080}

© JMA 2020. All rights reserved

Spring Cloud Gateway

- Ruta: el bloque de construcción básico de la puerta de enlace. Está definido por un ID, un URI de destino, una colección de predicados y una colección de filtros. Una ruta coincide si el predicado agregado es verdadero.
- Predicado: Patrón de coincidencia con las solicitud (Spring FrameworkServerWebExchange), es un predicado de función de Java 8. El tipo de entrada, permite hacer coincidir cualquier cosa desde la solicitud HTTP, como encabezados o parámetros. Spring Cloud Gateway incluye múltiples factorías de predicados de ruta integradas.
- Filtro: Se pueden modificar las solicitudes y respuestas antes o después de enviar la solicitud descendente. Spring Cloud Gateway incluye múltiples factorías de filtros integradas.
- Los clientes realizan solicitudes a Spring Cloud Gateway. Si la asignación del controlador de la puerta de enlace determina que una solicitud coincide con una ruta, se envía al controlador web de la puerta de enlace. Este controlador ejecuta la solicitud a través de una cadena de filtros que es específica de la solicitud. Los filtros pueden ejecutar la lógica antes y después de que se envíe la solicitud del proxy.
- Hay dos formas de configurar predicados y filtros: imperativamente por código o declarativamente en application.properties.

© JMA 2020. All rights reserved

Spring Cloud Gateway

```
spring:
  cloud:
    gateway:
      routes:
        - id: serv-catalogo
          #Se utiliza el esquema lb:// cuando se va a acceder a través de Eureka
          uri: lb://catalogo-service
          predicates:
            - Path=/catalogo/**
          filters:
            - RewritePath=/catalogo/*, /
        - id: serv-clientes
          uri: lb://clientes-service
          predicates:
            - Path=/clientes/**
          filters:
            - RewritePath=/clientes/*, /
        - id: serv-search
          uri: https://www.google.com/
          predicates:
            - Path=/search/**
          filters:
            - RewritePath=/search/*, /
```

© JMA 2020. All rights reserved

MONITORIZACIÓN Y RESILIENCIA

© JMA 2020. All rights reserved

Estado y diagnóstico

- Como desarrollador, operador de TI, ingeniero de DevOps o ingeniero de Site Reliability Engineering (SRE), eres responsable del rendimiento y del estado de las aplicaciones que creas o utilizas. Gracias a los datos de telemetría, puedes determinar si una aplicación está en buen estado y tiene un rendimiento óptimo.
- Un microservicio debe notificar su estado y diagnóstico. En caso contrario, hay poca información desde una perspectiva operativa. Correlacionar eventos de diagnóstico en un conjunto de servicios independientes y tratar los desajustes en el reloj de la máquina para dar sentido al orden de los eventos suponen un reto.
- De la misma manera que interactúa con un microservicio según protocolos y formatos de datos acordados, hay una necesidad de estandarizar cómo registrar los eventos de estado y diagnóstico que, en última instancia, terminan en un almacén de eventos para que se consulten y se vean.
- En un enfoque de microservicios, es fundamental que distintos equipos se pongan de acuerdo en un formato de registro único.
- Debe haber un enfoque coherente para ver los eventos de diagnóstico en la aplicación.

© JMA 2020. All rights reserved

Comprobaciones de estado

- El estado es diferente del diagnóstico.
- El estado trata de cuando el microservicio informa sobre su estado actual para que se tomen las medidas oportunas.
- Un buen ejemplo es trabajar con los mecanismos de actualización e implementación para mantener la disponibilidad.
- Aunque un servicio podría actualmente estar en mal estado debido a un bloqueo de proceso o un reinicio de la máquina, puede que el servicio siga siendo operativo.
- Lo último que debe hacer es realizar una actualización que empeore esta situación.
- El mejor método consiste en realizar una investigación en primer lugar o dar tiempo a que el microservicio se recupere.
- Los eventos de estado (HealthChecks) de un microservicio nos ayudan a tomar decisiones informadas y, en efecto, ayudan a crear servicios de reparación automática.

© JMA 2020. All rights reserved

Monitorización

- Con la versión 2 de Spring Boot se ha adoptado Micrometer como librería para proporcionar las métricas.
- Micrometer permite exportar a cualquiera de los más populares sistemas de monitorización los datos de las métricas.
- Usando Micrometer la aplicación se abstrae del sistema de métricas empleado pudiendo cambiar en un futuro si se desea.
- Uno de los sistemas más populares de monitorización es Prometheus que se encarga de recoger y almacenar los datos de las métricas expuestas por las aplicaciones y ofrece un lenguaje de consulta de los datos con el que otras aplicaciones pueden visualizarlos en gráficas y paneles de control.
- Grafana es una de estas herramientas que permite visualizar los datos proporcionados por Prometheus.
- Estos sistemas de monitorización ofrecen un sistema de alertas que se integran entre otros con Slack.

