Microservicios con Spring

Enlaces

- Microservicios
  - https://martinfowler.com/articles/microservices.html
  - https://microservices.io/
- Spring:
  - https://spring.io/projects
- Spring Core
  - https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm
- Spring Data
  - https://docs.spring.io/spring-data/jpa/docs/2.1.5.RELEASE/reference/html/
- Spring MVC
  - https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html
- Spring HATEOAS
  - https://docs.spring.io/spring-hateoas/docs/0.25.1.RELEASE/reference/html
- Spring Data REST
  - https://docs.spring.io/spring-data/rest/docs/3.1.5.RELEASE/reference/html/
- Spring Cloud
  - https://spring.io/projects/spring-cloud
- Ejemplos:
  - https://github.com/spring-projects/spring-data-examples
  - https://github.com/spring-projects/spring-data-rest-webmvc
  - https://github.com/spring-projects/spring-hateoas-examples

INTRODUCCIÓN

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.

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.

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, en el que los servicios independientes realizan funciones distintas y se comunican utilizando un protocolo designado.
- 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.

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 asincrónica. 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 asincrónica.
  - 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.

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

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

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.

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.

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 actualiza 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.

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.

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.

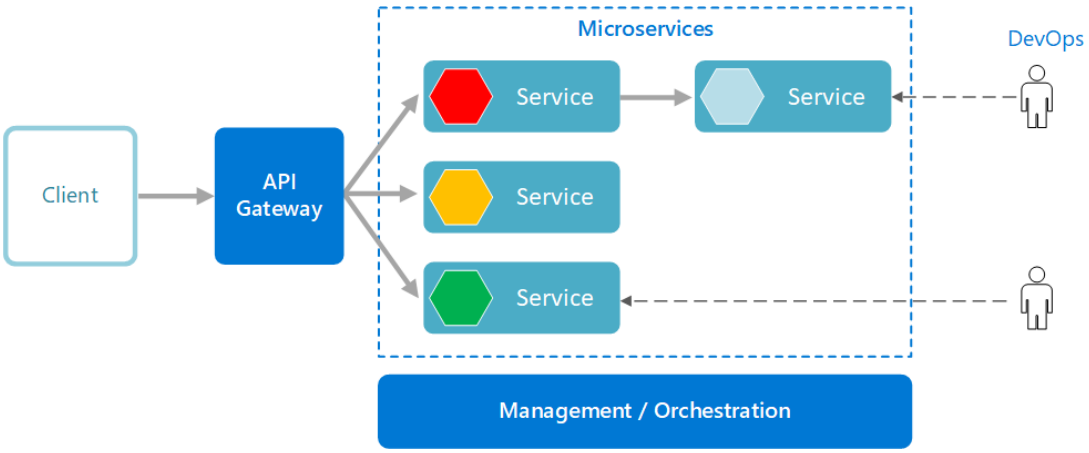
Arquitectura de Microservicios

- 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

Estilos de arquitectura

- **N Niveles**: es una 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 de 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.

Arquitectura de microservicios



Estilos de comunicación

- El cliente y los servicios, o los servicios entre si, pueden comunicarse a través de muchos tipos diferentes de comunicación, cada uno destinado a un escenario y unos objetivos distintos. Inicialmente, estos tipos de comunicaciones se pueden clasificar por dos criterios.
- El primer criterio define si el protocolo es sincrónico o asincrónico:
  - Protocolo síncrono: HTTP es el protocolo síncrono mas utilizado. El cliente envía una solicitud y espera una respuesta del servicio, solo puede continuar su tarea cuando recibe la respuesta del servidor. Es independiente de la ejecución de código de cliente, que puede ser síncrono (el subproceso está bloqueado) o asíncrono (el subproceso no está bloqueado y la respuesta dispara una devolución de llamada).
  - Protocolo asíncrono: Otros protocolos como AMQP (un protocolo compatible con muchos sistemas operativos y entornos de nube) usan mensajes asíncronos. Normalmente el código de cliente o el remitente del mensaje no espera ninguna respuesta. Simplemente se envía el mensaje a una cola de un agente de mensajes, que son escuchadas por los consumidores.
- El segundo criterio define si la comunicación tiene un único receptor o varios:
  - Receptor único: Cada solicitud debe ser procesada por un receptor o servicio exactamente (como en el patrón Command).
  - Varios receptores: Cada solicitud puede ser procesada por entre cero y varios receptores. Este tipo de comunicación debe ser asíncrona (basada en un bus de eventos o un agente de mensajes).

