Microservicios con Spring

Enlaces

- Microservicios
  - https://martinfowler.com/articles/microservices.html
  - https://microservices.io/
- Spring:
  - https://spring.io/projects
  - https://docs.spring.io/spring-boot/docs/current/reference/html/
- Spring Core
  - https://docs.spring.io/spring-framework/reference/core.html
- Spring Data
  - https://docs.spring.io/spring-data/commons/docs/current/reference/html/
  - https://docs.spring.io/spring-data/jpa/docs/current/reference/html/
  - https://docs.spring.io/spring-data/mongodb/docs/current/reference/html/
  - https://docs.spring.io/spring-data/redis/docs/current/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/current/reference/html/
- Spring Data REST
  - https://docs.spring.io/spring-data/rest/docs/current/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

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 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.

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.

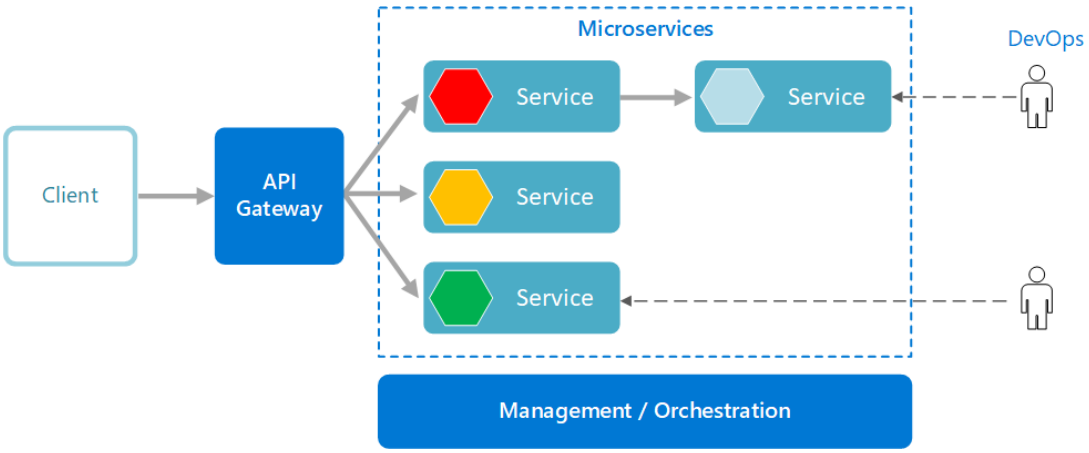
Microservicios: Inconvenientes

- 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.

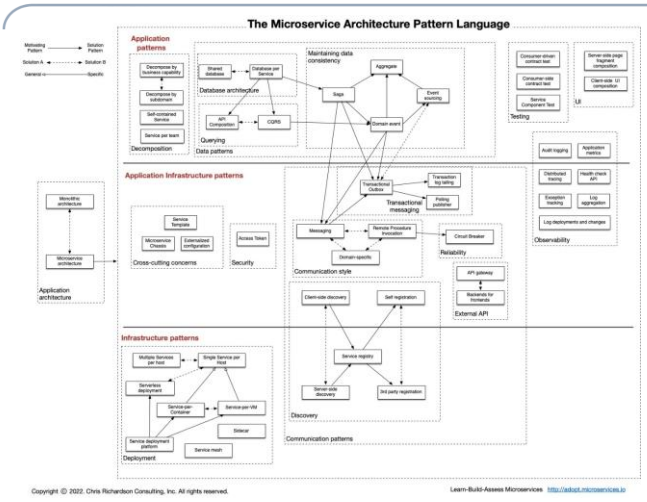
Arquitectura de Microservicios (Lewis/Fowler)

- 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

Arquitectura de microservicios



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

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 diferentes criterios.
- El primer criterio define si el proceso es sincrónico o asincrónico:
  - Proceso 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).
  - Proceso 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 1:1 (comando, punto a punto, P2P): Cada solicitud debe ser procesada por un receptor o servicio exactamente (como en el patrón Command).
  - Varios receptores 1:N (eventos, publicación/subscripción, Pub/Sub): 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).

Criterio según su base

- Basados en Recursos: Los servicios exponen información, documentos que incluyen tanto identificadores de datos como de acciones (enlaces y formularios), las operaciones CRUD están predefinidas. 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, con una información mínima, 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.

Evolución histórica de los servicios

- 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-RPC
1999	Aparición de SOAP 1.0, WSDL, UDDI
2000	Definición del REST
2012	Propuesta de GraphQL por Facebook
2015	Desarrollo de gRPC por Google

Servicios SOAP

- SOAP es un protocolo de comunicación web altamente estandarizado basado en formatos de tipo texto en XML. Publicado por Microsoft en 1999, un año después de XML-RPC, SOAP heredó mucho de él.
- Basado en operaciones, de tipos texto, en formato XML y comunicaciones síncronas.
- Ventajas:
  - Independiente del lenguaje y la plataforma: La funcionalidad incorporada para crear servicios basados en web permite a SOAP manejar las comunicaciones y hacer que las respuestas sean independientes del lenguaje y la plataforma.
  - Vinculado a una variedad de protocolos de transporte: SOAP es flexible en términos de protocolos de transferencia para adaptarse a múltiples escenarios.
  - Manejo de errores incorporado: La especificación de la API SOAP permite devolver el mensaje Retry XML con el código de error y su explicación.
  - Varias extensiones de seguridad: Integrado con los protocolos WS-Security, SOAP cumple con una calidad de transacción de nivel empresarial. Proporciona privacidad e integridad dentro de las transacciones al tiempo que permite el cifrado a nivel de mensaje.

Servicios SOAP

- Inconvenientes:
  - Solamente acepta formato XML: Los mensajes SOAP contienen una gran cantidad de metadatos y solo admiten estructuras XML detalladas para solicitudes y respuestas.
  - De peso pesado: Debido al gran tamaño de los archivos XML, los servicios SOAP requieren un gran ancho de banda hasta para la información mas nimia.
  - Conocimientos estrictamente especializados: La creación de servidores de API SOAP requiere un conocimiento profundo de todos los protocolos involucrados y sus reglas altamente restringidas o disponer de framework que simplifiquen su utilización que no están disponibles en todas las plataformas y lenguajes.
  - Actualización de mensajes tediosa: Al requerir un esfuerzo adicional para agregar o eliminar las propiedades del mensaje, el esquema SOAP rígido ralentiza la adopción.

Servicios REST

- REpresentational State Transfer es un estilo arquitectónico autoexplicativo basado en el uso del protocolo HTTP e hipermedia y establece un conjunto de restricciones arquitectónicas y destinado a una amplia adopción. Definido en el 2000 por Roy Fielding, para no reinventar la rueda, se basa en aprovechar lo que ya estaba definido en el HTTP pero que no se utilizaba. RESTful hace referencia a un servicio web que implementa la arquitectura REST.
- Restringe la forma de usar de las URLs, los métodos (verbos) de HTTP, sus encabezados (Accept, Content-type, …) y sus códigos de estado.
- Basado en recursos, independiente de formato (texto o binario) y comunicaciones síncronas.
- Ventajas:
  - Soporte de múltiples formatos: La capacidad de admitir múltiples formatos (a menudo JSON y XML) para almacenar e intercambiar datos es una de las razones por las que REST es actualmente una opción predominante para crear APIs públicas.
  - Documentación mínima, basados en convenios y el descubrimiento hipermedia.
  - Mínima curva de aprendizaje: usa tecnologías ampliamente conocidas: HTTP, URL, MIME, …

Servicios REST

- Inconvenientes:
  - Uso de HTTP: Aunque es la infraestructura mas ampliamente difundida y utilizada, está restringido a ella.
  - Grandes cargas útiles: REST devuelve una gran cantidad de metadatos enriquecidos para que el cliente pueda comprender todo lo necesario sobre el estado de la aplicación solo a partir de sus respuestas.
  - Sin estructura REST única: No existe una forma exacta y correcta de crear una API REST. Cómo modelar los recursos y qué recursos modelar dependerá de cada escenario. Esto hace que REST sea simple en teoría, pero difícil en la práctica.
  - Problemas de recuperación excesiva o insuficiente: Devolver todo suele conllevar demasiados datos y el convenio no estable mecanismos de filtrado, paginación o proyecciones. El uso de hipermedia para obtener datos relacionados requiere solicitudes adicionales y encadenadas.
  - Convenio genérico: En muchos casos no acorde con las reglas de negocio (create o replace, delete, …) lo que acaba requiriendo documentación.

Servicios GraphQL

- GraphQL es una sintaxis que describe cómo obtener la descripción del modelo de datos y cómo realizar una solicitud de datos precisa, consultas y mutaciones, pensada para los modelos de datos con muchas entidades complejas que hacen referencia entre sí. Fue propuesta por Facebook en 2012, publicada en 2015 y posteriormente pasada a open source en 2018.
- Basado en consultas, en formato JSON y comunicaciones síncronas.
- Ventajas:
  - Un esquema de GraphQL establece una fuente única de información, ofrece una forma de unificar todo en un único servicio.
  - Las llamadas a GraphQL se gestionan mediante HTTP POST en un solo recorrido de ida y vuelta. Los clientes obtienen lo que solicitan sin que se genere una sobrecarga.
  - Los tipos de datos bien definidos reducen los problemas de comunicación entre el cliente y el servidor. Un cliente puede solicitar una lista de los tipos de datos disponibles y esto es ideal para la generación automática de documentación.
  - GraphQL permite que las APIs de las aplicaciones evolucionen sin afectar a las consultas actuales.
  - GraphQL no exige una arquitectura de aplicación específica. Puede incorporarse sobre una API de REST actual y funcionar con las herramientas de gestión de APIs actuales. Hay muchas extensiones open source de GraphQL que ofrecen características que no están disponibles con las APIs de REST.

Servicios GraphQL

- Inconvenientes:
  - GraphQL intercambia complejidad por flexibilidad. Tener demasiados campos anidados en una solicitud puede provocar una sobrecarga del sistema. Además, delega gran parte del trabajo de las consultas de datos en el servidor, lo cual representa una mayor complejidad para los desarrolladores de servidores.
  - GraphQL no indica cómo almacenar los datos ni qué lenguaje de programación utilizar, requiere disponer de framework que simplifiquen su utilización que no están disponibles en todas las plataformas.
  - Como GraphQL no utiliza la semántica de almacenamiento en caché HTTP, requiere un esfuerzo de almacenamiento en caché personalizado.
  - GraphQL presenta una curva de aprendizaje elevada para desarrolladores que tienen experiencia con las APIs de REST.

Servicios gRPC

- gRPC es un marco de llamada a procedimiento remoto (RPC) de alto rendimiento e independiente de lenguaje. Desarrollado por Google en el año 2015, y luego convertido en código abierto. Como GraphQL, es una especificación que se implementa en una variedad de lenguajes.
- Basado en contratos, en el formato binario Protocol Buffers y comunicaciones síncronas/asíncronas.
- Las principales ventajas de gRPC son:
  - Marco de RPC moderno, ligero y de alto rendimiento.
  - La especificación gRPC es preceptiva con respecto al formato que debe seguir un servicio gRPC, formaliza un contrato independiente de lenguaje.
  - Es compatible con el intercambio de datos bidireccional y asíncrono al estar basado en HTTP/2, así como con la compresión, multiplexación y streaming.
  - Uso reducido de red al usar un formato de mensaje binario eficaz que genera cargas de mensajes pequeñas.

Servicios gRPC

- Inconvenientes:
  - Tanto el cliente como el servidor deben admitir la misma especificación de búfer de protocolo, requiere un estricto control de versiones. gRPC es un formato muy particular que proporciona una ejecución ultrarrápida a expensas de la flexibilidad.
  - Basado en HTTP/2 que no tiene soporte universal para interacciones cliente-servidor de cara al público general en Internet y una compatibilidad limitada con exploradores.
  - gRPC es una especificación, por lo que requiere disponer de framework que permitan su utilización tanto en el cliente como en le servidor y que no están disponibles en todas las plataformas y lenguajes.
  - El proceso de serialización/deserialización para su conversión a binario es costoso en términos de CPU.
  - gRPC presenta una curva de aprendizaje mas empinada que la de REST.

Tecnología de clientes web (Navegadores)

- JavaScript Asíncrono + XML (AJAX) no es una tecnología por sí misma, es un término que describe un nuevo modo de utilizar conjuntamente varias tecnologías existentes. Esto incluye: HTML o XHTML, CSS, JavaScript, DOM, XML, XSLT, y lo más importante, el objeto XMLHttpRequest (fetch). Cuando estas tecnologías se combinan en un modelo AJAX, es posible lograr aplicaciones web capaces de actualizarse continuamente sin tener que volver a cargar la página completa.
- WebSockets es una tecnología avanzada que hace posible abrir una sesión de comunicación bidireccional entre el navegador y un servidor: se puede enviar mensajes a un servidor y recibir respuestas controladas por eventos sin tener que consultar al servidor para una respuesta.
- Server Sent Events (SSE) es una tecnología basada en streaming HTTP que pretenden estandarizar la comunicación COMET (long polling) en los navegadores cuyo flujo de comunicación sólo va desde el servidor hacia el cliente (no es bidireccional como los WebSockets). La idea consiste en que el cliente crea una conexión con el servidor una sola vez y este le va enviando información (eventos) al cliente cuando hay nuevos datos.
- ASP.NET Signalr es una biblioteca para desarrolladores de ASP.NET que simplifica el proceso de agregar funcionalidad web en tiempo real: la posibilidad de que el código de servidor inserte el contenido en los clientes conectados al instante a medida que esté disponible sin espera a nuevas solicitudes. Signalr usa WebSocket cuando está disponible o recurre las otras cuando es necesario.

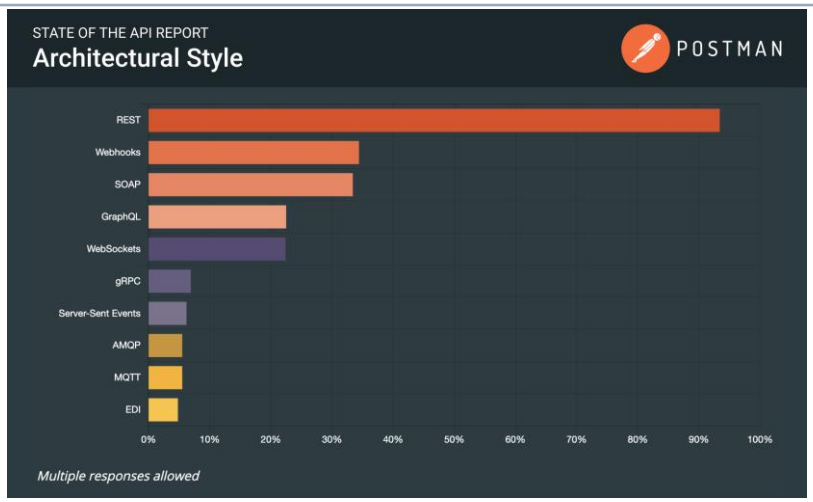
WebHooks

- Los webhooks son eventos que desencadenan acciones. Su nombre se debe a que funcionan como «ganchos» de los programas en Internet y casi siempre se utilizan para la comunicación entre sistemas. Son la manera más sencilla de obtener un aviso cuando algo ocurre en otro sistema y para el intercambio de datos entre aplicaciones web.
- Un webhook es una retro llamada HTTP, una solicitud HTTP POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento en JSON o XML) a una determinada dirección http:// o https://, que puede:
  - almacenar los datos del evento en JSON o XML
  - generar una respuesta que permita actualizarse al sistema donde se produce el evento
  - ejecutar un proceso en el sistema receptor del evento (Ej: enviar un correo electrónico)
- Los webhook están pensados para su utilización desde páginas web y sus diferentes consumidores: navegadores, correo electrónico, webapps, …
- Un ejemplo típico es su utilización en correos electrónico de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario a abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo POST).