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

- Los actuators de Spring Boot ofrecen funcionalidades listas para el entorno de producción.
- Supervisan la aplicación, recopilan métricas, comprenden y analizan el tráfico y el estado de la base de datos, y todo ello listo para usar.
- Los Actuators se utilizan principalmente para exponer información operacional sobre la aplicación en ejecución (health, metrics, info, dump, env, etc.).
- Los puntos finales de los actuadores permiten monitorear e interactuar con la aplicación. Spring Boot incluye varios puntos finales incorporados y permite agregar personalizados.
- Cada punto final individual puede ser habilitado o deshabilitado. Esto determina si el punto final se crea o no y si su bean existe en el contexto de la aplicación. Para ser accesible de forma remota, un punto final también debe estar expuesto a través de JMX o HTTP .
- La mayoría de las aplicaciones eligen HTTP, donde se asigna a una URL al ID del punto final con el prefijo de /actuator/.

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

ID	Descripción
auditevents	Expone la información de eventos de auditoría para la aplicación actual.
beans	Muestra una lista completa de todos los beans de la aplicación.
caches	Expone cachés disponibles.
conditions	Muestra las condiciones que se evaluaron en las clases de configuración y configuración automática así como los motivos por los que coincidieron o no.
configprops	Muestra una lista de todas las @ConfigurationProperties.
env	Expone propiedades de Spring's ConfigurableEnvironment.
health	Muestra información de salud de la aplicación.
httptrace	Muestra información de rastreo HTTP (por defecto, los últimos 100 intercambios de solicitud-respuesta HTTP).
info	Muestra información de la aplicación.
integrationgraph	Muestra el gráfico de integración de Spring.
loggers	Muestra y modifica la configuración de los loggers en la aplicación.
metrics	Muestra información de 'métricas' para la aplicación actual.
mappings	Muestra una lista ordenada de todas las rutas @RequestMapping.
scheduledtasks	Muestra las tareas programadas en la aplicación.
sessions	Permite la recuperación y eliminación de sesiones de usuario de un almacén de sesiones respaldado por Spring Session. No disponible cuando se usa el soporte de Spring Session para aplicaciones web reactivas.

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

- Instalación: Spring Ops Actuator
- Se agrega una "página de descubrimiento" con enlaces a todos los puntos finales: /actuator.
- De forma predeterminada, todos los puntos finales, excepto shutdown están habilitados:
management.endpoint.shutdown.enabled=true
- Dado que los puntos finales pueden contener información confidencial, se debe considerar cuidadosamente cuándo exponerlos:
management.endpoints.web.exposure.exclude=*
management.endpoints.web.exposure.include=info, health
management.endpoints.web.exposure.include=*
- Deberían asegurarse los puntos finales HTTP de la misma forma que se haría con cualquier otra URL sensible.
management.security.enabled=false
- Los diferentes puntos finales se pueden configurar:
management.endpoints.health.sensitive=*
info.app.name=\${spring.application.name}
info.app.description=Catalogo del videoclub
info.app.version=1.0.0
management.info.env.enabled=true

© JMA 2020. All rights reserved

Información de salud

- Se puede usar la información de salud para verificar el estado de la aplicación en ejecución.
- A menudo, el software de monitoreo lo utiliza para alertar cuando un sistema de producción falla.
- La información expuesta por el punto final `health` depende de la propiedad `management.endpoint.health.show-details`:
 - `never`: Los detalles nunca se muestran (por defecto).
 - `when-authorized`: Los detalles solo se muestran a usuarios autorizados. Los roles autorizados se pueden configurar usando `management.endpoint.health.roles`.
 - `always`: Los detalles se muestran a todos los usuarios.

© JMA 2020. All rights reserved

Métricas

- Spring Boot Actuator proporciona administración de dependencias y configuración automática para Micrometer, una fachada de métricas de aplicaciones que admite numerosos sistemas de monitoreo, que incluyen:

AppOptics	Atlas	Datadog
Dynatrace	Elastic	Ganglia
Graphite	Humio	Influx
JMX	KairosDB	New Relic
Prometheus	SignalFx	Simple (in-memory)
StatsD	Wavefront	
- Para habilitar un sistema de monitorización:
 - `management.metrics.export.datadog.enabled = false`
- Se pueden crear métricas personalizadas.