Estilos arquitectónicos

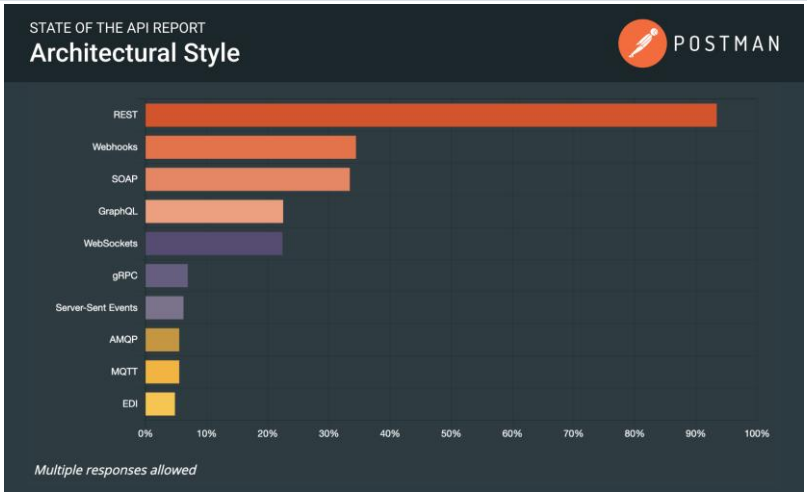
- Precursores:
  - RPC: Llamadas a Procedimientos Remotos
  - Binarios: CORBA, Java RMI, .NET Remoting
  - XML-RPC: Precursor del SOAP
- Actuales:
  - Servicios Web XML o Servicios SOAP
  - Servicios Web REST o API REST
    - WebHooks
  - Servicios GraphQL
  - Servicios gRPC

AÑO	Descripción
1976	Aparición de RPC (Remote Procedure Call) en Sistema Unix
1990	Aparición de DCE (Distributed Computing Environment) que es un sistema de software para computación distribuida, basado en RPC.
1991	Aparición de Microsoft RPC basado en DCE para sistemas Windows.
1992	Aparición de DCOM (Microsoft) y CORBA (ORB) para la creación de componentes software distribuidos.
1997	Aparición de Java RMI en JDK 1.1
1998	Aparición de XML-SOAP
1999	Aparición de SOAP 1.0, WSDL, UDDI
2000	Definición del REST
2012	Propuesta de GraphQL por Facebook
2015	Desarrollo de gGRPC por Google

Estilos arquitectónicos

- Basados en Recursos: Los servicios REST / Hypermedia exponen documentos que incluyen tanto identificadores de datos como de acciones (enlaces y formularios). REST es un estilo arquitectónico que separa las preocupaciones del consumidor y del proveedor de la API al depender de comandos que están integrados en el protocolo de red subyacente. REST (Representational State Transfer) es extremadamente flexible en el formato de sus cargas útiles de datos, lo que permite una variedad de formatos de datos populares como JSON y XML, entre otros.
- Basados en Procedimientos: Las llamadas a procedimiento remoto, o RPC, generalmente requieren que los desarrolladores ejecuten bloques específicos de código en otro sistema: operaciones. RPC es independiente del protocolo, lo que significa que tiene el potencial de ser compatible con muchos protocolos, pero también pierde los beneficios de usar capacidades de protocolo nativo (por ejemplo, almacenamiento en caché). La utilización de diferentes estándares da como resultado un acoplamiento más estrecho entre los consumidores y los proveedores de API y las tecnologías implicadas, lo que a su vez sobrecarga a los desarrolladores involucrados en todos los aspectos de un ecosistema de APIs impulsado por RPC. Los patrones de arquitectura de RPC se pueden observar en tecnologías API populares como SOAP, GraphQL y gRPC.
- Basados en Eventos/Streaming: a veces denominadas arquitecturas de eventos, en tiempo real, de transmisión, asíncronas o push, las APIs impulsadas por eventos no esperan a que un consumidor de la API las llame antes de entregar una respuesta. En cambio, una respuesta se desencadena por la ocurrencia de un evento. Estos servicios exponen eventos a los que los clientes pueden suscribirse para recibir actualizaciones cuando cambian los valores del servicio. Hay un puñado de variaciones para este estilo que incluyen (entre otras) reactivo, publicador/suscriptor, notificación de eventos y CQRS.

Estilos arquitectónicos mas utilizados



https://www.postman.com/state-of-api/api-technologies/#api-technologies

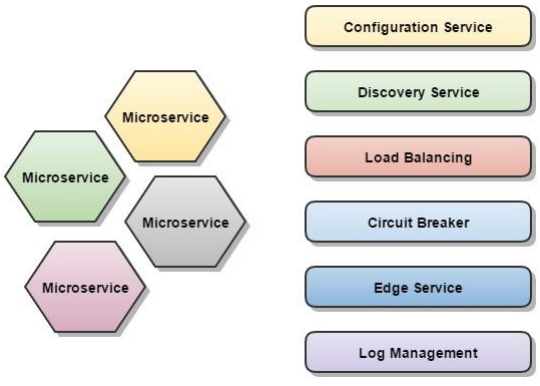
Preocupaciones transversales

- 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 …

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.