Protocolos y Estándares

- Síncronos:
  - SOAP: Basado en operaciones, de tipos texto, en formato XML y comunicaciones síncronas.
  - REST: Basado en recursos, independiente de formato (texto/binario) y comunicaciones síncronas.
  - GraphQL: Basado en consultas, en formato JSON y comunicaciones síncronas.
  - gRPC: Basado en contratos, en el formato binario Protocol Buffers y comunicaciones síncronas/asíncronas.
- Asíncronos:
  - WebSockets: protocolo estándar abierto, elemental, texto y binario.
  - STOMP (Simple/Streaming Text Oriented Messaging Protocol): protocolo estándar abierto, basado en texto, soluciones simples y ligero.
  - AMQP (Advanced Message Queuing Protocol): protocolo estándar abierto, binario, encolamiento, P2P y Pub/Sub, exactitud y seguridad
  - MQTT (Message Queue Telemetry Transport): protocolo estándar abierto, binario, P2P, liviano, soluciones simples y seguridad
  - JMS (Java Message Service): API, binario Java, P2P y Pub/Sub

Estilos arquitectónicos mas utilizados



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

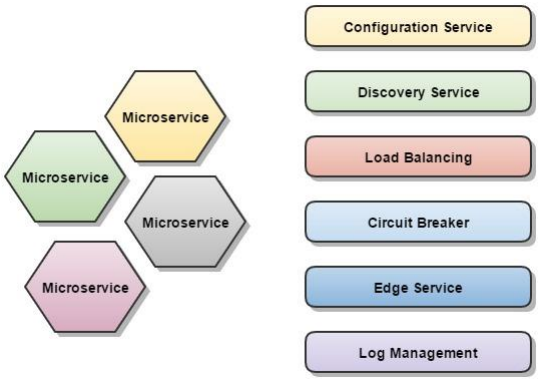
Preocupaciones transversales

- 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

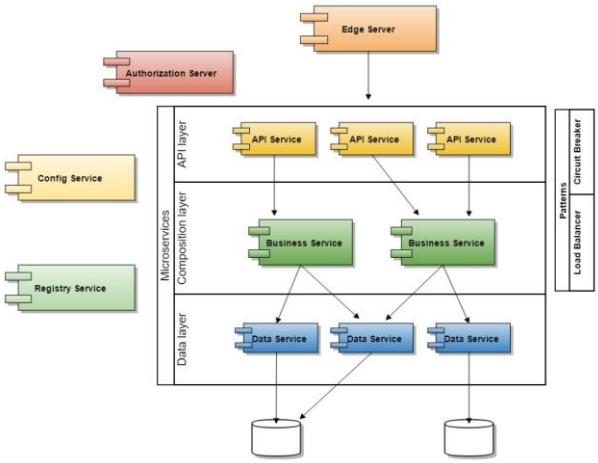
- Servidor perimetral / exposición de servicios (Edge server)
  - Será un gateway en el que se expondrán los servicios a consumir.
- Servicio de registro / descubrimiento
  - Este servicio centralizado será el encargado de proveer los endpoints de los servicios para su consumo. Todo microservicio, en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

Modelo de referencia

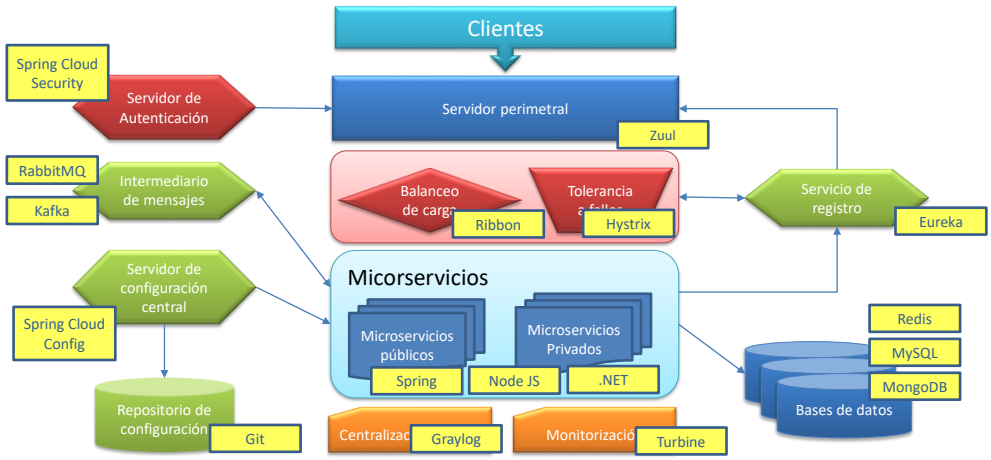
Modelo de referencia



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 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.

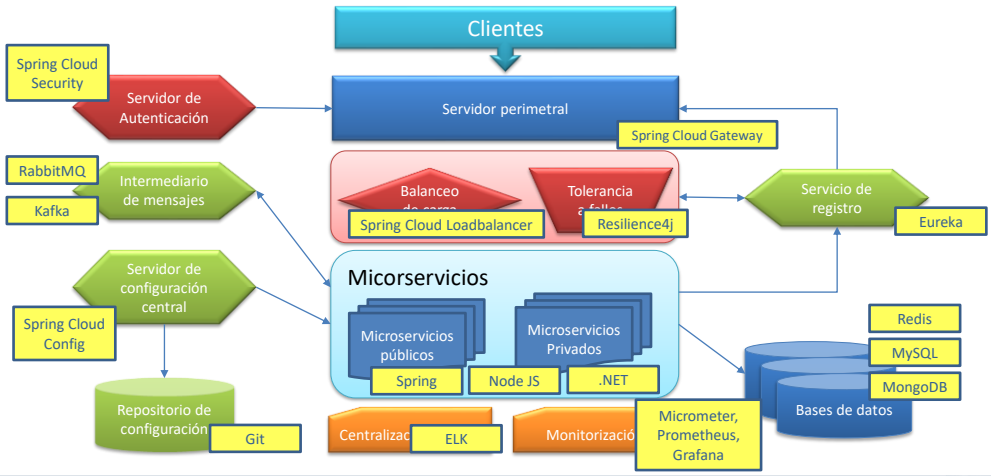
Modelo de implementación (Netflix OSS)



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 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.

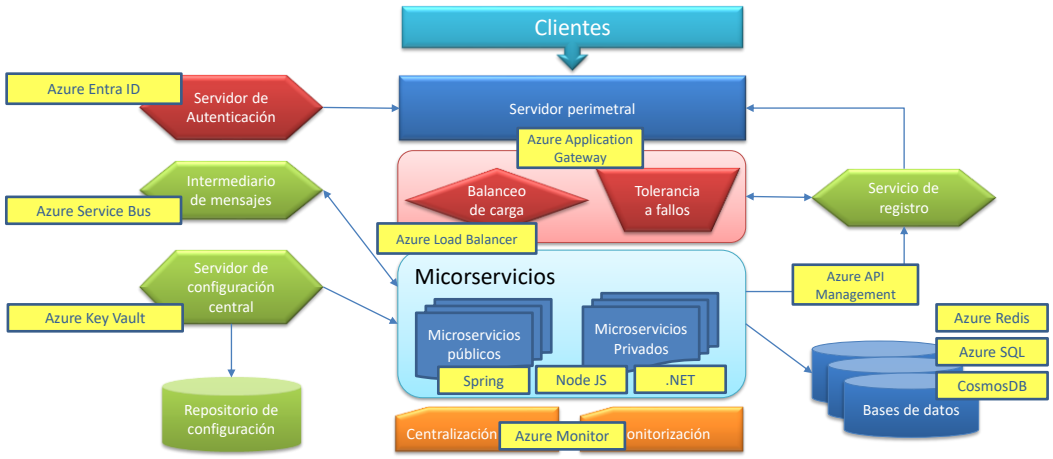
Modelo de implementación (Spring Cloud)



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 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.

Modelo de implementación (Azure)

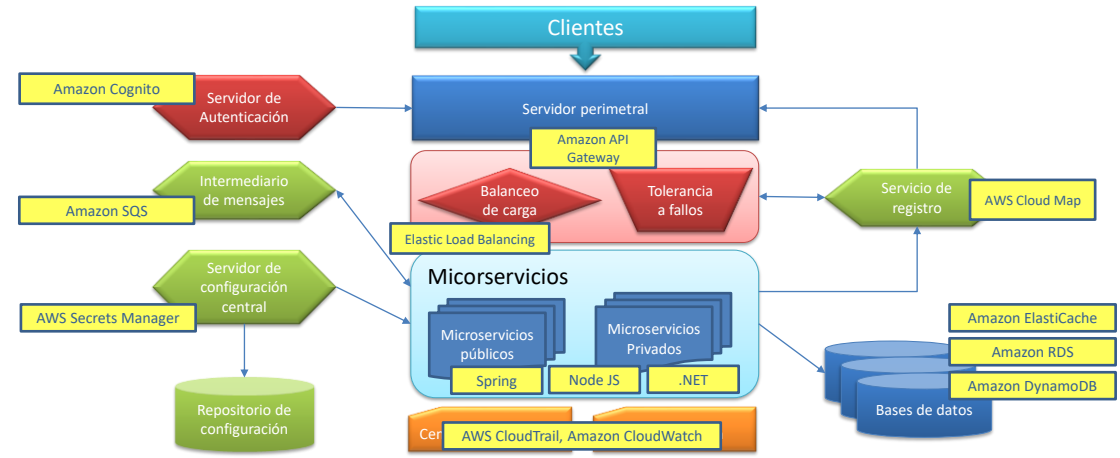


Modelo de implementación (AWS)

- 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

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

Modelo de implementación (AWS)



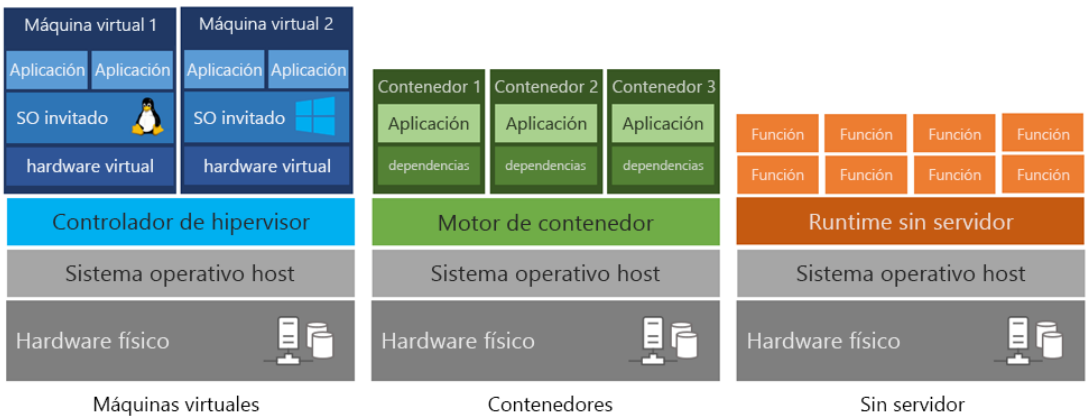
Modelo de despliegue

- 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 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.

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



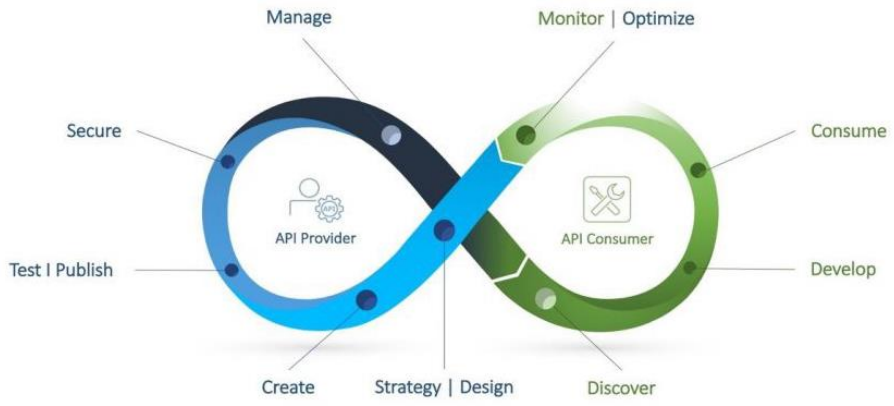
Modelo de despliegue

- 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

¿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 versionadas 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.

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 despido. Gracias, ¡pasad un buen día!”. Desde hace ya varios años Amazon es el primer proveedor IaaS mundial distanciado significativamente de sus competidores.

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.

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.

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.

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

Ciclo de vida de las API

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

http://spring.io

SPRING CON SPRING BOOT

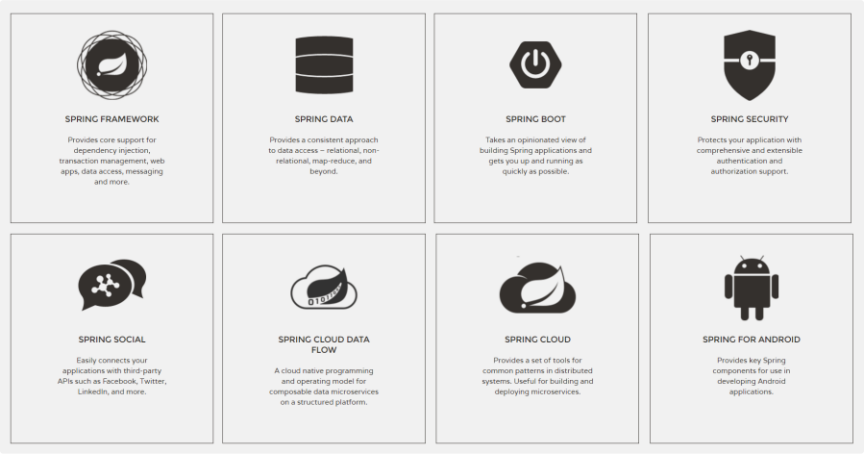
Spring

- Spring Framework es un marco de desarrollo de aplicaciones Java que facilita la creación de aplicaciones empresariales robustas y modulares al proporcionar un conjunto integral de funcionalidades, como inversión de control, contenedor de beans o gestión de transacciones. Utilizar Spring y Spring Boot proporciona ventajas como la flexibilidad, la modularidad, la escalabilidad y una amplia gama de características integradas que aceleran el desarrollo de aplicaciones empresariales.
- Inicialmente era un ejemplo hecho para el libro “J2EE design and development” de Rod Johnson en 2003, que defendía alternativas a la “visión oficial” de aplicación JavaEE basada en EJBs. Actualmente es un framework open source que facilita el desarrollo de aplicaciones java JEE & JSE (no esta limitado a aplicaciones Web ni a java).
- Provee de un contenedor encargado de manejar el ciclo de vida de los objetos (beans) para que los desarrolladores se enfoquen a la lógica de negocio. Permite integración con diferentes frameworks. Surge como una alternativa a EJB’s
- Actualmente es un framework completo compuesto por múltiples módulos/proyectos que cubre todas las capas de la aplicación, con decenas de desarrolladores y miles de descargas al día.

Características

- Ligero
  - No se refiere a la cantidad de clases sino al mínimo impacto que se tiene al integrar Spring.
- No intrusivo
  - Generalmente los objetos que se programan no tienen dependencias de clases específicas de Spring
- Flexible
  - Aunque Spring provee funcionalidad para manejar las diferentes capas de la aplicación (vista, lógica de negocio, acceso a datos) no es necesario usarlo para todo. Brinda la posibilidad de utilizarlo en la capa o capas que queramos.
- Multiplataforma
  - Escrito en Java, corre sobre JVM

Proyectos



Módulos necesarios

- Spring Framework
  - Spring Core
    - Contenedor IoC (inversión de control) - inyector de dependencia
  - Spring MVC
    - Framework basado en MVC para aplicaciones web y servicios REST
- Spring Data
  - Simplifica el acceso a los datos: JPA, bases de datos relacionales / NoSQL, nube
- Spring Boot
  - Simplifica el desarrollo de Spring: inicio rápido con menos codificación

Spring Boot

- Spring tiene una gran cantidad de módulos que implican multitud de configuraciones. Estas configuraciones pueden requerir mucho tiempo, pueden ser desconocidas para principiantes y suelen ser repetitivas.
- La solución de Spring es Spring Boot, que aplica el concepto de Convention over Configuration (CoC).
- CoC es un paradigma de programación que minimiza las decisiones que tienen que tomar los desarrolladores, simplificando tareas.
- No obstante, la flexibilidad no se pierde, ya que a pesar de otorgar valores por defecto, siempre se puede configurar de forma extendida.
- De esta forma se evita la repetición de tareas básicas a la hora de construir un proyecto.
- Spring Boot es una herramienta que nace con la finalidad de simplificar aun más el desarrollo de aplicaciones basadas en el framework Spring Core: que el desarrollador solo se centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.

Spring Boot

- Resolución de dependencias:
  - Con Spring Boot solo hay que determinar que tipo de proyecto estaremos utilizando y el se encarga de resolver todas las librerías/dependencias para que la aplicación funcione.
- Configuración:
  - Spring Boot cuenta con un complejo módulo que autoconfigura todos los aspectos de nuestra aplicación para poder simplemente ejecutar la aplicación, sin tener que definir absolutamente nada.
- Despliegue:
  - Spring Boot se puede ejecutar como una aplicación Stand-alone, pero también es posible ejecutar aplicaciones web, ya que es posible desplegar las aplicaciones mediante un servidor web integrado, como es el caso de Tomcat, Jetty o Undertow.