© JMA 2020. All rights reserved

Spring Boot Admin

- Spring Boot Admin es una herramienta para la monitorización de nuestras aplicaciones Spring Boot.
- La aplicación nos proporciona una interfaz gráfica desarrollada para monitorizar aplicaciones Spring Boot aprovechando la información proporcionada por los endpoints de spring-boot-actuator.
- Servidor:
 - Dependencia: Ops > Spring Boot Admin (Server)
 - Anotar la clase principal con `@EnableAdminServer`
 - Configurar puerto y, opcionalmente, URL alternativa a la raíz:
 - `spring.boot.admin.context-path=/admin`
- Clientes:
 - Dependencia: Ops > Spring Boot Admin (Client)
 - Si no se dispone de un servidor de registro/descubrimiento hay que configurar la url del Spring Boot Admin Server
 - `spring.boot.admin.client.url=http://localhost:8000`

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

- Micrometer permite exportar a cualquiera de los más populares sistemas de monitorización los datos de las métricas. Usando Micrometer la aplicación se abstrae del sistema de métricas empleado pudiendo cambiar en un futuro si se desea.
- Uno de los sistemas más populares de monitorización es Prometheus, que se encarga de recoger y almacenar los datos de las métricas expuestas por las aplicaciones y ofrece un lenguaje de consulta de los datos con el que otras aplicaciones pueden visualizarlos en gráficas y paneles de control.
- Grafana es una herramienta especializada en crear cuadros de mando y gráficos a partir de múltiples fuentes, lo que permite visualizar los datos proporcionados por Prometheus.
- Estos sistemas de monitorización ofrecen un sistema de alertas que se integran entre otros con Slack.

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

- Para exponer un actuador específico es necesario instalar la dependencia:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```
- Se puede verificar accediendo a `/actuador/prometheus`
- Una vez expuestas las métricas en el formato que espera Prometheus este ya puede recolectarlas. Es necesario configurar la recopilación (ej: `prometheus.yml`):

```
global:
  scrape_interval: 10s
scrape_configs:
  - job_name: 'spring_micrometer'
    metrics_path: '/actuador/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['192.168.1.11:8010', 'host.docker.internal:8011']
```

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

- Para consultar la recopilación: <http://localhost:9090>
- Para acceder y configurar Grafana:
 - <http://host.docker.internal:3000> user: admin password: admin
- Para poder visualizar las métricas de Prometheus, primero debe agregarlo como fuente de datos (Data Source) en Grafana:
 - Configuration → Data Sources → Add data source → Prometheus.
 - URL: `http://host.docker.internal:9090`, Access: Browser, Save & Test.
- Se puede utilizar Explore para crear consultas ad-hoc para comprender las métricas expuestas por la aplicación.
- Un dashboard (tablero de mando) permite ver de un vistazo de los datos y permite rastrear métricas a través de diferentes visualizaciones. Los dashboards constan de paneles, cada uno de los cuales representa una parte de la historia que se desea que cuente el dashboard. Se pueden importar dashboards ya creados (<https://grafana.com/grafana/dashboards/>) mediante su id o JSON (Ej: *JVM* (*Micrometer*) id: 4701, *Spring Boot* id: 11378, 19004)
- Las alertas permiten identificar problemas en el sistema momentos después de que ocurran. Al identificar rápidamente los cambios no deseados en el sistema, se puede minimizar las interrupciones en los servicios.

© JMA 2020. All rights reserved

Resiliencia

- **Resiliencia (RAE):**
 - Capacidad de un material, mecanismo o sistema para recuperar su estado inicial cuando ha cesado la perturbación a la que había estado sometido.
- Tratar errores inesperados es uno de los problemas más difíciles de resolver, especialmente en un sistema distribuido. Gran parte del código que los desarrolladores escriben implica controlar las excepciones, y aquí también es donde se dedica más tiempo a las pruebas.
- El problema es más complejo que escribir código para controlar los errores. ¿Qué ocurre cuando se produce un error en la máquina en que se ejecuta el microservicio? No solo es necesario detectar este error de microservicio (un gran problema de por sí), sino también contar con algo que reinicie su microservicio.
- Un microservicio debe ser resistente a errores y poder reiniciarse a menudo en otra máquina a efectos de disponibilidad. Esta resistencia también se refiere al estado que se guardó en nombre del microservicio, en los casos en que el estado se puede recuperar a partir del microservicio, y al hecho de si el microservicio puede reiniciarse correctamente. En otras palabras, debe haber resistencia en la capacidad de proceso (el proceso puede reiniciarse en cualquier momento), así como en el estado o los datos (sin pérdida de datos y que se mantenga la consistencia de los datos).