Modelo de referencia



Modelo de referencia

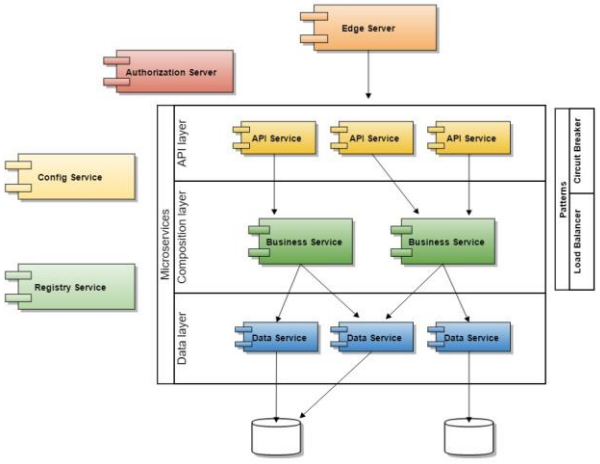
- Servidor perimetral / exposición de servicios (Edge server)
  - Será un gateway en el que se expondrán los servicios a consumir.
- Servicio de registro / descubrimiento
  - Este servicio centralizado será el encargado de proveer los endpoints de los servicios para su consumo. Todo microservicio, en 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).

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 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...

Modelo de referencia

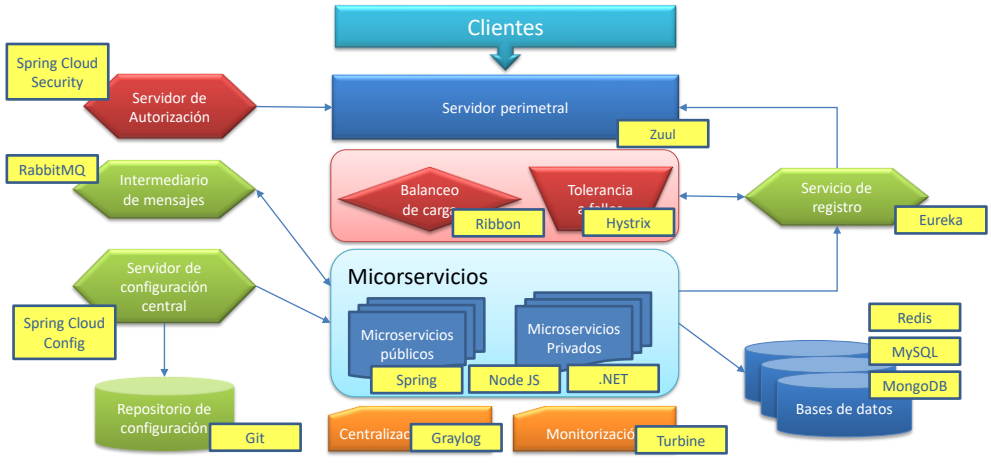
Modelo de referencia



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 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.

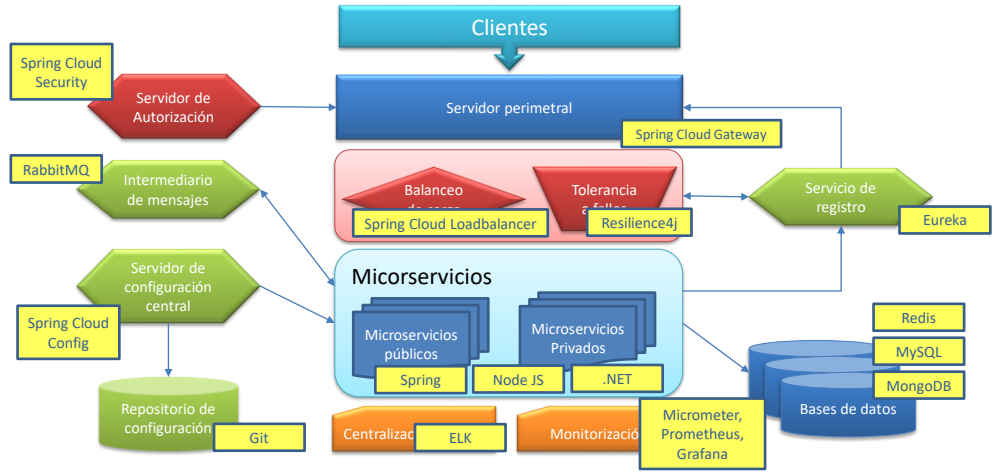
Modelo de implementación (Netflix OSS)



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 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.