Spring Boot

- Métricas:
  - Por defecto, Spring Boot cuenta con servicios que permite consultar el estado de salud de la aplicación, permitiendo saber si la aplicación está encendida o apagada, memoria utilizada y disponible, número y detalle de los Bean's creado por la aplicación, controles para el prendido y apagado, etc.
- Extensible:
  - Spring Boot permite la creación de complementos, los cuales ayudan a que la comunidad de Software Libre cree nuevos módulos que faciliten aún más el desarrollo.
- Productividad:
  - Herramientas de productividad para desarrolladores como Spring Initializr, Lombok, LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

Dependencias: starters

- Los starters son un conjunto de descriptores de dependencias convenientes (versiones compatibles, ya probadas) que se pueden incluir en la aplicación.
- Se obtiene una ventanilla única para el módulo de Spring y la tecnología relacionada que se necesita, sin tener que buscar a través de códigos de ejemplo y copiar/pegar cargas de descriptores de dependencia.
- Por ejemplo, si desea comenzar a utilizar Spring con JPA para un acceso CRUD a base de datos, basta con incluir la dependencia spring-boot-starter-data-jpa en el proyecto.

devtools

- Al realizar nuevos cambios en nuestra aplicación, podemos hacer que el arranque se reinicie automáticamente. Para eso es necesario incluir una dependencia Maven extra: spring-boot-devtools.
- Durante el tiempo de ejecución, Spring Boot supervisa la carpeta que se encuentra en classpath (en maven, las carpetas que están en la carpeta "target"). Solo necesitamos activar la compilación de las fuentes en los cambios que causarán la actualización de la carpeta 'destino' y Spring Boot reiniciará automáticamente la aplicación. Si estamos utilizando Eclipse IDE, la acción de guardar puede desencadenar la compilación.
- El módulo spring-boot-devtools incluye un servidor LiveReload incorporado que activa una actualización del navegador cuando se cambia un recurso. Las extensiones del navegador LiveReload están disponibles gratuitamente para Chrome, Firefox y Safari desde:
  http://livereload.com/extensions/
- Para configurar LiveReload:

```
spring.devtools.restart.additional-paths=META-INF/resources/**
spring.devtools.livereload.enabled=true # por defecto
```

Spring Tools

https://spring.io/tools

- Spring Tools 4 es la próxima generación de herramientas Spring para los entorno IDE de codificación favoritos. Proporciona soporte de primera clase para el desarrollo de aplicaciones empresariales basadas en Spring, ya sea que se prefiera Eclipse, Visual Studio Code o Theia IDE.
  - Help → Eclipse Marketplace …
    - Spring Tools 4 for Spring Boot
- Spring Tool Suite (STS) es un IDE basado en la versión Java EE de Eclipse, pero altamente personalizable para trabajar con Spring Framework.
  - IDE gratuito, personalización del Eclipse

Crear proyecto

- Desde web:
  - https://start.spring.io/
  - Descomprimir en el workspace
  - Import → Maven → Existing Maven Project
- Desde Eclipse:
  - New Project → Sprint Boot → Spring Started Project
- Dependencias
  - Web
  - JPA
  - JDBC (o proyecto personalizado)

Application

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class DemoApplication implements CommandLineRunner {

	public static void main(String[] args) {
		SpringApplication.run(ApiHrApplication.class, args);
	}

	@Override
	public void run(String... args) throws Exception {
		// Opcional: Procesar los args una vez arrancado SprintBoot

	}

}
```


Configuración

- @Configuration: Indica que esta es una clase usada para configurar el contenedor Spring.
- @ComponentScan: Escanea los paquetes de nuestro proyecto en busca de los componentes que hayamos creado, ellos son, las clases que utilizan las siguientes anotaciones: @Component, @Service, @Controller, @Repository.
- @EnableAutoConfiguration: Habilita la configuración automática, esta herramienta analiza el classpath y el archivo application.properties para configurar nuestra aplicación en base a las librerías y valores de configuración encontrados, por ejemplo: al encontrar el motor de bases de datos H2 la aplicación se configura para utilizar este motor de datos, al encontrar Thymeleaf se crearan los beans necesarios para utilizar este motor de plantillas para generar las vistas de nuestra aplicación web.
- @SpringBootApplication: Es el equivalente a utilizar las anotaciones: @Configuration, @EnableAutoConfiguration y @ComponentScan

Configuración

- Editar src/main/resources/application.properties:

```
server.port=${PORT:8080}
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

# MySQL settigs
spring.datasource.url=jdbc:mysql://localhost:3306/sakila
spring.datasource.username=root
spring.datasource.password=root
#spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
logging.level.org.hibernate.SQL=debug
```

- Repetir con src/test/resources/application.properties
- Eclipse: Run Configurations → Arguments → VM Arguments: -DPORT=8888

Instalación de MySQL

- Descargar e instalar:
  - https://mariadb.org/download/
- Incluir en la seccion [mysqld] de %MYSQL_ROOT%/data/my.ini
  - default_time_zone='+01:00'
- Descargar bases de datos de ejemplos:
  - https://dev.mysql.com/doc/index-other.html
- Instalar bases de datos de ejemplos:
  - mysql -u root -p < employees.sql
  - mysql -u root -p < sakila-schema.sql
  - mysql -u root -p < sakila-data.sql

Instalación con Docker

- Docker Toolbox
  - Windows 10 Pro ++: https://docs.docker.com/docker-for-windows/install/
  - Otras: https://github.com/docker/toolbox/releases
- Ejecutar Docker QuickStart
- Para crear el contenedor de MySQL con la base de datos Sakila:
  - docker run -d --name mysql-sakila -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 jamarton/mysql-sakila
- Para crear el contenedor de MongoDB:
  - docker run -d --name mongodb -p 27017:27017 mongo
- Para crear el contenedor de Redis:
  - docker run --name redis -p 6379:6379 -d redis
  - docker run --name redis-insight -d -v redisinsight:/db -p 6380:8001 redislabs/redisinsight:latest
  - docker run -d --name redis-commander -p 8081:8081 rediscommander/redis-commander:latest
- Para crear el contenedor de RabbitMQ:
  - docker run -d --hostname rabbitmq --name rabbitmq -p 4369:4369 -p 5671:5671 -p 5672:5672 -p 15671:15671 -p 15672:15672 -p 25672:25672 -e RABBITMQ_DEFAULT_USER=admin -e RABBITMQ_DEFAULT_PASS=curso rabbitmq:management-alpine

Estilos de comunicación

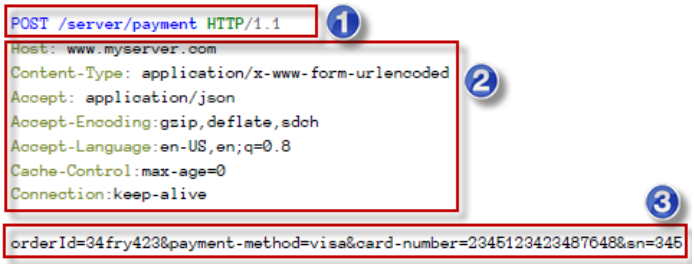
SERVICIOS REST

REST (REpresentational State Transfer)

- En 2000, Roy Fielding propuso la transferencia de estado representacional (REST) como enfoque de arquitectura para el diseño de servicios web. REST **es un estilo de arquitectura** para la creación de sistemas distribuidos basados en hipermedia. REST es independiente de cualquier protocolo subyacente y no está necesariamente unido a HTTP. Sin embargo, en las implementaciones más comunes de REST se usa HTTP como protocolo de aplicación, y esta guía se centra en el diseño de API de REST para HTTP.
- Originalmente se basaba en lo que ya estaba disponible en HTTP:
  - URL como identificadores de recursos
  - HTTP ya define 8 métodos (algunas veces referidos como "verbos") que indica la acción que desea que se efectúe sobre el recurso identificado: HEAD, GET, POST, PUT, DELETE, TRACE, OPTIONS, CONNECT + PATCH (HTTP1.1)
  - HTTP permite transmitir en el encabezado la información de comportamiento: Content-type, Accept, Authorization, Cache-control, …
  - HTTP utiliza códigos de estado en la respuesta para indicar como se ha completado una solicitud HTTP específica: respuestas informativas (1xx), respuestas satisfactorias (2xx), redirecciones (3xx), Errores en la petición (4xx) y errores de los servidores (5xx).

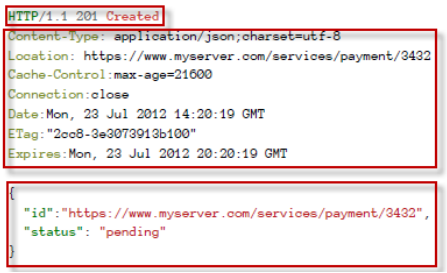
Petición HTTP

- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después el cuerpo del mensaje, que contiene un documento, que puede estar en cualquier formato (XML, HTML, JSON → Content-type)



Respuesta HTTP

- Los mensajes HTTP de respuesta siguen el mismo formato que los de envío.
- Sólo difieren en la primera línea
  - Donde se indica un código de respuesta junto a una explicación textual de dicha respuesta.
  - El código de respuesta indica si la petición tuvo éxito o no.



Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único, cualquier tipo de objeto, dato o servicio que sea direccionable a través de internet.
- Un recurso es un objeto que es lo suficientemente importante como para ser referenciado por sí mismo. Un recurso tiene datos, relaciones con otros recursos y métodos que operan contra él para permitir el acceso y la manipulación de la información asociada. Un grupo de recursos se llama colección. El contenido de las colecciones y los recursos depende de los requisitos de la organización y de los consumidores.
- En REST todos los recursos comparten una interfaz única y constante, la URI. (https://...)
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE
- Normalmente estos recursos son accesibles en una red o sistema.
- Las URI son el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Para que un URI sea correcto, debe cumplir los requisitos de formato, REST no indica de forma específica un formato obligatorio.
  - <esquema>://<host>:puerto/<ruta><querystring><fragmento>
- Los URI asociados a los recursos pueden cambiar si modificamos el recurso (nombre, ubicación, características, etc)

Tipos MIME

- Otro aspecto muy importante es la posibilidad de negociar distintos formatos (representaciones) a usar en la transferencia del estado entre servidor y cliente (y viceversa). La representación de los recursos es el formato de lo que se envía un lado a otro entre clientes y servidores. Como REST utiliza HTTP, podemos transferir múltiples tipos de información.
- Los datos se transmiten a través de TCP/IP, el navegador sabe cómo interpretar las secuencias binarias (CONTENT-TYPE) por el protocolo HTTP.
- La representación de un recurso depende del tipo de llamada que se ha generado (Texto, HTML, PDF, etc).
- En HTTP cada uno de estos formatos dispone de su propio tipos MIME, en el formato <tipo>/<subtipo>.
  - application/json application/xml text/html text/plain image/jpeg
- Para negociar el formato:
  - El cliente, en la cabecera ACCEPT, envía una lista priorizada de tipos MIME que entiende.
  - Tanto cliente como servidor indican en la cabecera CONTENT-TYPE el formato MIME en que está codificado el body.
- Si el servidor no entiende ninguno de los tipos MIME propuestos (ACCEPT) devuelve un mensaje con código 406 (incapaz de aceptar petición).

Métodos HTTP

HTTP	REST	Descripción
GET	RETRIEVE	Sin identificador: Recuperar el estado completo de un recurso (HEAD + BODY) Con identificador: Recuperar el estado individual de un recurso (HEAD + BODY)
HEAD		Recuperar la cabecera del estado de un recurso (HEAD)
POST	CREATE or REPLACE	Crea o modifica un recurso (sin identificador)
PUT	CREATE or REPLACE	Crea o modifica un recurso (con identificador)
DELETE	DELETE	Sin identificador: Elimina todo el recurso Con identificador: Elimina un elemento concreto del recurso
CONNECT		Comprueba el acceso al host
TRACE		Solicita al servidor que introduzca en la respuesta todos los datos que reciba en el mensaje de petición
OPTIONS		Devuelve los métodos HTTP que el servidor soporta para un URL específico
PATCH	REPLACE	HTTP 1.1 Reemplaza parcialmente un elemento del recurso

Códigos HTTP (status)

status	statusText	Descripción
100	Continue	Una parte de la petición (normalmente la primera) se ha recibido sin problemas y se puede enviar el resto de la petición
101	Switching protocols	El servidor va a cambiar el protocolo con el que se envía la información de la respuesta. En la cabecera Upgrade indica el nuevo protocolo
200	OK	La petición se ha recibido correctamente y se está enviando la respuesta. Este código es con mucha diferencia el que mas devuelven los servidores
201	Created	Se ha creado un nuevo recurso (por ejemplo una página web o un archivo) como parte de la respuesta
202	Accepted	La petición se ha recibido correctamente y se va a responder, pero no de forma inmediata
203	Non-Authoritative Information	La respuesta que se envía la ha generado un servidor externo. A efectos prácticos, es muy parecido al código 200
204	No Content	La petición se ha recibido de forma correcta pero no es necesaria una respuesta
205	Reset Content	El servidor solicita al navegador que inicialice el documento desde el que se realizó la petición, como por ejemplo un formulario
206	Partial Content	La respuesta contiene sólo la parte concreta del documento que se ha solicitado en la petición

Códigos de redirección

status	statusText	Descripción
300	Multiple Choices	El contenido original ha cambiado de sitio y se devuelve una lista con varias direcciones alternativas en las que se puede encontrar el contenido
301	Moved Permanently	El contenido original ha cambiado de sitio y el servidor devuelve la nueva URL del contenido. La próxima vez que solicite el contenido, el navegador utiliza la nueva URL
302	Found	El contenido original ha cambiado de sitio de forma temporal. El servidor devuelve la nueva URL, pero el navegador debe seguir utilizando la URL original en las próximas peticiones
303	See Other	El contenido solicitado se puede obtener en la URL alternativa devuelta por el servidor. Este código no implica que el contenido original ha cambiado de sitio
304	Not Modified	Normalmente, el navegador guarda en su caché los contenidos accedidos frecuentemente. Cuando el navegador solicita esos contenidos, incluye la condición de que no hayan cambiado desde la última vez que los recibió. Si el contenido no ha cambiado, el servidor devuelve este código para indicar que la respuesta sería la misma que la última vez
305	Use Proxy	El recurso solicitado sólo se puede obtener a través de un proxy, cuyos datos se incluyen en la respuesta
307	Temporary Redirect	Se trata de un código muy similar al 302, ya que indica que el recurso solicitado se encuentra de forma temporal en otra URL

Códigos de error en la petición

status	statusText	Descripción
400	Bad Request	El servidor no entiende la petición porque no ha sido creada de forma correcta
401	Unauthorized	El recurso solicitado requiere autorización previa
402	Payment Required	Código reservado para su uso futuro
403	Forbidden	No se puede acceder al recurso solicitado por falta de permisos o porque el usuario y contraseña indicados no son correctos
404	Not Found	El recurso solicitado no se encuentra en la URL indicada. Se trata de uno de los códigos más utilizados y responsable de los típicos errores de *Página no encontrada*
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

Códigos de error en la petición

status	statusText	Descripción
409	Conflict	El navegador no puede procesar la petición, ya que implica realizar una operación no permitida (como por ejemplo crear, modificar o borrar un archivo)
410	Gone	Similar al código 404. Indica que el recurso solicitado ha cambiado para siempre su localización, pero no se proporciona su nueva URL
411	Length Required	El servidor no procesa la petición porque no se ha indicado de forma explícita el tamaño del contenido de la petición
412	Precondition Failed	No se cumple una de las condiciones bajo las que se realizó la petición
413	Request Entity Too Large	La petición incluye más datos de los que el servidor es capaz de procesar. Normalmente este error se produce cuando se adjunta en la petición un archivo con un tamaño demasiado grande
414	Request-URI Too Long	La URL de la petición es demasiado grande, como cuando se incluyen más de 512 bytes en una petición realizada con el método GET
415	Unsupported Media Type	Al menos una parte de la petición incluye un formato que el servidor no es capaz procesar
416	Requested Range Not Suitable	El trozo de documento solicitado no está disponible, como por ejemplo cuando se solicitan bytes que están por encima del tamaño total del contenido
417	Expectation Failed	El servidor no puede procesar la petición porque al menos uno de los valores incluidos en la cabecera Expect no se pueden cumplir

Códigos de error del servidor

status	statusText	Descripción
500	Internal Server Error	Se ha producido algún error en el servidor que impide procesar la petición
501	Not Implemented	Procesar la respuesta requiere ciertas características no soportadas por el servidor
502	Bad Gateway	El servidor está actuando de proxy entre el navegador y un servidor externo del que ha obtenido una respuesta no válida
503	Service Unavailable	El servidor está sobrecargado de peticiones y no puede procesar la petición realizada
504	Gateway Timeout	El servidor está actuando de proxy entre el navegador y un servidor externo que ha tardado demasiado tiempo en responder
505	HTTP Version Not Supported	El servidor no es capaz de procesar la versión HTTP utilizada en la petición. La respuesta indica las versiones de HTTP que soporta el servidor

Estilo de arquitectura

- Las APIs de REST se diseñan en torno a recursos, que son cualquier tipo de objeto, dato o servicio al que puede acceder el cliente.
- Un recurso tiene un identificador, que es un URI que identifica de forma única ese recurso.
- Los clientes interactúan con un servicio mediante el intercambio de representaciones de recursos.
- Las APIs de REST usan una interfaz uniforme, que ayuda a desacoplar las implementaciones de clientes y servicios. En las APIs REST basadas en HTTP, la interfaz uniforme incluye el uso de verbos HTTP estándar para realizar operaciones en los recursos. Las operaciones más comunes son GET, POST, PUT, PATCH y DELETE. El código de estado de la respuesta indica el éxito o error de la petición.
- Las APIs de REST usan un modelo de solicitud sin estado. Las solicitudes HTTP deben ser independientes y pueden producirse en cualquier orden, por lo que no es factible conservar la información de estado transitoria entre solicitudes. El único lugar donde se almacena la información es en los propios recursos y cada solicitud debe ser una operación atómica.
- Las APIs de REST se controlan mediante vínculos de hipermedia.

Estilo de arquitectura

- Métodos GET
  - Petición de consulta a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia) y la cabecera accept para la respuesta.
  - Normalmente, un método GET correcto devuelve el código de estado HTTP 200 (Correcto), el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type. Si no se encuentra el recurso, el método debe devolver HTTP 404 (No encontrado).
- Métodos POST
  - Petición de crear o reemplazar a la URL sin identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Si un método POST crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), el URI del nuevo recurso se incluye en el encabezado Location de la respuesta. Con el código de estado HTTP 200, el cuerpo de respuesta contiene una representación del recurso y la cabecera content-type.
  - Si no crea un nuevo recurso, puede devolver el código de estado HTTP 200 e incluir el resultado de la operación en el cuerpo de respuesta y la cabecera content-type. O bien, si no hay ningún resultado para devolver, devolverá el código de estado HTTP 204 (Sin contenido) y sin cuerpo de respuesta.
  - Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver el código de estado HTTP 400 (Solicitud incorrecta), un 409 (Conflicto) o un 412 (fallo de concurrencia). El cuerpo de respuesta puede contener información adicional sobre el error o un vínculo a un URI que proporciona más detalles.

Estilo de arquitectura

- Métodos PUT
  - Petición de crear o reemplazar (idempotente) a la URL con identificador, la instancia en el cuerpo, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Al igual que con un método POST, si un método PUT crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), y si actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto), un 412 (fallo de concurrencia) o el recurso no existe, puede devolver un 404 (No encontrado).
  - Hay que considerar la posibilidad de implementar operaciones HTTP PUT masivas que pueden procesar por lotes las actualizaciones de varios recursos de una colección. La solicitud PUT debe especificar el URI de la colección y el cuerpo de solicitud debe especificar los detalles de los recursos que se van a modificar. Este enfoque puede ayudar a reducir el intercambio de mensajes y mejorar el rendimiento.
- Métodos DELETE
  - Petición de borrado a la URL sin identificador (todas las instancias del recurso) o con identificador (una instancia)
  - El método debe responder con un 204 (Sin contenido), que indica que la operación de eliminación es correcta, el cuerpo de respuesta no contiene información adicional. Si el recurso no existe, puede devolver un 404 (No encontrado), un 409 (Conflicto) o un 412 (fallo de concurrencia).

Estilo de arquitectura

- Métodos PATCH
  - Petición de reemplazo parcial a la URL con identificador, en el cuerpo el conjunto de cambios a aplicar, la cabecera content-type y, si espera respuesta, la cabecera accept.
  - Con una solicitud PATCH, el cliente envía un conjunto de actualizaciones a un recurso existente, en forma de un documento de revisión. El servidor procesa el documento de revisión para realizar la actualización que, por definición, no es idempotente..
  - La estructura del documento de revisión debería de seguir una sintaxis estándar como JSON Patch (RFC6902) o JSON Merge (RFC7386).
  - Los formatos MIME del documento de revisión aceptados por el servidor deberían aparecer en la cabecera Accept-Patch en respuesta a una petición OPTIONS.
  - Si el método actualiza un recurso existente, devuelve un 200 (Correcto) o un 204 (Sin contenido). Si el cliente coloca datos no válidos en la solicitud o el documento de revisión tiene un formato incorrecto, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o un 412 (fallo de concurrencia), si no se admite el formato de documento de revisión devolverá un 415 (Tipo de medio no compatible) o el recurso no existe, puede devolver un 404 (No encontrado).

Encabezado HTTP Cache-Control

- El encabezado HTTP Cache-Control especifica directivas (instrucciones) para almacenar temporalmente (caching) tanto en peticiones como en respuestas. Una directiva dada en una petición no significa que la misma directiva estar en la respuesta.
- Los valores estándar que pueden ser usados por el servidor en una respuesta HTTP son:
  - public: La respuesta puede estar almacenada en cualquier memoria cache.
  - private: La respuesta puede estar almacenada sólo por el cache de un navegador.
  - no-cache: La respuesta puede estar almacenada en cualquier memoria cache pero DEBE pasar siempre por una validación con el servidor de origen antes de utilizarse.
  - no-store: La respuesta puede no ser almacenada en cualquier cache.
  - max-age=<seconds>: La cantidad máxima de tiempo un recurso es considerado reciente.
  - s-maxage=<seconds>: Anula el encabezado max-age o el Expires, pero solo para caches compartidos (e.g., proxies).
  - must-revalidate: Indica que una vez un recurso se vuelve obsoleto, el cache no debe usar su copia obsoleta sin validar correctamente en el servidor de origen.
  - proxy-revalidate: Similar a must-revalidate, pero solo para caches compartidos (es decir, proxies). Ignorado por caches privados.
  - no-transform: No deberían hacerse transformaciones o conversiones al recurso.

Encabezados HTTP ETag, If-Match y If-None-Match

- El encabezado de respuesta de HTTP ETag es un identificador (resumen hash) para una versión específica de un recurso y los encabezados If-Match e If-None-Match de la solicitud HTTP hace que la solicitud sea condicional.
- Para los métodos GET y HEAD con If-None-Match: si el ETag no coincide con los dados, el servidor devolverá el recurso solicitado con un estado 200, si coincide el servidor debe devolver el código de estado HTTP 304 (No modificado) y DEBE generar cualquiera de los siguientes campos de encabezado que se habrían enviado en una respuesta 200 (OK) a la misma solicitud: Cache-Control, Content-Location, Date, ETag, Expires y Vary.
- Para los métodos PUT y DELETE con If-Match: si el ETag coincide con los dados, se realiza la actualización o borrado y se devuelve un estado HTTP 204 (sin contenido) incluyendo el Cache-Control y el ETag de la versión actualizada del recurso en el PUT. Si no coinciden, se ha producido un error de concurrencia, la versión del servidor ha sido modificada desde que la recibió el cliente, debe devolver una respuesta HTTP con un cuerpo de mensaje vacío y un código de estado 412 (Precondición fallida).
- Si los datos solicitados ya no existen, el servido debe devolver una respuesta HTTP con el código de estado 404 (no encontrado).

Estilo de arquitectura

- Request: Método /uri?parámetros
  - GET: Recupera el recurso (200)
    - Todos: Sin identificador
    - Uno: Con identificador
  - POST: Crea o reemplaza un nuevo recurso (201)
  - PUT: Crea o reemplaza el recurso identificado (200, 204)
  - DELETE: Elimina el recurso (204)
    - Todos: Sin identificador
    - Uno: Con identificador
- Cabeceras:
  - Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
  - Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario definir una serie de cosas:
  - Analizar el/los recurso/s a implementar
  - Diseñar la REPRESENTACION del recurso.
    - Deberemos definir el formato de trabajo del recurso: XML, JSON, HTML, imagen, RSS, etc
  - Definir el URI de acceso.
    - Deberemos indicar el/los URI de acceso para el recurso
  - Establecer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - Fijar qué códigos de estado pueden ser devueltos
    - Los códigos de estado HTTP típicos que podrían ser devueltos
- Todo lo anterior dependerá del servicio a implementar.

Definir operaciones

- Sumario y descripción de la operación.
- Dirección: URL
  - Sin identificador
  - Con identificador
  - Con parámetros de consulta
- Método: GET | POST | PUT | DELETE | PATCH
- Solicitud:
  - Cabeceras:
    - ACCEPT: formatos aceptables si espera recibir datos
    - CONTENT-TYPE: formato de envío de los datos en la solicitud
    - Otras cabeceras: Authorization, Cache-control, X-XSRF-TOKEN, …
  - Cuerpo: en caso de envío, estructura de datos formateados según el CONTENT-TYPE.
- Respuesta:
  - Cabeceras:
    - Códigos de estado HTTP: posibles y sus causas.
    - CONTENT-TYPE: formato de envío de los datos en la respuesta
    - Otras cabeceras
  - Cuerpo: en caso de respuesta, estructura de datos según código de estado y formateados según el CONTENT-TYPE.

Richardson Maturity Model

http://www.crummy.com/writing/speaking/2008-QCon/act3.html

- Nivel 0: Definir un URI y todas las operaciones son solicitudes POST a este URI.
- Nivel 1 (Pobre): Crear distintos URI para recursos individuales pero utilizan solo un método.
  - Se debe identificar un recurso
    /entities/?invoices =2 → entities/invoices/2
  - Se construyen con nombres nunca con verbos
    /getUser/{id} → /users/{id}/
    /users/{id}/edit/login → users/{id}/access-token
  - Deberían tener una estructura jerárquica
    /invoices/user/{id} → /user/{id}/invoices
- Nivel 2 (Medio): Usar métodos HTTP para definir operaciones en los recursos.
- Nivel 3 (Óptimo): Usar hipermedia (HATEOAS, se describe a continuación).

Hypermedia

- Uno de los principales propósitos que se esconden detrás de REST es que debe ser posible navegar por todo el conjunto de recursos sin necesidad de conocer el esquema de URI. Cada solicitud HTTP GET debe devolver la información necesaria para encontrar los recursos relacionados directamente con el objeto solicitado mediante los hipervínculos que se incluyen en la respuesta, y también se le debe proporcionar información que describa las operaciones disponibles en cada uno de estos recursos.
- Este principio se conoce como HATEOAS, del inglés Hypertext as the Engine of Application State (Hipertexto como motor del estado de la aplicación). El sistema es realmente una máquina de estado finito, y la respuesta a cada solicitud contiene la información necesaria para pasar de un estado a otro; ninguna otra información debería ser necesaria.
- Se basa en la idea de enlazar recursos: propiedades que son enlaces a otros recursos.
- Para que sea útil, el cliente debe saber que en la respuesta hay contenido hypermedia.
- En content-type es clave para esto
  - Un tipo genérico no aporta nada:
    Content-Type: text/xml
  - Se pueden crear tipos propios
    Content-Type:application/servicio+xml

JSON Hypertext Application Language

- RFC4627 http://tools.ietf.org/html/draft-kelly-json-hal-00
- HATEOAS: Content-Type: application/hal+json