© JMA 2020. All rights reserved

Patrón: Circuit Breaker

- **Motivación:**
 - Se has aplicado la arquitectura de microservicio. Los servicios a veces colaboran en el manejo de solicitudes. Cuando un servicio invoca de forma síncrona a otro, siempre existe la posibilidad de que el otro servicio no esté disponible o muestre una latencia tan alta que sea esencialmente inutilizable. Se pueden consumir recursos preciosos, como subprocessos, en el servicio que hace la petición mientras se espera que el otro servicio responda. Esto podría llevar al agotamiento de los recursos, lo que haría que el servicio que hace la petición no pudiera manejar otras solicitudes. El fallo de un servicio puede potencialmente pasar a otros servicios por toda la aplicación.
- **Intención:**
 - ¿Cómo evitar que un fallo de red o servicio provoque una caída en cascada a otros servicios?
- **Solución:**
 - Las peticiones a un servicio remoto se deben invocar a través de un proxy que funciona de manera similar a un interruptor de circuito eléctrico (disyuntor). Cuando el número de fallos consecutivos cruza un umbral, el interruptor se dispara y, durante un período de tiempo de espera, todos los intentos de invocar el servicio remoto fallarán de inmediato. Una vez que el tiempo de espera expira, el interruptor permite que pase un número limitado de solicitudes de prueba. Si esas solicitudes son correctas, el interruptor reanuda el funcionamiento normal. De lo contrario, si persiste el fallo, el período de tiempo de espera comienza nuevamente.

© JMA 2020. All rights reserved

Patrón: Circuit Breaker

- Implementación:
 - Crear servicios con Spring Boot y Resilience4J, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Los servicios manejan los fallos de los servicios que invocan.
 - Este patrón tiene los siguientes problemas:
 - Es un desafío elegir los valores de tiempo de espera sin crear falsos positivos o introducir una latencia excesiva.
- Patrones relacionados:
 - El Chasis Microservice podría implementar este patrón.
 - El API Gateway usa este patrón para invocar servicios
 - Un enrutador de descubrimiento del lado del servidor podría usar este patrón para invocar servicios

© JMA 2020. All rights reserved

Spring Cloud Circuit Breaker con Resilience4j

- El disyuntor Spring Cloud proporciona una abstracción a través de diferentes implementaciones de disyuntores. Proporciona una API coherente para usar en las aplicaciones, lo que le permite al desarrollador elegir la implementación de disyuntor que mejor se adapte a sus necesidades para su aplicación.
- Las siguientes implementaciones son compatibles:
 - Netflix Hystrix (spring-cloud-starter-netflix-hystrix)
 - Resilience4J (spring-cloud-starter-circuitbreaker-resilience4j)
 - Sentinel (spring-cloud-starter-circuitbreaker-spring-retry)
 - Spring Retry (spring-cloud-starter-circuitbreaker-sentinal)

© JMA 2020. All rights reserved

Resilience4j

- Resilience4j es una biblioteca de tolerancia a fallos liviana y fácil de usar inspirada en Netflix Hystrix, pero diseñada para Java 8 y programación funcional. Liviana, porque la biblioteca solo usa Vavr, que no tiene ninguna otra dependencia de biblioteca externa
- Resilience4j proporciona funciones de orden superior (decoradores) para mejorar cualquier interfaz funcional, expresión lambda o referencia de método con un disyuntor, limitador de velocidad, reintento o mamparo, permitiendo concatenar más de un decorador. La ventaja es que se tiene la opción de seleccionar los decoradores que necesita y nada más.
- Los patrones soportados para aumentar la tolerancia a fallos debido a problemas de red o fallo de alguno de los múltiples servicios son:
 - Circuit breaker: para dejar de hacer peticiones cuando un servicio invocado está fallando.
 - Retry: realiza reintentos cuando un servicio ha fallado de forma temporal.
 - Bulkhead: limita el número de peticiones concurrentes salientes a un servicio para no sobrecargarlo.
 - Rate limit: limita el número de llamadas que recibe un servicio en un periodo de tiempo.
 - Cache: intenta obtener un valor de la cache y si no está presente de la función de la que lo recupera.
 - Time limiter: limita el tiempo de ejecución de una función para no esperar indefinidamente a una respuesta.

<https://resilience4j.readme.io/docs>

© JMA 2020. All rights reserved

Resilience4j

- Instalación: Spring Cloud Circuit Breaker > Resilience4J
- Para proporcionar una configuración predeterminada para todos los disyuntores:
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {
 return factory -> factory.**configureDefault**(id -> new Resilience4JConfigBuilder(id)
 .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build())
 .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
 .build());
}
- De manera similar, se puede proporcionar una configuración personalizada:
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
 return factory -> factory.configure(builder -> builder.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
 .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2)).build()),
 "slow");
}

© JMA 2020. All rights reserved

Resilience4j

- Se puede configurar las instancias de CircuitBreaker e TimeLimiter en el archivo de propiedades de configuración de la aplicación.

```
resilience4j.circuitbreaker:
instances:
  backendA:
    registerHealthIndicator: true
    slidingWindowSize: 100
  backendB:
    registerHealthIndicator: true
    slidingWindowSize: 10
    permittedNumberOfCallsInHalfOpenState: 3
    slidingWindowType: TIME_BASED
resilience4j.timelimiter:
instances:
  backendA:
    timeoutDuration: 2s
    cancelRunningFuture: true
  backendB:
    timeoutDuration: 1s
    cancelRunningFuture: false
```