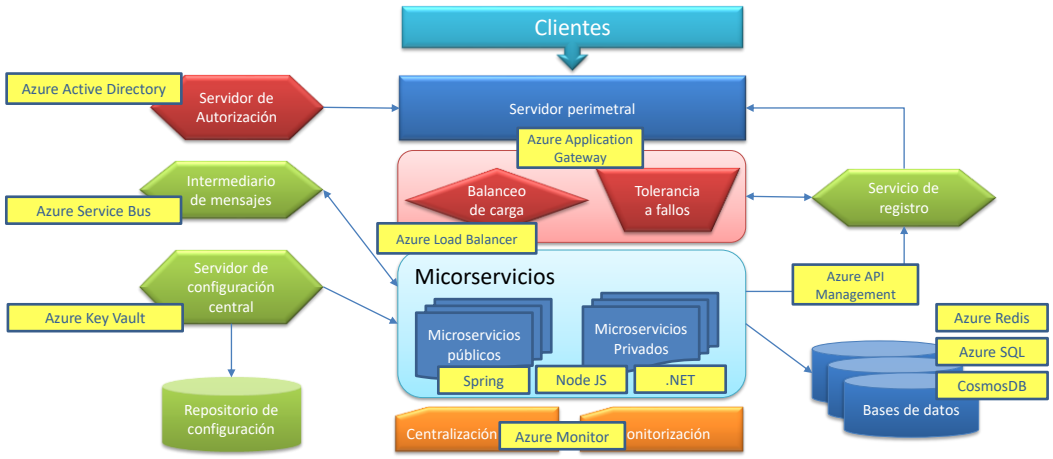
Modelo de implementación (Spring Cloud)



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 Active Directory (Azure AD) 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.

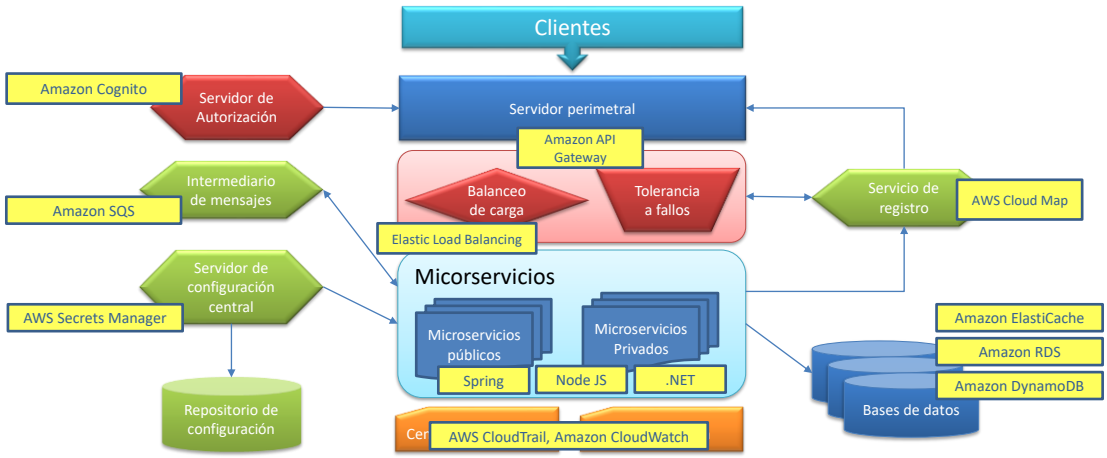
Modelo de implementación (Azure)



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

Modelo de implementación (AWS)



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 dos patrones en este sentido a la hora de encapsular microservicios:
  - Máquinas virtuales.
  - Contenedores.
- 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.

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.

Contenedores

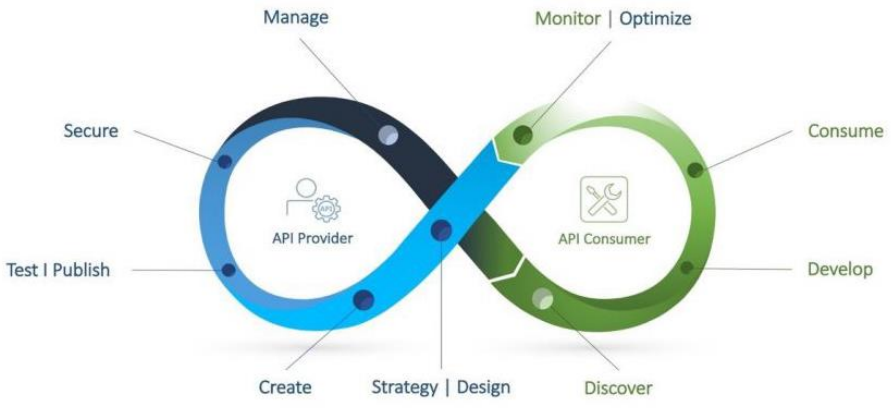
Modelo de despliegue

- 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.

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).

Ciclo de vida



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.

Ciclo de vida