```
{
    "_links": {
      "self": {"href": "/orders/523" },
      "warehouse": {"href": "/warehouse/56" },
      "invoice": {"href": "/invoices/873"}
    },
    , "currency": "USD"
    , "status": "shipped"
    , "total": 10.20
}
```


Características de una API bien diseñada

- **Fácil de leer y trabajar**: con una API bien diseñada será fácil trabajar, y sus recursos y operaciones asociadas pueden ser memorizados rápidamente por los desarrolladores que trabajan con ella constantemente.
- **Difícil de usar mal**: la implementación e integración con una API con un buen diseño será un proceso sencillo, y escribir código incorrecto será un resultado menos probable porque tiene comentarios informativos y no aplica pautas estrictas al consumidor final de la API.
- **Completa y concisa**: Finalmente, una API completa hará posible que los desarrolladores creen aplicaciones completas con los datos que expone. Por lo general, la completitud ocurre con el tiempo, y la mayoría de los diseñadores y desarrolladores de API construyen gradualmente sobre las APIs existentes. Es un ideal por el que todo ingeniero o empresa con una API debe esforzarse.

Guía de diseño

- Organización de la API en torno a los recursos
- Definición de operaciones en términos de métodos HTTP
- Conformidad con la semántica HTTP
- Filtrado y paginación de los datos
- Compatibilidad con respuestas parciales en recursos binarios de gran tamaño
- Uso de HATEOAS para permitir la navegación a los recursos relacionados
- Control de versiones en la API RESTful
- Documentación Open API

Definición de recursos

- La organización de la API en torno a los recursos se centran en las entidades de dominio que debe exponer la API. Por ejemplo, en un sistema de comercio electrónico, las entidades principales podrían ser clientes y pedidos. La creación de un pedido se puede lograr mediante el envío de una solicitud HTTP POST que contiene la información del pedido. La respuesta HTTP indica si el pedido se realizó correctamente o no.
- Un recurso no tiene que estar basado en un solo elemento de datos físico o tablas de una base de datos relacional. La finalidad de REST es modelar entidades y las operaciones que un consumidor externo puede realizar sobre esas entidades, no debe exponerse a la implementación interna.
- Es necesario adoptar una convención de nomenclatura coherente para los URI. Los URI de recursos deben basarse en nombres (de recurso), nunca en verbos (las operaciones en el recurso) y, en general, resulta útil usar nombres plurales que hagan referencia a colecciones. Debe seguir una estructura jerárquica que refleje las relaciones entre los diferentes tipos de recursos.
- Hay que considerar el uso del enfoque HATEOAS para permitir el descubrimiento y la navegación a los recursos relacionados o el enfoque del patrón agregado.

Definición de recursos

- Exponer una colección de recursos con un único URI puede dar lugar a que las aplicaciones capturen grandes cantidades de datos cuando solo se requiere un subconjunto de la información. A través de la definición de parámetros de cadena de consulta se pueden realizar particiones horizontales con filtrado, ordenación y paginación, o particiones verticales con la proyección de las propiedades a recuperar:
  - https://host/users?page=1&rows=20&projection=userId,name,lastAccess
- La definición de operaciones con el recurso se realiza en términos de métodos HTTP, estableciendo cuales serán soportadas. Las operaciones no soportadas por métodos HTTP deben sustantivarse al crearles una URI especifica y utilizar el método HTTP semánticamente mas próximo.
  - DELETE https://host/users/bloqueo (desbloquear)
  - POST https://host/pedido/171/factura (facturar)
- Hay que establecer el tipo o tipos de formatos mas adecuados para las representaciones de recursos. Los formatos se especifican mediante el uso de tipos de medios, también denominados tipos MIME. En el caso de datos no binarios, la mayoría de las APIs web admiten JSON (application/json) y, posiblemente, XML (application/xml).

Políticas de versionado

- Es muy poco probable que una API permanezca estática. Conforme los requisitos empresariales cambian, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos de los recursos debe adecuarse.
- Los cambios rupturistas no son compatibles con la versión anterior, el consumidor tendrá que adaptar su código para pasar su aplicación existente a la nueva versión y evitar que se rompa.
- Hay dos razones principales por las que las APIs HTTP se comportan de manera diferente al resto de las APIs:
  - El código del cliente dicta lo que lo romperá: Un proveedor de API no tiene control sobre las herramientas que un consumidor puede usar para interpretar una respuesta de la API y la tolerancia al cambio que tienen esas herramientas varían ampliamente, si es rupturista o no.
  - El proveedor de API elige si los cambios son opcionales o transparentes: Los proveedores de API pueden actualizar su API y los cambios en las respuestas afectarán inmediatamente a los clientes. Los clientes no pueden decidir si adoptar o no la nueva versión, lo que puede generar fallos en cascada en los cambios rupturistas.

Políticas de versionado

- El control de versiones permite que una API indique la versión expuesta y que una aplicación cliente pueda enviar solicitudes que se dirijan a una versión específica con una característica o un recurso.
  - Sin control de versiones: Este es el enfoque más sencillo y puede ser aceptable para algunas APIs internas. Los grandes cambios podrían representarse como nuevos recursos o nuevos vínculos.
  - Control de versiones en URI: Cada vez que modifica la API web o cambia el esquema de recursos, agrega un número de versión al URI para cada recurso. Los URI ya existentes deben seguir funcionando como antes y devolver los recursos conforme a su esquema original.

    http://host/v2/users
  - Control de versiones en cadena de consulta: En lugar de proporcionar varios URI, se puede especificar la versión del recurso mediante un parámetro dentro de la cadena de consulta anexada a la solicitud HTTP:

    http://host/users?versión=2.0

Políticas de versionado

  - Control de versiones en encabezado: En lugar de anexar el número de versión como un parámetro de cadena de consulta, se podría implementar un encabezado personalizado que indica la versión del recurso. Este enfoque requiere que la aplicación cliente agregue el encabezado adecuado a las solicitudes, aunque el código que controla la solicitud de cliente puede usar un valor predeterminado (versión actual) si se omite el encabezado de versión.

    GET https://host/users HTTP/1.1

    Custom-Header: api-version=1
  - Control de versiones por MIME (tipo de medio): Cuando una aplicación cliente envía una solicitud HTTP GET a un servidor web, debe prever el formato del contenido que puede controlar mediante el uso de un encabezado Accept.

    GET https:// host/users/3 HTTP/1.1

    Accept: application/vnd.mi-api.v1+json
- Si la versión no está soportada, el servicio podría generar un mensaje de respuesta HTTP 406 (no aceptable) o devolver un mensaje con un tipo de medio predeterminado.
- Los esquemas de control de versiones de URI y de cadena de consulta son compatibles con la caché HTTP puesto que la misma combinación de URI y cadena de consulta hace referencia siempre a los mismos datos.

Políticas de versionado

- Dentro de la política de versionado es conveniente planificar la obsolescencia y la política de desaprobación.
- La obsolescencia programada establece el periodo máximo, como una franja temporal o un número de versiones, en que se va a dar soporte a cada versión, evitando los sobrecostes derivados de mantener versiones obsoletas indefinidamente.
- Dentro de la política de desaprobación, para ayudar a garantizar que los consumidores tengan tiempo suficiente y una ruta clara de actualización, se debe establecer el número de versiones en que se mantendrá una característica marcada como obsoleta antes de su desaparición definitiva.
- La obsolescencia programada y la política de desaprobación beneficia a los consumidores de la API porque proporcionan estabilidad y sabrán qué esperar a medida que las APIs evolucionen.
- Para mejorar la calidad y avanzar las novedades, se podrán realizar lanzamientos de versiones Beta y Release Candidatos (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

Guía de implementación

- Procesamiento de solicitudes
  - Las acciones GET, PUT, DELETE, HEAD y PATCH deben ser idempotentes.
  - Las acciones POST que crean nuevos recursos no deben tener efectos secundarios no relacionados.
  - Evitar implementar operaciones POST, PUT y DELETE que generen mucha conversación.
  - Seguir la especificación HTTP al enviar una respuesta.
  - Admitir la negociación de contenido.
  - Proporcionar vínculos que permitan la navegación y la detección de recursos de estilo HATEOAS.

Guía de implementación

- Administración de respuestas y solicitudes de gran tamaño
  - Optimizar las solicitudes y respuestas que impliquen objetos grandes.
  - Admitir la paginación de las solicitudes que pueden devolver grandes cantidades de objetos.
  - Implementar respuestas parciales para los clientes que no admitan operaciones asincrónicas.
  - Evitar enviar mensajes de estado 100-Continuar innecesarios en las aplicaciones cliente.
- Mantenimiento de la capacidad de respuesta, la escalabilidad y la disponibilidad
  - Ofrecer compatibilidad asincrónica para las solicitudes de ejecución prolongada.
  - Comprobar que ninguna de las solicitudes tenga estado.
  - Realizar un seguimiento de los clientes e implementar limitaciones para reducir las posibilidades de ataques de denegación de servicio.
  - Administrar con cuidado las conexiones HTTP persistentes.

Guía de implementación

- Control de excepciones
  - Capturar todas las excepciones y devolver una respuesta significativa a los clientes.
  - Distinguir entre los errores del lado cliente y del lado servidor.
  - Evitar las vulnerabilidades por exceso de información.
  - Controlar las excepciones de una forma coherente y registrar la información sobre los errores.
- Optimización del acceso a los datos en el lado cliente
  - Admitir el almacenamiento en caché del lado cliente.
  - Proporcionar ETags para optimizar el procesamiento de las consultas.
  - Usar ETags para admitir la simultaneidad optimista.

Guía de implementación

- Publicación y administración de una API web
  - Todas las solicitudes deben autenticarse y autorizarse, y debe aplicarse el nivel de control de acceso adecuado.
  - Una API web comercial puede estar sujeta a diversas garantías de calidad relativas a los tiempos de respuesta. Es importante asegurarse de que ese entorno de host es escalable si la carga puede variar considerablemente con el tiempo.
  - Puede ser necesario realizar mediciones de las solicitudes para fines de monetización.
  - Es posible que sea necesario regular el flujo de tráfico a la API web e implementar la limitación para clientes concretos que hayan agotado sus cuotas.
  - Los requisitos normativos podrían requerir un registro y una auditoría de todas las solicitudes y respuestas.
  - Para garantizar la disponibilidad, puede ser necesario supervisar el estado del servidor que hospeda la API web y reiniciarlo si hiciera falta.

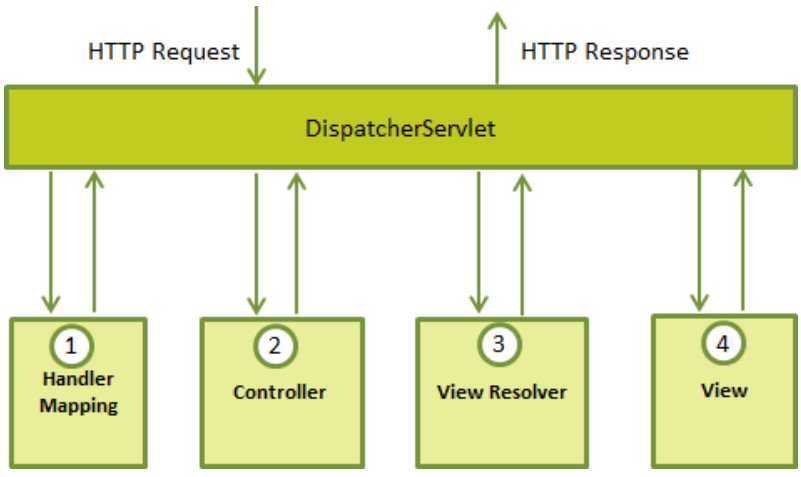
Guía de implementación

- Pruebas de la API
  - Ejercitar todas las rutas y parámetros para comprobar que invocan las operaciones correctas.
  - Verificar que cada operación devuelve los códigos de estado HTTP correctos para diferentes combinaciones de entradas.
  - Comprobar que todas las rutas estén protegidas correctamente y que estén sujetas a las comprobaciones de autenticación y autorización apropiadas.
  - Verificar el control de excepciones que realiza cada operación y que se devuelve una respuesta HTTP adecuada y significativa de vuelta a la aplicación cliente.
  - Comprobar que los mensajes de solicitud y respuesta están formados correctamente.
  - Comprobar que todos los vínculos dentro de los mensajes de respuesta no están rotos.

Spring Web MVC

- Spring Web MVC es el marco web original creado para dar soporte a las aplicaciones web de Servlet-stack basadas en la API de Servlet y desplegadas en los contenedores de Servlet. Está incluido Spring Framework desde el principio.
- El Modelo Vista Controlador (MVC) es un patrón de arquitectura de software (presentación) que separa los datos y la lógica de negocio de una aplicación del interfaz de usuario y del módulo encargado de gestionar los eventos y las comunicaciones.
- Este patrón de diseño se basa en las ideas de reutilización de código y la separación de conceptos, características que buscan facilitar la tarea de desarrollo de aplicaciones, prueba y su posterior mantenimiento.
- Para todo tipo de sistemas (Escritorio, Web, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

Procesamiento de una solicitud



Recursos

- Son clases Java con la anotación @RestController (@Controller + @ResponseBody) y representan a los servicios REST, son controller que reciben y responden a las peticiones de los clientes.