© JMA 2020. All rights reserved

Resilience4j

- Si resilience4j-bulkhead está en el classpath, Spring Cloud CircuitBreaker ajustará todos los métodos con un mamparo Resilience4j Bulkhead.
- Spring Cloud CircuitBreaker Resilience4j proporciona dos implementaciones de patrón de mamparo (bulkhead):
 - SemaphoreBulkhead: que usa semáforos (predeterminado)
 - FixedThreadPoolBulkhead: que usa una cola limitada y un grupo de subprocesos fijo.
- El Customizer<Resilience4jBulkheadProvider> permite proporcionar una configuración predeterminada para Bulkhead y ThreadPoolBulkhead.

```
@Bean
public Customizer<Resilience4jBulkheadProvider> defaultBulkheadCustomizer() {
    return provider -> provider.configureDefault(id -> new Resilience4jBulkheadConfigurationBuilder()
        .bulkheadConfig(BulkheadConfig.custom().maxConcurrentCalls(4).build())
        .threadPoolBulkheadConfig(ThreadPoolBulkheadConfig.custom().coreThreadPoolSize(1)
            .maxThreadPoolSize(1).build())
        .build());
}
```

© JMA 2020. All rights reserved

Spring Retry

- Spring Retry proporciona compatibilidad con reintentos declarativos para aplicaciones Spring. Spring Retry proporciona una implementación de disyuntor mediante una combinación de él `CircuitBreakerRetryPolicy` y reintentos con estado . Todos los disyuntores creados con Spring Retry se crearán con `CircuitBreakerRetryPolicy` y un `DefaultRetryState`. Ambas clases se pueden configurar usando `SpringRetryConfigBuilder`.
- Para proporcionar una configuración predeterminada para todos los disyuntores:

@Bean

```
public Customizer<SpringRetryCircuitBreakerFactory> defaultCustomizer() {  
    return factory -> factory.configureDefault(id -> new SpringRetryConfigBuilder(id)  
        .retryPolicy(new TimeoutRetryPolicy()).build());  
}
```

© JMA 2020. All rights reserved

Micrometer Tracing y Zipkin

- Una de las funcionalidades esenciales en una aplicación distribuida es la trazabilidad de una petición, desde que entra por el API Gateway pasando por las diferentes peticiones que hacen los microservicios por la red o envío de mensajes. Es necesaria la funcionalidad que relacione las trazas de todos los servicios para depuración o consulta en un futuro para dar visibilidad a las acciones que se realizan en el sistema.
- La técnica que se emplea es asignar a cada petición entrante un identificador para la transacción de forma global y un identificador para la transacción en cada microservicio que varía en cada comunicación de red. Cuando un microservicio se comunica con otro envía en su petición el identificador de la transacción global y el de su transacción (si no los ha recibido, los genera). En el protocolo HTTP los identificadores se envían y reciben a través de las cabeceras.
- Micrometer Tracing proporciona la infraestructura para que las peticiones salientes envíen un identificador de correlación de la petición global y para las peticiones entrantes relacionarlo con la petición global. Se encarga de propagar las cabeceras del servicio cliente al servicio servidor automáticamente instrumentando los clientes `HTTP RestTemplate`, `AsyncRestTemplate` y `WebClient`. Se integra con `OpenZipkin Brave`.
- Zipkin es una herramienta que recolecta las transacciones creadas por Micrometer en la ejecución de los microservicios e información de los tiempos de respuesta de las invocaciones que han intervenido en una transacción. Ofrece las dos funcionalidades la recolección de datos y la obtención de los mismos. Tanto la recolección como el almacenamiento ofrecen diferentes herramientas para implementarlo: la recolección puede ser mediante peticiones HTTP, RabbitMQ o Kafka y el almacenamiento en memoria, MySQL, Cassandra o Elasticsearch.

<https://micrometer.io/docs/tracing>

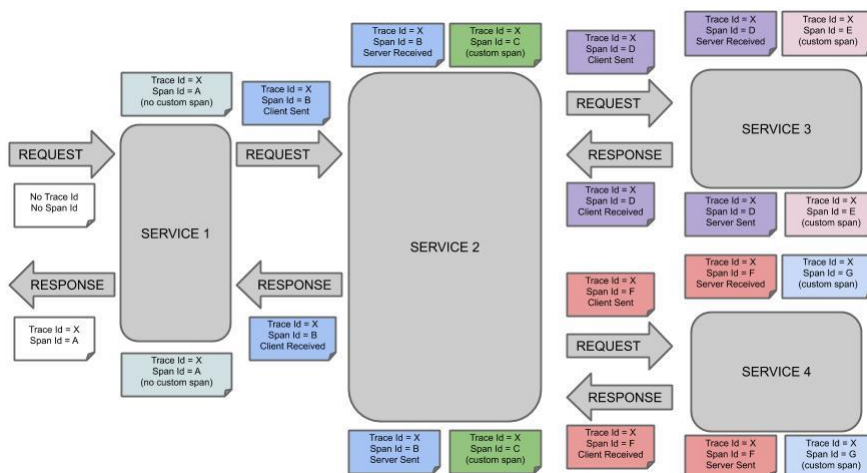
© JMA 2020. All rights reserved

Micrometer Tracing y Zipkin

- **Span:** La unidad básica de trabajo. Los span pueden tener descripciones, eventos con marcas temporales, anotaciones de valor-clave (etiquetas), el ID del span que los provocó y los ID de proceso (normalmente direcciones IP). Los span se pueden iniciar y detener, y realizan un seguimiento de la información de tiempo. Una vez que crea un span, debe detenerse en algún momento en el futuro.
- **Traza:** un conjunto de span que forman una estructura en forma de árbol. Por ejemplo, si ejecuta un almacén de big data distribuido, una solicitud PUT podría formar un seguimiento.
- **Anotación / Evento:** se utiliza para registrar la existencia de un evento en el tiempo. Conceptualmente, en un escenario RPC típico, se marcan estos eventos para resaltar qué tipo de acción tuvo lugar (no significa que físicamente dicho evento se establecerá en un lapso).
 - **Client Sent (cs):** El cliente ha realizado una solicitud. Esta anotación indica el inicio del span.
 - **Server Received (sr):** El lado del servidor recibió la solicitud y comenzó a procesarla. La diferencia entre las marcas temporales del cs y del sr revela la latencia de la red.
 - **Server Sent (ss):** Anotado al completar el procesamiento de la solicitud (cuando la respuesta se envió al cliente). Restarlo de la marca temporal del sr revela el tiempo que necesita el servidor para procesar la solicitud.
 - **Client Received (cr):** Significa el final del span. El cliente ha recibido con éxito la respuesta del lado del servidor. Restarlo de la marca temporal del cs indica todo el tiempo que necesita el cliente para recibir la respuesta del servidor.

© JMA 2020. All rights reserved

Micrometer Tracing y Zipkin



© JMA 2020. All rights reserved

Micrometer Tracing y Zipkin

- Instalación del servidor Zipkin
 - `docker run -d -p 9411:9411 openzipkin/zipkin-slim`
- Agregar dependencias a todos los proyectos que participen en las trazas:
 - Observability > Distributed Tracing, Observability > Zipkin
- Configurar los clientes Zipkin en `application.properties`:
 - `management.zipkin.tracing.endpoint=http://localhost:9411/api/v2/spans`
 - `management.tracing.sampling.probability=1.0`
 - `logging.pattern.level=%5p [${spring.application.name:},%X{traceId:-},%X{spanId:-}]`
- Para consultar las trazas:
 - `http://localhost:9411/`

© JMA 2020. All rights reserved

Micrometer Tracing y Open Feign

- Dependencia:

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-micrometer</artifactId>
</dependency>
```
- Configuración:

```
@Bean
public Capability capability(final MeterRegistry registry) {
    return new MicrometerCapability(registry);
}
```

© JMA 2020. All rights reserved

Micrometer Tracing y Zipkin

- Spring Boot configura el controlador Rest y hace que nuestra aplicación se vincule a un puerto Tomcat. Spring Cloud Sleuth con Brave Tracer proporcionará la instrumentación de la solicitud entrante que suministra a Zipkin como repositorio de las trazas.
- El API de Spring Cloud Sleuth contiene todas las interfaces necesarias para ser implementadas por un trazador: crear, continuar y terminar manualmente un span, agregar anotaciones y eventos a la traza, ...