```
@RestController
public class PaisController {
        @Autowired
        private PaisRepository paisRepository;
```

- Los métodos de la clase (métodos de acción u operaciones) que interactúan con el cliente deben llevar la anotación @RequestMapping, con la subruta y el RequestMethod.

```
@RequestMapping(value = "/paises/{id}", method = RequestMethod.GET)
public ResponseEntity<Pais> getToDoById(@PathVariable("id") String id) {
      return new ResponseEntity<Pais>(paisRepository.findById(id).get(), HttpStatus.OK);
}
```


RequestMapping

- La anotación @RequestMapping permite asignar solicitudes a los métodos de acción de los controladores.
- Tiene varios atributos para definir URL, método HTTP, parámetros de solicitud, encabezados y tipos de medios.
- Se puede usar a el nivel de clase para expresar asignaciones compartidas o a el nivel de método para limitar a una asignación de endpoint específica.
- También hay atajos con el método HTTP predefinido:
  - @GetMapping
  - @PostMapping
  - @PutMapping
  - @DeleteMapping
  - @PatchMapping

Patrones de URI

- Establece que URLs serán derivadas al controlador.
- Puede asignar solicitudes utilizando los siguientes patrones globales y comodines:

?	Coincide con un carácter
*	Coincide con cero o más caracteres dentro de un segmento de ruta
**	Coincide con cero o más segmentos de ruta hasta el final de la ruta. Solo puede ir al final del patrón.
{param}	Coincide con un segmento de ruta (*) y lo captura como un parámetro
	Con {param:\d+} debe coincidir con la expresión regular
	Con {*param} captura hasta el final de la ruta. Solo puede ir al final del patrón.

- También puede declarar variables en la URI y acceder a sus valores con anotando con @PathVariable los parámetros, debe respetarse la correspondencia de nombres:

```
@GetMapping("/owners/{ownerId}/pets/{petId}")
public Pet findPet(@PathVariable Long ownerId, @PathVariable Long petId) {
  // ...
}
```


Restricciones

- cosumes: formatos MIME permitidos del encabezado Content-type

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
```

- produces : formatos MIME permitidos del encabezado Accept

```
@GetMapping(path = "/pets/{petId}", produces = "application/json;charset=UTF-8")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
```

- params: valores permitidos de los QueryParams
- headers: valores permitidos de los encabezados

```
@GetMapping(path = "/pets/{petId}", params = "myParam=myValue", headers =
    "myHeader=myValue")
public void findPet(@PathVariable String petId) {
```

Inyección de Parámetros

- El API decodifica la petición e inyecta los datos como parámetros en el método.
- Es necesario anotar los parámetros para indicar la fuente del dato a inyectar.
- En las anotaciones será necesario indicar el nombre del origen en caso de no existir correspondencia de nombres con el de los parámetros.
- El tipo de origen, en la mayoría de los casos, es String que puede discrepar con los tipos de los parámetros, en tales casos, la conversión de tipo se aplica automáticamente en función de los convertidores configurados.
- Por defecto los parámetros son obligatorios, se puede indicar que sean opcionales, se inicializaran a null si no reciben en la petición salvo que se indique el valor por defecto:

```
@RequestParam(required=false, defaultValue="1")
```


Inyección de Parámetros

Anotación	Descripcción
@PathVariable	Para acceder a las variables de la plantilla URI.
@MatrixVariable	Para acceder a pares nombre-valor en segmentos de ruta URI.
@RequestParam	Para acceder a los parámetros de solicitud del Servlet (QueryString o Form), incluidos los archivos de varias partes. Los valores de los parámetros se convierten al tipo de argumento del método declarado.
@RequestHeader	Para acceder a las cabeceras de solicitud. Los valores de encabezado se convierten al tipo de argumento del método declarado.
@CookieValue	Para el acceso a las cookies. Los valores de las cookies se convierten al tipo de argumento del método declarado.

Inyección de Parámetros

Anotación	Descripcción
@RequestBody	Para acceder al cuerpo de la solicitud HTTP. El contenido del cuerpo se convierte al tipo de argumento del método declarado utilizando implementaciones HttpMessageConverter.
@RequestPart	Para acceder a una parte en una solicitud multipart/form-data, convertir el cuerpo de la parte con un HttpMessageConverter.
~~@ModelAttribute~~	~~Para acceder a un atributo existente en el modelo (instanciado si no está presente) con enlace de datos y validación aplicada.~~
@SessionAttribute	Para acceder a cualquier atributo de sesión, a diferencia de los atributos de modelo almacenados en la sesión como resultado de una declaración @SessionAttributes de nivel de clase .
@RequestAttribute	Para acceder a los atributos de solicitud.

Inyecciones adicionales

- Se definen los parámetros de los tipos adecuados Spring inyectara los objetos indicados.
- Para acceder a los valores originales de los parámetros anotados:
  - WebRequest, NativeWebRequest, MultipartRequest, ServletRequest, ServletResponse, MultipartHttpServletRequest, HttpSession, SessionStatus, HttpMethod,
- Para acceder a la configuración regional de la solicitud actual y la zona horaria asociada
  - java.util.Locale, java.util.TimeZone
- Para acceder al usuario autenticado actual
  - java.security.Principal
- Para acceder al cuerpo de la solicitud o de la respuesta sin procesar
  - java.io.InputStream, java.io.Reader, java.io.OutputStream, java.io.Writer
- Para acceder al modelo
  - java.util.Map, org.springframework.ui.Model, org.springframework.ui.ModelMap
- Para acceder a los errores de validación y enlace de datos
  - Errors, BindingResult
- Para preparar una URL relacionada con la solicitud actual
  - UriComponentsBuilder

Paginación y Ordenación

QueryString	Descripción
page	Número de página en base 0. Por defecto: página 0.
size	Tamaño de página. Por defecto: 20.
sort	Propiedades de ordenación en el formato property,property(,ASC|DESC). Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (?sort=firstname&sort=lastname,asc)

```
@GetMapping
public List<Employee> getAll(@PageableDefault(size = 10) Pageable pageable) {
    if(pageable.isPaged()) {
        return dao.findAll(pageable).getContent();
    } else
        return dao.findAll();
}
```

Los nombres de los parámetros se pueden configurar:

```
spring.data.web.pageable.page-parameter=_page
spring.data.web.pageable.size-parameter=_rows
spring.data.rest.sort-param-name=_sort
```


@RequestBody

- Se puede utilizar la anotación @RequestBody para que el cuerpo de la solicitud se lea y se deserialice a parámetro a través de HttpMessageConverter. Se pueden usar convertidores de mensajes para configurar o personalizar la conversión de mensajes.

```
@PostMapping("/accounts")
public void handle(@RequestBody Account account) {
```

- En combinación con la anotación javax.validation.Valid o la de Spring @Validated, se aplica la validación estándar de Bean. De forma predeterminada, los errores de validación causan una MethodArgumentNotValidException, que se convierte en una respuesta 400 (BAD_REQUEST). Alternativamente, se pueden manejar los errores de validación localmente dentro del controlador a través de un argumento Errors o BindingResult:

```
@PostMapping("/region")
public void handle(@Valid @RequestBody Region item, BindingResult result) {
```

Inyección de Parámetros

```
// http://localhost:8080/params/1?nom=kk

@GetMapping("/params/{id}")
public String cotilla(
      @PathVariable String id,
      @RequestParam String nom,
      @RequestHeader("Accept-Language") String language,
      @CookieValue("JSESSIONID") String cookie) {
   StringBuilder sb = new StringBuilder();
   sb.append("id: " + id + "\n");
   sb.append("nom: " + nom + "\n");
   sb.append("language: " + language + "\n");
   sb.append("cookie: " + cookie + "\n");
   return sb.toString();
}
```


Respuesta

- La anotación @ResponseBody (incluida en el @RestController) en un método indica que el retorno será serializado en el cuerpo de la respuesta a través de un HttpMessageConverter.

```
@PostMapping("/invierte")
@ResponseBody
public Punto body(@RequestBody Punto p) {
      int x = p.getX();
      p.setX(p.getY());
      p.setY(x);
      return p;
}
```

- El código de estado de la respuesta se puede establecer con la anotación @ResponseStatus:

```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public void add(@RequestBody Punto p) { … }
```

Respuesta personalizada

- La clase ResponseEntity permite agregar estado y encabezados a la respuesta (no requiere la anotación @ResponseBody).

```
@GetMapping(value="/pais")
public ResponseEntity<List<Pais>> getAll(){
    return new ResponseEntity<List<Pais>>(
            paisRepository.findAll(),
            HttpStatus.OK);
}
```

- La clase ResponseEntity dispone de builder para generar la respuesta:

```
return ResponseEntity.ok().eTag(etag).build(body);
```


Mapeo de respuestas genéricas a excepciones.

- Spring Framework no incluye automáticamente los detalles de error en el cuerpo de la respuesta porque la representación es específica de la aplicación.
- Los métodos de acción de los controladores pueden capturar las excepciones (try catch) y devuelven un ResponseEntity que permite establecer el estado y el cuerpo de la respuesta, tanto de las correctas como las erróneas.
- Una clase @RestController puede contar con métodos anotados con @ExceptionHandler que intercepten determinadas excepciones producidas en el resto de los métodos de la clase y pueden devolver un ResponseEntity, un ProblemDetail o estar anotadas con un @ResponseStatus y generar el cuerpo con los detalles de error.
- Esto mismo se puede hacer globalmente en clases anotadas con @ControllerAdvice que solo tienen los correspondientes métodos @ExceptionHandler.
- @RestControllerAdvice es una anotación compuesta que se anota con @ControllerAdvice y @ResponseBody, lo que esencialmente significa que los métodos @ExceptionHandler se representan en el cuerpo de la respuesta a través de la conversión del mensaje (en comparación con la resolución de la vista o la representación de la plantilla).

Excepciones personalizadas

```
public class NotFoundException extends Exception {
    private static final long serialVersionUID = 1L;
    public NotFoundException() {
        super("NOT FOUND");
    }
    public NotFoundException(String message) {
        super(message);
    }
    public NotFoundException(Throwable cause) {
        super("NOT FOUND", cause);
    }
    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public NotFoundException(String message, Throwable cause, boolean enableSuppression, boolean writableStackTrace) {
        super(message, cause, enableSuppression, writableStackTrace);
    }
}
```

Nota: *Pueden extender a org.springframework.web.ErrorResponseException si pertenecen a las capas de presentación.*

Error Personalizado

```
public class ErrorMessage implements Serializable {
    private static final long serialVersionUID = 1L;
    private String error, message;
    public ErrorMessage(String error, String message) {
        this.error = error;
        this.message = message;
    }
    public String getError() { return error; }
    public void setError(String error) { this.error = error; }
    public String getMessage() { return message; }
    public void setMessage(String message) { this.message = message; }
}
```

Respuestas de error

- Un requisito común para los servicios REST es incluir detalles en el cuerpo de las respuestas de error. Spring Framework admite la especificación RFC 7807 para "Detalles del problema para las API HTTP". Las principales abstracciones son:
  - ProblemDetail: representación RFC 7807 del detalle de un problema; un contenedor simple para los campos estándar y no estándar definidos en la especificación.
  - ErrorResponse: interface para exponer los detalles de la respuesta de error de HTTP, incluido el estado de HTTP, los encabezados de respuesta y un cuerpo en el formato de RFC 7807; esto permite que las excepciones encapsulen y expongan los detalles de cómo se asignan a una respuesta HTTP. Todas las excepciones de Spring MVC lo implementan.
  - ErrorResponseException: implementación base de ErrorResponse que otras clases de excepción pueden usar como clase base.
  - ResponseStatusException: subclase de ErrorResponseException con el estado y una razón que, de forma predeterminada, asigna al status y detail de un ProblemDetail.
  - ResponseEntityExceptionHandler: clase base conveniente para un @ControllerAdvice que maneja todas las excepciones de Spring MVC, y cualquiera ErrorResponseException, y genera una respuesta de error con un cuerpo.

Problem Details (RFC 7807)

- El objeto de detalles del problema puede tener los siguientes miembros:
  - "type" (cadena): URI que identifica el tipo de problema y proporciona documentación legible por humanos para el tipo de problema. El valor "about:blank" (predeterminado) indica que el problema no tiene semántica adicional a la del código de estado HTTP.
  - "title" (cadena): Breve resumen legible por humanos del problema escribe. NO DEBE cambiar de una ocurrencia a otra del mismo problema, excepto para fines de localización. Con "type": "about:blank", DEBE coincidir con la versión textual del status.
  - "status" (número): Código de estado HTTP (por conveniencia, opcional, debe coincidir).
  - "detail" (cadena): Explicación legible por humanos específica de la ocurrencia concreta del problema.
  - "instance" (cadena): URI de referencia que identifica el origen de la ocurrencia del problema.
- Las definiciones de tipo de problema PUEDEN extender el objeto con miembros adicionales.

@RestControllerAdvice

```
@RestControllerAdvice
public class ApiExceptionHandler {
    @ExceptionHandler({ NotFoundException.class })
    @ResponseStatus(HttpStatus.NOT_FOUND)
    public ErrorMessage notFoundRequest(Exception exception) {
        return new ErrorMessage(exception.getMessage(), ServletUriComponentsBuilder.fromCurrentRequest().build().toUriString());
    }
    @ExceptionHandler({ NotFoundException.class })
    public ProblemDetail notFoundRequest(Exception exception) {
        return ProblemDetail.forStatus(HttpStatus.NOT_FOUND);
    }
    @ExceptionHandler({ ErrorResponseException.class })
    public ProblemDetail defaultResponse(ErrorResponse  exception) {
        return exception.getBody();
    }
    @ExceptionHandler({ BadRequestException.class, DuplicateKeyException.class })
    public ProblemDetail badRequest(Exception exception) {
        return ProblemDetail.forStatusAndDetail(HttpStatus.BAD_REQUEST, exception.getMessage());
    }
}
```


Controlador de error global

- La aplicación Spring Boot tiene una configuración predeterminada para el manejo de errores. Se puede configurar con:
  server.error.include-stacktrace=never
  server.error.include-binding-errors=always
- Si la aplicación tiene un controlador que lo implementa ErrorController, reemplaza a BasicErrorController.

```
@RestController
public class CustomErrorController implements ErrorController {
   @RequestMapping(path = "/error")
   public Map<String, Object> handle(HttpServletRequest request) {
      Map<String, Object> map = new HashMap<String, Object>();
      map.put("status", request.getAttribute("javax.servlet.error.status_code"));
      map.put("error", request.getAttribute("javax.servlet.error.message"));
      return map;
   }
```

Servicio Web RESTful

```
import jakarta.validation.ConstraintViolation;
import jakarta.validation.Valid;
import jakarta.validation.Validator;
import jakarta.websocket.server.PathParam;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.PutMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;
import org.springframework.http.HttpStatus;

import curso.api.exceptions.BadRequestException;
import curso.api.exceptions.NotFoundException;
import curso.model.Actor;
import curso.repositories.ActorRepository;
```


Servicio Web RESTful

```
@RestController
@RequestMapping("/api/actores")
public class ActorResource {
	@Autowired
	private ActorRepository dao;

	@Autowired
	private Validator validator;

	@GetMapping
	public List<Actor> getAll() {
		// …
	}

	@GetMapping(path = "/{id}")
	public Actor getOne(@PathVariable int id) throws NotFoundException {
		// …
	}
```

Servicio Web RESTful

```
    @PostMapping
    public ResponseEntity<Object> create(@Valid @RequestBody Actor item) throws BadRequestException {
        // ...
        URI location = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
            .buildAndExpand(newItem.getActorId()).toUri();
        return ResponseEntity.created(location).build();

    }

    @PutMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void update(@PathVariable int id, @Valid @RequestBody Actor item) throws BadRequestException, NotFoundException {
        // ...
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void delete(@PathVariable int id) {
        // ..
    }
}
```


HATEOAS

- HATEOAS es la abreviación de Hypermedia as the Engine of Application State (hipermedia como motor del estado de la aplicación).
- Es una restricción de la arquitectura de la aplicación REST que lo distingue de la mayoría de otras arquitecturas.
- El principio es que un cliente interactúa con una aplicación de red completamente a través de hipermedia proporcionadas dinámicamente por los servidores de aplicaciones.
- El cliente REST debe ir navegando por la información y no necesita ningún conocimiento previo acerca de la forma de interactuar con cualquier aplicación o servidor más allá de una comprensión genérica de hipermedia.
- En otras palabras cuando el servidor nos devuelva la representación de un recurso parte de la información devuelta serán identificadores únicos en forma de hipervínculos a otros recursos asociados.
- Spring HATEOAS proporciona algunas API para facilitar la creación de representaciones REST que siguen el principio de HATEOAS cuando se trabaja con Spring y especialmente con Spring MVC. El problema central que trata de resolver es la creación de enlaces y el ensamblaje de representación

Spring HATEOAS

- La clase base ResourceSupport con soporte para la colección _links.
  class PersonaDTO extends ResourceSupport {
- El objeto de valor Link sigue la definición de enlace Atom que consta de los atributos rel y href. Contiene algunas constantes para relaciones conocidas como self, next, etc.
- Spring Hateoas ahora proporciona una ControllerLinkBuilder que permite crear enlaces apuntando a clases de controladores:
  import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
- Para añadir una referencia a si mismo:
  personaDTO.add(linkTo(PersonaResource.class).withSelfRel());
  personaDTO.add(linkTo(PersonaResource.class).slash(personaDTO.getId()).withSelfRel());
- Para añadir una referencia a si mismo como método:
  personaDTO.add(linkTo(PersonaResource.class.getMethod("get", Long.class),
  personaDTO.getId()).withSelfRel());
- Para crear una referencia a un elemento interno:
  personaDTO.add(linkTo(PersonaResource.class).
  slash(personaDTO.getId()).slash("direciones").withRel("direciones"));

Spring HATEOAS

- La interfaz EntityLinks permite generar la referencia a partir de la entidad del modelo.
- Para configurarlo:
  @Configuration
  @EnableEntityLinks
  public class MyConfig {
- Hay que asociar las entidades a los RestController:
  @RestController
  @RequestMapping(value = "/api/personas")
  @ExposesResourceFor(Persona.class)
  public class PersonaResource {
- Se inyecta:
  @Autowired EntityLinks entityLinks;
- Para añadir una referencia:
  personaDTO.add(entityLinks.linkToSingleResource(PersonaResource.class, personaDTO.getId()).withSelfRel());

Spring Data Rest

- Spring Data REST se basa en los repositorios de Spring Data y los exporta automáticamente como recursos REST. Aprovecha la hipermedia para que los clientes encuentren automáticamente la funcionalidad expuesta por los repositorios e integren estos recursos en la funcionalidad relacionada basada en hipermedia. Spring Data REST es en sí misma una aplicación Spring MVC y está diseñada de tal manera que puede integrarse con las aplicaciones Spring MVC existentes con un mínimo esfuerzo.
- De forma predeterminada, Spring Data REST ofrece los recursos REST en la URI raíz, '/', se puede cambiar la URI base configurando en el fichero application.properties:
  - spring.data.rest.basePath=/api
- Dado que la funcionalidad principal de Spring Data REST es exportar como recursos los repositorios de Spring Data, el artefacto principal será la interfaz del repositorio.

Spring Data Rest

- Spring Data REST es expone los metodos del repositorio como métodos REST:
  - GET: findAll(), findAll(Pageable), findAll(Sort)
    - Si el repositorio tiene capacidades de paginación, el recurso toma los siguientes parámetros:
      - page: El número de página a acceder (base 0, por defecto a 0).
      - size: El tamaño de página solicitado (por defecto a 20).
      - sort: Una colección de directivas de género en el formato ($propertyname,)+[asc|desc]?.
  - POST, PUT, PATCH: save(item)
  - DELETE: deleteById(id)
- Devuelve el conjunto de códigos de estado predeterminados:
  - 200 OK: Para peticiones GET .
  - 201 Created: Para solicitudes POST y PUT que crean nuevos recursos.
  - 204 No Content: Para solicitudes PUT, PATCH y DELETE cuando está configurada para no devolver cuerpos de respuesta para actualizaciones de recursos (RepositoryRestConfiguration.returnBodyOnUpdate). Si se configura incluir respuestas para PUT, se devuelve 200 OK para las actualizaciones y 201 Created si crea nuevos recursos.

Spring Data Rest

- Para cambiar la configuración predeterminada del REST:

```
@RepositoryRestResource(path="personas", itemResourceRel="persona", collectionResourceRel="personas")
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(path = "por-nombre")
    List<Person> findByNombre(String nombre);
    // http://localhost:8080/personas/search/nombre?nombre=terry
```

- Para ocultar ciertos repositorios, métodos de consulta o campos

```
@RepositoryRestResource(exported = false)
interface PersonaRepository extends JpaRepository<Persona, Integer> {
    @RestResource(exported = false)
    List<Person> findByName(String name);
    @Override
    @RestResource(exported = false)
    void deleteById(Long id);
```


Spring Data Rest

- Spring Data REST presenta una vista predeterminada del modelo de dominio que exporta. Sin embargo, a veces, es posible que deba modificar la vista de ese modelo por varias razones.

  Mediante un interfaz **en el paquete de las entidades o en uno de subpaquetes** se crea un proyección con nombre:

```
@Projection(name = "personasAcortado", types = { Personas.class })
public interface PersonaProjection {
    public int getPersonaId();
    public String getNombre();
    public String getApellidos();
}
```