```
@Autowired Tracer tracer
:
Span newSpan = this.tracer.nextSpan().name("calculateTax");
try (Tracer.SpanInScope ws = this.tracer.withSpan(newSpan.start())) {
    :
    newSpan.tag("taxValue", taxValue);
    :
    newSpan.event("taxCalculated");
}
finally {
    // Once done remember to end the span.
    newSpan.end();
}
```

© JMA 2020. All rights reserved

Patrón: Log Aggregation

- **Motivación:**
 - Ha aplicado el patrón de arquitectura de microservicio. La aplicación consta de múltiples servicios e instancias de servicio que se ejecutan en varias máquinas. Las solicitudes suelen abarcar varias instancias de servicio.
 - Cada instancia de servicio genera información escrita sobre lo que está haciendo en un archivo de registro en un formato estandarizado. El archivo de registro contiene errores, advertencias, información y mensajes de depuración.
- **Intención:**
 - ¿Cómo comprender el comportamiento de la aplicación y solucionar problemas?
- **Requisitos:**
 - Cualquier solución debe tener una sobrecarga mínima de tiempo de ejecución
- **Solución:**
 - Utilizar un servicio de registro centralizado que agregue registros de cada instancia de servicio. Los usuarios pueden buscar y analizar los registros. Pueden configurar alertas que se activan cuando aparecen determinados mensajes en los registros.
- **Implementación:**
 - ELK, AWS Cloud Watch, ...
- **Consecuencias:**
 - Este patrón tiene el siguiente problema: el manejo de un gran volumen de registros requiere una infraestructura sustancial.
- **Patrones relacionados:**
 - Distributed tracing: incluir el ID de solicitud externa en cada mensaje de registro
 - Exception tracking: además de registrar las excepciones, se informa a un servicio de seguimiento de excepciones..

© JMA 2020. All rights reserved

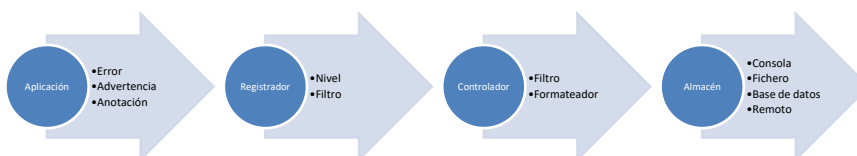
Registros

- El registro es el proceso de escribir en un lugar central mensajes con sucesos, errores o eventos durante la ejecución de un programa. Este registro permite notificar y conservar mensajes de error y advertencia, así como mensajes de información (por ejemplo, estadísticas de tiempo de ejecución) para que se puedan recuperar y analizar posteriormente.
- El objeto que realiza el registro en las aplicaciones generalmente se llama Logger.
- Java define la API de Registro de Java. Esta API de registro permite configurar qué tipos de mensajes se escriben. Las clases individuales pueden usar este registrador para escribir mensajes en los archivos de registro configurados.
- El paquete `java.util.logging` proporciona las capacidades de registro a través de la clase `Logger`.

© JMA 2020. All rights reserved

Registro

- Las aplicaciones realizan llamadas de registro en objetos `Logger` (registradores). Estos objetos `Logger` asignan objetos `LogRecord` que se pasan a los objetos `Handler` (controladores) para su publicación.
- Tanto los registradores como los controladores pueden usar niveles de registro y (opcionalmente) filtros para decidir si están interesados en un registro de registro en particular .
- Cuando es necesario publicar un `LogRecord` externamente, un controlador puede (opcionalmente) usar un formateador para localizar y formatear el mensaje antes de publicarlo en un flujo de E / S.
- Las API está estructuradas de modo que las llamadas al `Logger` tengan un coste mínimo cuando el registro está desactivado.



© JMA 2020. All rights reserved

Niveles de registro

- Cada mensaje de registro tiene un nivel de registro asociado. El nivel ofrece una guía aproximada de la importancia y urgencia de un mensaje de registro. Los objetos de nivel de registro encapsulan un valor entero, y los valores más altos indican prioridades más altas.
- La clase Level define siete niveles de registro estándar:
 - SEVERE (más alta)
 - WARNING
 - INFO
 - CONFIG
 - FINE
 - FINER
 - FINEST (más leve)

© JMA 2020. All rights reserved

Controladores y Formateadores

- Java SE proporciona los siguientes controladores:
 - ConsoleHandler: un controlador simple para escribir registros formateados en System.err
 - StreamHandler: un controlador simple para escribir registros formateados en un OutputStream.
 - FileHandler: un controlador que escribe registros de registro formateados en un solo archivo o en un conjunto de archivos de registro rotativos.
 - SocketHandler: un controlador que escribe registros de registro formateados en puertos TCP remotos.
 - MemoryHandler: un controlador que almacena los registros de registro en la memoria.