- Para acceder a la proyección:
  - http://localhost:8080/personas?projection=personasAcortado
- Para fijar la proyección por defecto:

```
@RepositoryRestResource(excerptProjection = PersonaProjection.class)
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

Spring Data Rest

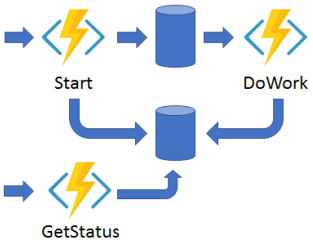
- Spring Data REST usa HAL para representar las respuestas, que define los enlaces que se incluirán en cada propiedad del documento devuelto.
- Spring Data REST proporciona un documento ALPS (Semántica de perfil de nivel de aplicación) para cada repositorio exportado que se puede usar como un perfil para explicar la semántica de la aplicación en un documento con un tipo de medio agnóstico de la aplicación (como HTML, HAL, Collection + JSON, Siren, etc.).
  - http://localhost:8080/profile
  - http://localhost:8080/profile/personas

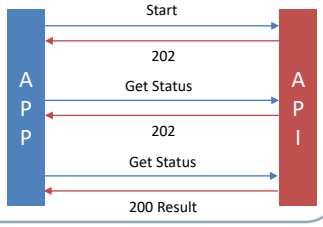
WebHooks

- Los webhooks son eventos HTTP que desencadenan acciones. Su nombre se debe a que funcionan como «enganches» de los programas en Internet y casi siempre se utilizan para la comunicación entre sistemas. Son la manera más sencilla de obtener un aviso cuando algo ocurre en otro sistema y para el intercambio de datos entre aplicaciones web, permitiendo las llamada asíncronas en HTTP.
- Un webhook es una retro llamada HTTP, una solicitud HTTP GET/POST insertada en una página web, que interviene cuando ocurre algo (una notificación de evento a través de HTTP GET/POST).
- Los webhooks se utilizan para las notificaciones en tiempo real (con los datos del evento como parámetros o en cuerpo en JSON o XML) a una determinada dirección http:// o https://, que puede:
  - almacenar los datos del evento en JSON o XML
  - generar una respuesta que permita actualizarse al sistema donde se produce el evento
  - ejecutar un proceso en el sistema receptor del evento (Ej: enviar un correo electrónico)
- Los webhook están pensados para su utilización desde páginas web y sus diferentes consumidores: navegadores, correo electrónico, webapps, …
- Un ejemplo típico es su utilización en correos electrónico de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario a abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo GET/POST).

WebHooks

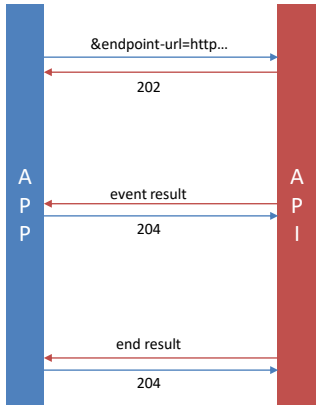
- El patrón Async HTTP APIs soluciona el problema de coordinar el estado de las operaciones de larga duración con los clientes externos. Una forma habitual de implementar este patrón es que un punto de conexión HTTP desencadene la acción de larga duración. A continuación, el cliente se redirige a un punto de conexión de estado al que sondea para saber el estado actual del proceso y cuando finaliza la operación.
- Un endpoint HTTP desencadena el proceso con la acción de larga duración y devuelve inmediatamente un 202 si la petición es correcta y, opcionalmente, como carga útil las URLs de los endpoints de estado y control. Las peticiones incorrectas recibirán el 4XX apropiado.
- El cliente sondea periódicamente el endpoint de estado y obtiene:
  - 202 con el estado actual del proceso como carga útil mientras el proceso este en marcha.
  - 200 con el estado final (correcto o fallido) del proceso como carga útil cuando haya finalizado la operación.
- El endpoint HTTP desencadenador puede suministrar endpoints adicionales para pausar, reanudar, cancelar/terminar, reiniciar o enviar eventos al proceso en marcha.





WebHooks

- El patrón Reverse API soluciona el problema del sondeo periódico, las API inversas invierten esta situación, para que sea la API invocada la que notifique automáticamente cuando se ha producido un determinado evento. El cliente debe crear su propia API para recibir las notificaciones.
- Al realizar la petición al endpoint HTTP desencadenador, se suministra un endpoint propio para que el proceso desencadenado pueda notificar cuándo ocurre algo de interés. Le da la vuelta a la comunicación, el cliente pasa a ser servidor y el servidor a cliente. El desencadenador devuelve inmediatamente un 202 si la petición es correcta o el 4XX apropiado se es incorrecta.
- Este esquema de funcionamiento tiene muchas ventajas en ambos extremos:
  - **Ahorro de recursos y tiempo**: con el sondeo se harán muchas llamadas "para nada", que no devolverán información relevante. Los dos extremos gastan recursos para hacer y responder a muchas llamadas que no tienen utilidad alguna (no nos llame, ya les llamaremos).
  - **Eliminación de los retrasos**: la aplicación usuaria recibirá una llamada en el momento exacto en el que se produce y no tendrá retrasos al próximo sondeo.
  - **Velocidad de las llamadas**: generalmente la llamada que se hace a un webhook es muy rápida porque solo se envía una pequeña información sobre el evento y se suele procesar asíncronamente. Muchas veces ni siquiera se espera por el resultado: se hace una llamada del tipo "fire and forget" (o sea, dispara y olvídate), pues se trata de notificar el evento y listo.



DOCUMENTACIÓN

Enfoque 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.

Beneficios de API First

- Los equipos de desarrollo pueden trabajar en paralelo.
  - Los equipos pueden simular APIs y probar sus dependencias en función de la definición de la API establecida.
- Reduce el coste de desarrollar aplicaciones
  - Las APIs y el código se pueden reutilizar en muchos proyectos diferentes.
- Aumenta la velocidad de desarrollo.
  - Gran parte del proceso de creación de API se puede automatizar mediante herramientas que permiten importar archivos de definición de API y generar el esqueleto del backend y el cliente frontend, así como un mocking server para las pruebas.

Beneficios de API First

- Asegura buenas experiencias de desarrollador
  - Las APIs bien diseñadas, bien documentadas y consistentes brindan experiencias positivas para los desarrolladores porque es más fácil reutilizar el código y los desarrollos integrados, reduciendo la curva de aprendizaje.
- Reduce el riesgo de fallos
  - Reduce el riesgo de fallos al facilitar las pruebas para garantizar que las APIs sean confiables, consistentes y fáciles de usar para los desarrolladores.

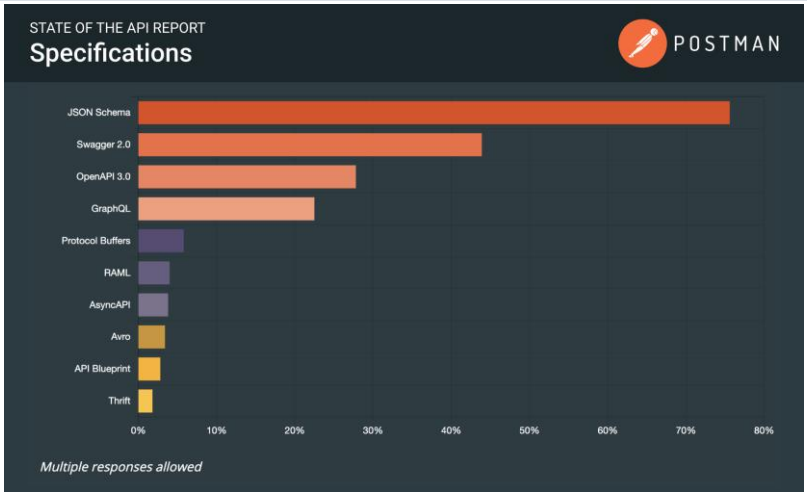
Documentar servicios Rest

- Dado que las API están diseñadas para ser consumidas, es importante asegurarse de que el cliente o consumidor pueda implementar rápidamente una API y comprender qué está sucediendo con ella. Desafortunadamente, muchas API hacen que la implementación sea extremadamente difícil, frustrando su propósito.
- La documentación es uno de los factores más importantes para determinar el éxito de una API, ya que la documentación sólida y fácil de entender hace que la implementación de la API sea muy sencilla, mientras que la documentación confusa, desincronizada, incompleta o intrincada hace que sea una aventura desagradable, una que generalmente conduce a desarrolladores frustrados a utilizar las soluciones de la competencia.
- Una buena documentación debe actuar como referencia y como formación, permitiendo a los desarrolladores obtener rápidamente la información que buscan de un vistazo, mientras también leen la documentación para obtener una comprensión de cómo integrar el recurso / método que están viendo.
- Con la expansión de especificaciones abiertas como OpenApi, RAML, … y las comunidades que las rodean, la documentación se ha vuelto mucho más fácil, aun así requiere invertir tiempo y recursos, todo ello con una cuidadosa planificación.

Documentar servicios Rest

- Web Application Description Language (WADL) (https://www.w3.org/Submission/wadl/)
  - Especificación de W3C, que la descripción XML legible por máquina de aplicaciones web basadas en HTTP (normalmente servicios web REST). Modela los recursos proporcionados por un servicio y las relaciones entre ellos. Está diseñado para simplificar la reutilización de servicios web basados en la arquitectura HTTP existente de la web. Es independiente de la plataforma y del lenguaje, tiene como objetivo promover la reutilización de aplicaciones más allá del uso básico en un navegador web.
- Spring REST Docs (https://spring.io/projects/spring-restdocs)
  - Documentación a través de los test (casos de uso), evita enterrar el código entre anotaciones.
- RAML (https://raml.org/)
  - RESTful API Modeling Language es una forma práctica de describir un API RESTful de una manera que sea muy legible tanto para humanos como para maquinas.
- Open API (anteriormente Swagger)
  - Especificación para describir, producir, consumir y visualizar servicios web RESTful. Es el más ampliamente difundido y cuenta con un ecosistema propio.
- JSON Schema (https://json-schema.org/)
  - JSON Schema es una especificación para definir, anotar y validar las estructuras de datos JSON.

Especificaciones mas utilizadas



Swagger

https://swagger.io/

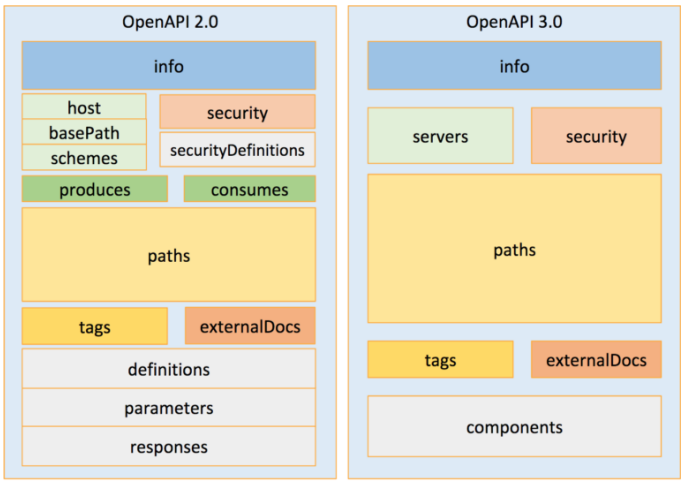
- Swagger (OpenAPI Specification) es una especificación abierta y su correspondiente implementación para probar y documentar servicios REST. Uno de los objetivos de Swagger es que podamos actualizar la documentación en el mismo instante en que realizamos los cambios en el servidor.
- Un documento Swagger es el equivalente de API REST de un documento WSDL para un servicio web basado en SOAP.
- El documento Swagger especifica la lista de recursos disponibles en la API REST y las operaciones a las que se puede llamar en estos recursos.
- El documento Swagger especifica también la lista de parámetros de una operación, que incluye el nombre y tipo de los parámetros, si los parámetros son necesarios u opcionales, e información sobre los valores aceptables para estos parámetros.

OpenAPI

https://www.openapis.org/

- OpenAPI es un estándar para definir contratos de API. Los cuales describen la interfaz de una serie de servicios que vamos a poder consumir por medio de una signatura. Conocido previamente como Swagger, ha sido adoptado por la Linux Foundation y obtuvo el apoyo de compañías como Google, Microsoft, IBM, Paypal, etc. para convertirse en un estándar para las APIs REST.
- Las definiciones de OpenAPI se pueden escribir en JSON o YAML. La versión actual de la especificación es la 3.0.3 y orientada a YAML y la versión previa la 2.0, que es idéntica a la especificación 2.0 de Swagger antes de ser renombrada a “Open API Specification”.
- Actualmente nos encontramos en periodo de transición de la versión 2 a la 3, sin soporte en muchas herramientas.

Cambio de versión



Sintaxis

- Un documento de OpenAPI que se ajusta a la especificación de OpenAPI es en sí mismo un objeto JSON con propiedades, que puede representarse en formato JSON o YAML.
- YAML es un lenguaje de serialización de datos similar a XML pero que utiliza el sangrado para indicar el anidamiento, estableciendo la estructura jerárquica, y evitar la necesidad de tener que cerra los elementos.
- Para preservar la capacidad de ida y vuelta entre los formatos YAML y JSON, se RECOMIENDA la versión 1.2 de YAML junto con algunas restricciones adicionales:
  - Las etiquetas DEBEN limitarse a las permitidas por el conjunto de reglas del esquema JSON .
  - Las claves utilizadas en los mapas YAML DEBEN estar limitadas a una cadena escalar, según lo definido por el conjunto de reglas del esquema YAML Failsafe.
- Todos los nombres de propiedades o campos de la especificación distinguen entre mayúsculas y minúsculas. Esto incluye todas los propiedades que se utilizan como claves asociativas, excepto donde se indique explícitamente que las claves no distinguen entre mayúsculas y minúsculas .
- El esquema expone dos tipos de propiedades:
  - propiedades fijas: tienen el nombre establecido en el estándar
  - propiedades con patrón: sus nombres son de creación libre pero deben cumplir una expresión regular (patrón) definida en el estándar y deben ser únicos dentro del objeto contenedor.

Sintaxis

- El sangrado utiliza espacios en blanco, no se permite el uso de caracteres de tabulación.
- Los miembros de las listas van entre corchetes ([ ]) y separados por coma espacio (,), o uno por línea con un guion (-) inicial.
- Los vectores asociativos se representan usando los dos puntos seguidos por un espacio, "clave: valor", bien uno por línea o entre llaves ({ }) y separados por coma seguida de espacio (,).
- Un valor de un vector asociativo viene precedido por un signo de interrogación (?), lo que permite que se construyan claves complejas sin ambigüedad.
- Los valores sencillos (o escalares) por lo general aparecen sin entrecomillar, pero pueden incluirse entre comillas dobles ("), o apostrofes (').

Sintaxis

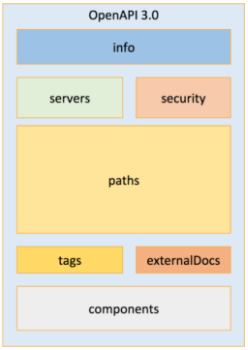
- Los comentarios vienen encabezados por la almohadilla (#) y continúan hasta el final de la línea.
- Es sensible a mayúsculas y minúsculas, todas las propiedades (palabras reservadas) de la especificación deben ir en minúsculas y terminar en dos puntos (:).
- Las propiedades requieren líneas independiente, su valor puede ir a continuación en la misma línea (precedido por un espacio) o en múltiples líneas (con sangrado)
- Las descripciones textuales pueden ser multilínea y admiten el dialecto CommonMark de Markdown para una representación de texto enriquecido. El HTML es compatible en la medida en que lo proporciona CommonMark (Bloques HTML en la Especificación 0.27 de CommonMark).
- $ref permite sustituir, reutilizar y enlazar una definición local con una externa.

CommonMark de Markdown

- Requiere doble y triple salto de línea para saltos de párrafos y cierre de bloques. Dos espacios al final de la línea lo convierte en salto de línea.
- Regla horizontal (separador): ---
- Énfasis: *cursiva* **negrita** ***cursiva y negrita***
- Enlaces: <http://www.example.com> [texto](http://www. example.com)
- Imágenes: ![Image](http://www.example.com/logo.png "icon")
- Citas: > Texto de la cita con sangría
- Bloques de códigos: `Encerrados entre tildes graves`
- Listas: Dos espacios en blanco por nivel de sangrado.
  + Listas desordenadas 1. Listas ordenadas
- Encabezado: dos líneas debajo del texto, añadir cualquier número de caracteres = para el nivel de título 1, <h1> … <h6> (el # es interpretado como comentario).

Estructura básica

- Un documento de OpenAPI puede estar compuesto por un solo documento o dividirse en múltiples partes conectadas a discreción del usuario. En el último caso, los campos $ref deben utilizarse en la especificación para hacer referencia a esas partes.
- Se recomienda que el documento raíz de OpenAPI se llame: openapi.json u openapi.yaml.
- La especificación de la API se puede dividir en 3 secciones principales:
  - Meta información
  - Elementos de ruta (puntos finales):
    - Parámetros de las solicitud
    - Cuerpo de las solicitud
    - Respuestas
  - Componentes reutilizables:
    - Esquemas (modelos de datos)
    - Parámetros
    - Respuestas
    - Otros componentes



Estructura básica

```
openapi: 3.0.0
info:
  title: Sample API
  description: Optional multiline or single-line description in ...
  version: 0.1.9
servers:
  - url: http://api.example.com/v1
    description: Optional server description, e.g. Main (production) server
  - url: http://staging-api.example.com
    description: Optional server description, e.g. Internal staging server for testing
paths:
  /users:
    get:
      summary: Returns a list of users.
      description: Optional extended description in CommonMark or HTML.
      responses:
        '200':    # status code
          description: A JSON array of user names
          content:
            application/json:
              schema:
                type: array
                items:
                  type: string
```

Estructura básica (cont)

```
components:
  schemas:
    User:
      properties:
        id:
          type: integer
        name:
          type: string
      # Both properties are required
      required:
        - id
        - name
  securitySchemes:
      BasicAuth:
        type: http
        scheme: basic
security:
  - BasicAuth: []
```


Prologo

- Cada definición de API debe incluir la versión de la especificación OpenAPI en la que se basa el documento en la propiedad openapi.
- La propiedad info contiene información de la API:
  - title es el nombre de API.
  - description es información extendida sobre la API.
  - version es una cadena arbitraria que especifica la versión de la API (no confundir con la revisión del archivo o la versión del openapi).
  - también admite otras palabras clave para información de contacto (nombre, url, email), licencia (nombre, url), términos de servicio (url) y otros detalles.
- La propiedad servers especifica el servidor API y la URL base. Se pueden definir uno o varios servidores (elementos precedidos por -).
- Con la propiedad externalDocs se puede referenciar la documentación externa adicional.

Rutas

- La sección paths define los puntos finales individuales (rutas) en la API y los métodos (operaciones) HTTP admitidos por estos puntos finales.
- Las ruta es relativa a la ruta del objeto Server.
- Los parámetros de la ruta se pueden usar para aislar un componente específico de los datos con los que el cliente está trabajando. Los parámetros de ruta son parte de la ruta y se expresan entre llaves (/users/{userId}), participan en la jerarquía de la URL y, por lo tanto, se apilan secuencialmente. Los parámetros de ruta deben describirse obligatoriamente en parameters (común para todas las operaciones) o a nivel de operación individual.
- No puede haber dos rutas iguales o ambiguas, que solo se diferencian por el parámetro de ruta.
- La definición de la ruta pude tener con un resumen (summary) y una descripción (description).
- Una ruta debe contar con un conjunto de operaciones, al menos una.
- Opcionalmente, servers permite dar una matriz alternativa de server que den servicio a todas las operaciones en esta ruta.

Rutas

```
'/users/{id}/roles':
  get:
    summary: Returns a list of users's roles.
    operationId: getDirecciones
    parameters:
      - in: path
        name: id
        description: User ID
        required: true
        schema:
          type: number
      - in: query
        name: size
        schema:
          type: string
          enum: [long, medium, short]
        required: true
      - in: query
        name: page
        schema:
          type: integer
          minimum: 0
          default: 0
```

```
responses:
  '200':
    description: List of roles
    content:
      application/json:
        schema:
          $ref: '#/components/schemas/Roles'
  '400':
    description: Bad request. User ID must be an
      integer and larger than 0.
  '401':
    description: Authorization information is missing
      or invalid.
  '404':
    description: A user with the specified ID was not
      found.
  '5XX':
    description: Unexpected error.
  default:
    description: Default error sample response
```

Operaciones

- Describe una única operación de API en una ruta y se identifica con el nombre del método HTTP: get, put, post, delete, options, head, patch, trace.
- Una definición de operación puede incluir un breve resumen de lo que hace (summary), una explicación detallada del comportamiento (description), una referencia a documentación externa adicional (externalDocs), un identificador único para su uso en herramientas y bibliotecas (operationId) y si está obsoleta y debería dejar de usarse (deprecated).
- Las operaciones pueden tener parámetros pasados a través de la ruta URL (/users/{userId}), cadena de consulta (/users?role=admin), encabezados (X-CustomHeader: Value) o cookies (Cookie: debug=0).
- Si la petición (POST, PUT, PATCH) envía un cuerpo en la solicitud (body), la propiedad requestBody permite describir el contenido del cuerpo y el tipo de medio.
- Para cada las respuestas de la operación, se pueden definir los posibles códigos de estado y el schema del cuerpo de respuesta. Los esquemas pueden definirse en línea o referenciarse mediante $ref. También se pueden proporcionar ejemplos para los diferentes tipos de respuestas.

Parámetros

- Un parámetro único se define mediante una combinación de nombre (name) y ubicación (in: "query", "header", "path" o "cookie") en la propiedad parameters.
- Opcionalmente puede ir acompañado por una breve descripción del parámetro (description), si es obligatorio (required), si permite valores vacíos (allowemptyvalue) y si está obsoleto y debería dejar de usarse (deprecated).
- Las reglas para la serialización del parámetro se especifican dos formas:
  - Para los escenarios más simples, con schema y style se puede describir la estructura y la sintaxis del parámetro.
  - Para escenarios más complejos, la propiedad content puede definir el tipo de medio y el esquema del parámetro.
- Un parámetro debe contener la propiedad schema o content, pero no ambas.
- Se puede proporcionar un example o examples pero debe seguir la estrategia de serialización prescrita para el parámetro.

Parámetros

```
paths:
  /users:
    get:
      description: Returns a list of users
      parameters:
        - name: rows
          in: query
          description: Limits the number of items on a page
          schema:
            type: integer
        - name: page
          in: query
          description: Specifies the page number of the users to be displayed
          schema:
            type: integer
```


Cuerpo de la solicitud

- En versiones anteriores, el cuerpo de la solicitud era un parámetro mas in: body.
- Actualmente se utiliza la propiedad requestBody con una breve descripción (description) y si es obligatorio para la solicitud (required), ambas opcionales.
- La descripción del contenido (content) es obligatoria y se estructura según los tipos de medios que actúan como identificadores. Para las solicitudes que coinciden con varias claves, solo se aplica la clave más específica (text/plain → text/* → */*).
- Por cada tipo de medio se puede definir el esquema del contenido de la solicitud (schema), uno (example) o varios (examples) ejemplos y la codificación (encoding).
- El requestBody sólo se admite en métodos HTTP donde la especificación HTTP 1.1 RFC7231 haya definido explícitamente semántica para cuerpos de solicitud.

Cuerpo de la solicitud

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody:
        required: true
        content:
          application/json:
            schema:
              type: object
              required:
                - username
              properties:
                username:
                  type: string
                password:
                  type: string
                  format: password
                name:
                  type: string
```


Respuestas

- Es obligaría la propiedad responses con la lista de posibles respuestas que se devuelven al ejecutar esta operación.
- No se espera necesariamente que la documentación cubra todos los códigos de respuesta HTTP posibles porque es posible que ni se conozcan de antemano. Sin embargo, se espera que cubra la respuesta de la operación cuando tiene éxito y cualquier error previsto.
- La posibles respuestas se identifican con el código de respuesta HTTP. Con default se puede definir las respuesta por defecto para todos los códigos HTTP que no están cubiertos por la especificación individual.
- La respuesta cuenta con una breve descripción de la respuesta (description) y, opcionalmente, el contenido estructurado según los tipos de medios (content), los encabezados (headers) y los enlaces de operaciones que se pueden seguir desde la respuesta (links).