- Java SE también incluye dos formateadores estándar:
 - SimpleFormatter : escribe breves resúmenes "legibles por humanos" de los registros.
 - XMLFormatter : escribe información detallada estructurada en XML.
- Se pueden desarrollar nuevos controladores y formateadores que requieran una funcionalidad específica con las abstracciones e interfaces del API.

© JMA 2020. All rights reserved

Configuración

- La configuración de registro se puede inicializar mediante un archivo de configuración de registro que se leerá al inicio (tiene el formato estándar `java.util.Properties`). Alternativamente, la configuración de registro se puede inicializar especificando una clase que se puede utilizar para leer las propiedades de inicialización de fuentes arbitrarias, como LDAP, JDBC, etc.
 - `java.util.logging.MemoryHandler.size=100`
- La configuración inicial puede especificar niveles para registradores particulares. Estos niveles se aplican al registrador nombrado, o cualquier registrador debajo de él en la jerarquía de nombres, y se aplican en el orden en que están definidos en el archivo de configuración.
- La configuración inicial puede contener propiedades arbitrarias para que las utilicen los controladores o los subsistemas que realizan el registro.
- La configuración predeterminada que se envía con JRE pero los ISV, los administradores del sistema y los usuarios finales pueden anularla.
- La configuración predeterminada hace un uso limitado del espacio en disco, no inunda al usuario con información, pero se asegura de capturar siempre la información clave de fallas. Establece un solo controlador en el registrador raíz para enviar la salida a la consola.

© JMA 2020. All rights reserved

Realizar notificaciones

- Para crear un registrador asociado a una clase:
`private final static Logger LOGGER = Logger.getLogger(MyClass.class.getName());`
- Para establecer el nivel de registro:
`LOGGER.setLevel(Level.INFO);`
- Para escribir en el registro:
`LOGGER.severe("Es un error");`
`LOGGER.warning("Es un aviso");`
`LOGGER.info("Solo notifica");`
`LOGGER.finest("Carece de importancia");`
- Para configurar el registro:
`logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n`
`logging.level.org.springframework.web.servlet.DispatcherServlet=DEBUG`
`logging.level.org.hibernate.SQL=debug`
`logging.file.name=C:/curso/logs/demos-elk.log`

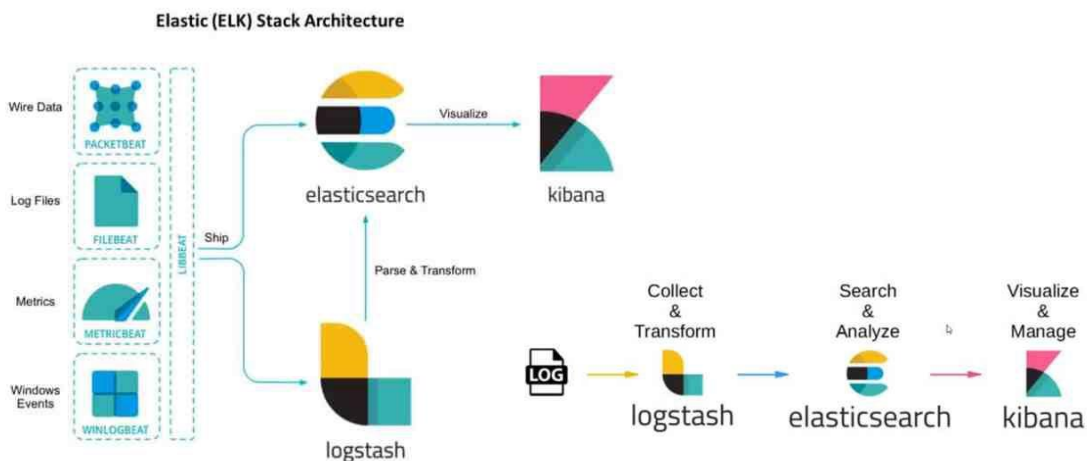
© JMA 2020. All rights reserved

ELK: Elasticsearch, Logstash y Kibana

- “ELK” son las siglas para tres proyectos open source: Elasticsearch, Logstash y Kibana. Elasticsearch es un motor de búsqueda textual y analítica. Logstash es un pipeline (ETL) de procesamiento de datos del lado del servidor que obtiene datos de una multitud de fuentes simultáneamente, los transforma y luego los envía a un almacenamiento como Elasticsearch. Kibana permite a los usuarios visualizar los datos de Elasticsearch en cuadros y gráficas.
- Elasticsearch es un motor de búsqueda y analítica RESTful distribuido capaz de abordar multitud de casos de uso. Como núcleo del Elastic Stack, almacena e indexa de forma centralizada los datos para una búsqueda (estructuradas, no estructuradas, geográficas, métricas, ...) a gran velocidad, con relevancia refinada y analíticas que escalan con facilidad.
- Kibana es una interfaz de usuario gratuita y abierta que permite visualizar los datos de Elasticsearch y navegar en el Elastic Stack, desde rastrear la carga de búsqueda hasta comprender la forma en que las solicitudes fluyen por las apps. Como una imagen vale más que mil líneas de log, Kibana envía los datos en forma de histogramas, grafos de líneas, gráficos circulares, proyecciones solares y más, para análisis de logs, monitoreo de infraestructura, APM (Application Performance Monitoring), operaciones de seguridad, analítica de negocios, ...
- Logstash es un ETL que obtiene, transforma y envía de forma dinámica los datos independientemente de su formato o complejidad. Admite una variedad de entradas de una manera de transmisión continua que extraen eventos de una multitud de fuentes comunes, todo al mismo tiempo: logs, métricas, aplicaciones web, almacenes de datos, varios servicios de AWS, ... los filtros transforman cada evento, para que converjan en un formato común para el análisis y un valor comercial más poderosos. Los resultados se pueden enviar a Elasticsearch para búsquedas y análisis o a una variedad de salidas.

© JMA 2020. All rights reserved

ELK: Elasticsearch, Logstash y Kibana



© JMA 2020. All rights reserved