Respuestas

```
paths:
  /users:
    post:
      description: Lets a client post a new user
      requestBody: # …
      responses:
        '201':
          description: Successfully created a new user
        '400':
          description: Invalid request
          content:
            application/json:
              schema:
                type: object
                properties:
                  code:
                    type: integer
                  message:
                    type: string
```


Etiquetas

- Las etiquetas son metadatos adicionales que permiten organizar la documentación de la especificación de la API y controlar su presentación. Las etiquetas se pueden utilizar para la agrupación lógica de operaciones por recursos o cualquier otro calificador. El orden de las etiquetas se puede utilizar para reflejar un orden en las herramientas de análisis.
- Cada nombre de etiqueta en la lista debe ser único (name) y puede ir acompañado por una explicación detallada (description) y una referencia a documentación externa adicional (externalDocs).
- Las etiquetas se pueden declarar en la propiedad tags del documento:

```
tags:
  - name: security-resource
    description: Gestión de la seguridad
```

Etiquetas

- Las etiquetas se aplican en la propiedad tags de las operaciones:

```
paths:
  /users:
    get:
      tags:
        - security-resource
  /roles:
    get:
      tags:
        - security-resource
        - read-only-resource
```

- No es necesario declarar todas las etiquetas, se pueden usar directamente pero no se podrá dar información adicional y se mostraran ordenadas al azar o según la lógica de las herramientas.

Componentes

- La propiedad global components permite definir las estructuras de datos comunes utilizadas en la especificación de la API: Contiene un conjunto de objetos reutilizables para diferentes aspectos de la especificación.
- Todos los objetos definidos dentro del objeto de componentes no tendrán ningún efecto en la API a menos que se haga referencia explícitamente a ellos desde propiedades fuera del objeto de componentes.
- La sección components dispones de propiedades para schemas, responses, parameters, examples, requestBodies, headers, securitySchemes, links y callbacks.
- Se puede hacer referencia a ellos con $ref cuando sea necesario. $ref acepta referencias internas con # o externas con el nombre de un fichero. La referencia debe incluir la trayectoria para encontrar el elemento referenciado:

  $ref: '#/components/schemas/Rol'

  $ref: responses.yaml#/404Error
- El uso de referencias permite la reutilización de elementos ya definidos, facilitando la mantenibilidad y disminuyendo sensiblemente la longitud de la especificación, por lo que se deben utilizar extensivamente. Las referencias no interfieren con la presentación en el UI.

Esquemas de datos

- Los schemas definen los modelos de datos consumidos y devueltos por la API.
- Los tipos de datos OpenAPI se basan en un subconjunto extendido del JSON Schema Specification Wright Draft 00 (también conocido como Draft 5).
- Los tipos base son string, number, integer, boolean, array y object.
- Con la propiedad format se pueden especificar otros tipos especiales partiendo de los tipos base: long, float, double, byte, binary, date, dateTime, password.
- Los tipos array se definen como una colección de ítems y en dicha propiedad se define el tipo y la estructura de los elementos que lo componen. Los objetos son un conjunto de propiedades, cada una definida dentro de properties.
- Cada tipo y propiedad se identifica por un nombre que no debe estar repetido en su ámbito.
- Cada propiedad puede definir description, default, minimum, maximum, maxLength, minLength, pattern, required, readOnly, …
- Para una propiedad se pueden definir varios tipos (tipos mixtos o unión).
- Los tipos pueden hacer referencia a otros tipos.

Tipos de datos

type	format	Comentarios
boolean		Booleanos: true y false
integer	int32	Enteros con signo de 32 bits
integer	int64	Enteros con signo de 64 bits (también conocidos como largos)
number	float	Reales cortos
number	double	Reales largos
string		Cadenas de caracteres
string	password	Una pista a las IU para ocultar la entrada.
string	date	Según lo definido por full-date RFC3339 (2018-11-13)
string	date-time	Según lo definido por date-time- RFC3339 (2018-11-13T20:20:39+00:00)
string	byte	Binario codificados en base64
string	binary	Binario en cualquier secuencia de octetos
array		Colección de items
object		Colección de properties

Propiedades de los objetos de esquema

- type: integer, number, boolean, string, array, object
- format: long, float, double, byte, binary, date, dateTime, password
- title: Nombre a mostrar en el UI
- description: Descripción de su uso
- maximum: Valor máximo
- exclusiveMaximum: Valor menor que
- minimum: Valor mínimo
- exclusiveMinimum: Valor mayor que
- multipleOf: Valor múltiplo de
- maxLength: Longitud máxima
- minLength: Longitud mínima
- pattern: Expresión regular del patrón
- deprecated: Si está obsoleto y debería dejar de usarse
- nullable: Si acepta nulos
- default: Valor por defecto
- enum: Lista de valores con nombre
- example: Ejemplo de uso
- externalDocs: referencia a documentación externa adicional
- items: Definición de los elementos del array
- maxItems: Número máximo de elementos
- minItems: Número mínimo de elementos
- uniqueItems: Elementos únicos
- properties: Definición de las propiedades del objeto,
- maxProperties: Número máximo de propiedades
- minProperties: Número mínimo de propiedades
- readOnly: propiedad de solo lectura
- writeOnly: propiedad de solo escritura
- additionalProperties: permite referenciar propiedades adicionales
- required: Lista de propiedades obligatorias

Modelos de entrada y salida

```
components:
  schemas:
    Roles:
      type: array
      items:
        $ref: '#/components/schemas/Rol'
    Rol:
      type: object
      description: Roles de usuario
      properties:
        rolId:
          type: integer
          format: int32
          minimum: 0
          maximum: 255
        name:
          type: string
          maxLength: 20
        description:
          type: string
        last_updated:
          type: string
          format: dateTime
          readOnly: true
        level:
            type: string
            description: Nivel de permisos
            enum:
            - high
            - normal
            - low
            default: normal
      required:
      - rolId
      - name
```

Autenticación

- La propiedad securitySchemes de components y la propiedad security del documento se utilizan para describir y establecer los métodos de autenticación utilizados en la API.
- securitySchemes define los esquemas de seguridad que pueden utilizar las operaciones. Los esquemas admitidos son la autenticación HTTP, una clave API (ya sea como encabezado, parámetro de cookie o parámetro de consulta), los flujos comunes de OAuth2 (implícito, contraseña, credenciales de cliente y código de autorización) tal y como se define en RFC6749 y OpenID Connect Discovery. Cada esquema cuenta con un identificador, un tipo (type: "apiKey", "http", "oauth2", "openIdConnect") y opcionalmente puede ir acompañado por una breve descripción (description).
- Según el tipo seleccionado será obligatorio:
  - apiKey: ubicación (in: "query", "header" o "cookie") y su nombre (name) de parámetro, encabezado o cookie.
  - http: esquema de autorización HTTP que se utilizará en el encabezado Authorization (scheme): Basic, Bearer, Digest, OAuth, … y, si es Bearer, prefijo del token de portador (bearerFormat).
  - openIdConnect: URL de OpenId Connect para descubrir los valores de configuración de OAuth2 (openIdConnectUrl).
  - oauth2: objeto que contiene información de configuración para los tipos de flujo admitidos (flows).
- La propiedad security enumera los esquemas de seguridad que se pueden utilizar en la API.

Autenticación

```
components:
  securitySchemes:
    BasicAuth:
      type: http
      scheme: basic
    JWTAuth:
      type: http
      scheme: bearer
      bearerFormat: JWT
    ApiKeyAuth:
      type: apiKey
      name: x-api-key
      in: header
    ApiKeyQuery:
      type: apiKey
      name: api-key
      in: query
security:
  - ApiKeyAuth: []
  - ApiKeyQuery: []
```

Ejemplos

- Los ejemplos son fundamentales para la correcta comprensión de la documentación. La especificación permite proporcionar uno (example) o varios (examples) ejemplos asociados a las estructuras de datos.
- Por cada uno se puede dar un resumen del ejemplo (summary), una descripción larga (description), el juego de valores de las propiedades de la estructura (value) o una URL que apunta al ejemplo literal para ejemplos que no se pueden incluir fácilmente en documentos JSON o YAML (externalValue). value y externalValue son mutuamente excluyentes. Cuando son varios ejemplos deber estar identificados por un nombre único.

```
examples:
  first-page:
    summary: Primera página
    value: 0
  second-page:
    summary: Segunda página
    value: 1
```

- Los ejemplos pueden ser utilizados automáticamente por las herramientas de UI y de generación de pruebas.

Ecosistema Swagger

- Swagger Open Source Tools (https://swagger.io/)
  - Swagger UI: Generar automáticamente la documentación desde la definición de OpenAPI para la interacción visual y un consumo más fácil.
  - Swagger Editor: Diseñar APIs en un potente editor de OpenAPI que visualiza la definición y proporciona comentarios de errores en tiempo real.
  - Swagger Codegen: Crear y habilitar el consumo de su API generando la fontanería del servidor y el cliente.
- Swagger Pro Tools
  - SwaggerHub: La plataforma de diseño y documentación para equipos e individuos que trabajan con la especificación OpenAPI.
  - Swagger Inspector: La plataforma de pruebas y generación de documentación de las APIs
- https://openapi.tools/

springdoc-openapi

https://springdoc.org/ *(sustituto de SpringFox)*

- La biblioteca Java springdoc-openapi ayuda a automatizar la generación de documentación de la API utilizando proyectos de spring boot. springdoc-openapi funciona examinando la aplicación en tiempo de ejecución para inferir la semántica de la API en función de las configuraciones de Spring, la estructura de clases y varias anotaciones.
- Genera automáticamente documentación en API de formato JSON/YAML y HTML. Esta documentación se puede completar con comentarios usando anotaciones swagger-api.
  - https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations
- Esta biblioteca admite:
  - OpenAPI 3
  - Spring Boot (1 y 2: v1, 3: v2), Actuator, Mvc, WebFlux, Security, Data Rest y Hateoas
  - JSR-303, específicamente para @NotNull, @Min, @Max y @Size.
  - Swagger-ui
  - OAuth 2
  - Imágenes nativas de GraalVM

Instalación (v3.0)

- Se debe añadir la dependencia Maven del starter de springdoc.

```
<dependency>
    <groupId>org.springdoc</groupId>
    <artifactId>springdoc-openapi-starter-webmvc-ui</artifactId>
    <version>2.2.0</version>
</dependency>
```

- Se pueden cambiar las configuraciones por defecto en el fichero application.properties

```
springdoc.swagger-ui.use-root-path=true
springdoc.show-actuator=true
springdoc.swagger-ui.path=/open-api
springdoc.packagesToScan=com.example.applications.resources
springdoc.pathsToMatch=/v1/**, /api/**, /auto/**
```

- Para acceder a la documentación:
  - http://localhost:8080/swagger-ui/index.html (versión HTML)
  - http://localhost:8080/v3/api-docs (versión JSON)
  - http://localhost:8080/v3/api-docs.yaml (versión YAML)

Soporte adicional (v3.0)

- Para habilitar la compatibilidad con javadoc, que mejora el soporte de etiquetas y comentarios javadoc, es necesario incluir:

```
<dependency>
    <groupId>com.github.therapi</groupId>
    <artifactId>therapi-runtime-javadoc</artifactId>
    <version>0.13.0</version>
</dependency>
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <annotationProcessorPaths>
          <path>
            <groupId>com.github.therapi</groupId>
            <artifactId>therapi-runtime-javadoc-scribe</artifactId>
            <version>0.15.0</version>
          </path>
        </annotationProcessorPaths>
      </configuration>
    </plugin>
  </plugins>
</build>
```


Anotar el modelo

- **@Schema**:
  - documenta la entidad, con nombre alternativo al de la clase (name) y una descripción más larga (description).

```
@Schema(name = "Entidad Personas", description = "Información completa de la personas")
public class Persona {
```

  - documenta las propiedades, con una descripción (description), ocultarla (hidden) y el resto de las propiedades de los objetos de esquema como format, maximum, exclusiveMaximum, minimum, exclusiveMinimum, multipleOf, maxLength, minLength, pattern, deprecated, nullable, default, required.

```
@ Schema(description = "Identificador de la persona", mínimum=0, required = true)
private Long id;
```

  - Dispone de soporte para las anotaciones de validación de bean JSR-303, específicamente para @NotNull (@NotEmpty, @NotBlank), @Min, @Max, @Size y @Pattern, que documenta automáticamente.

Anotar el modelo

- **@ArraySchema**: se usa para definir un esquema de tipo "array". Las anotaciones ArraySchema y Schema no pueden coexistir. Los parámetros arraySchema permite documentar el array dentro de su esquema y schema documenta los elementos dentro del array. Están disponibles propiedades adicionales como uniqueItems, maxItems o maxItems.

```
@ArraySchema(arraySchema = @Schema(description = "Lista de
        correos electrónicos"), schema = @Schema(format = "email",
        maxLength = 100), maxItems = 3, uniqueItems = true)
private List<String> correos;
```


Anotar el servicio

- **@Tag**: documenta el servicio REST en si. Permite establecer el nombre y la descripción.

```
@RestController
@Tag(name = "Microservice Personas", description = "API que permite el mantenimiento de personas")
public class PersonasResource {
```

- **@Operation**: documenta cada método del servicio con el summary, una breve descripción, y description. También puede definir:
  - etiquetas:@Tag
  - documentos externos: @ExternalDocumentation
  - parámetros: @Parameter
  - cuerpo de solicitud: @RequestBody
  - respuestas: @ApiResponse
  - seguridad: @SecurityRequirement
  - servidores: @Server
  - extensiones: @Extension
  - ocultar: @Hidden

```
@GetMapping(path = "/{id}")
@Operation(summary="Buscar una persona",  description = "Devuelve una persona por su identificador")
public Persona getOne(@Parameter(description = "Identificador de la persona", required = true) @PathVariable int id) {
```

Anotar el servicio

- @**Parameter**: documenta los parámetros de cada método del servicio.

```
public Persona getOne(@Parameter(description = "Identificador de la persona", required = true) @PathVariable int id) {
```

- @**ParameterObject**: puede extraer cada campo del objeto parámetro como un parámetro de solicitud independiente.

```
public Page<Persona> getAll(@ParameterObject Pageable page) {
```

- @**RequestBody**: documenta el cuerpo de solicitud de la operación y permite definir propiedades adicionales a las disponibles en @Parameter

```
@PostMapping
public ResponseEntity<Object> create(@Valid @RequestBody(description = "Datos de la
        persona", required = true, content = @Content(mediaType = "application/json",
        schema = @Schema(implementation = Persona.class))) Persona item) throws
        BadRequestException, DuplicateKeyException, InvalidDataException {
```


Anotar el servicio

- @**ApiResponse**: documenta las posibles respuestas del método, con un mensaje explicativo. El uso de @ResponseStatus en métodos en una clase @RestControllerAdvice generará automáticamente la documentación para los códigos de respuesta. @ApiResponses es un contenedor de respuestas.

```
@ApiResponse(responseCode = "200", description = "Persona encontrada")
@ApiResponse(responseCode = "404", description = "Persona no encontrada")
public Persona getOne(@PathVariable int id) throws NotFoundException {
```

- @**Hidden**: Marca un recurso u operación como oculto en la documentación.

```
@Hidden
@RestController
public class PersonasResource {
@Hidden
public Persona getOne(@PathVariable int id) throws NotFoundException {
```

Anotar el servicio

```
@PutMapping("/{id}")
@ResponseStatus(HttpStatus.ACCEPTED)
@Operation(summary = "Modificar una persona",
    description = "Sustituye una persona con los nuevos datos, los identificadores deben coincidir.",
    tags = {"Microservice Personas", "Modificaciones"},
    parameters = {
        @Parameter(in = ParameterIn.PATH, name = "id", required = true, description = "Identificador de la persona")
    },
    requestBody = @io.swagger.v3.oas.annotations.parameters.RequestBody(description = "Datos de la persona", required = true, content
        = @Content(mediaType = "application/json", schema = @Schema(implementation = Persona.class))),
    responses = {
        @ApiResponse(responseCode = "202", description = "Persona modificada"),
        @ApiResponse(responseCode = "400", description = "Datos invalidos", content = @Content(mediaType = "application/json",
            schema = @Schema(implementation = ErrorMessage.class))),
        @ApiResponse(responseCode = "404", description = "Persona no encontrada", content = @Content(mediaType =
            "application/json", schema = @Schema(implementation = ErrorMessage.class)))
    }
)
public void update(@PathVariable int id, @Valid @RequestBody ActorShort item) throws BadRequestException, NotFoundException,
InvalidDataException {
```


Configuración (OpenApi)

```
@OpenAPIDefinition(
    info = @Info(title = "Microservicio: Demos",  version = "1.0",
        description = "**Demos** de Microservicios.",
        license = @License(name = "Apache 2.0", url = "https://www.apache.org/licenses/LICENSE-2.0.html"),
        contact = @Contact(name = "Javier Martín", url = "https://github.com/jmagit", email = "support@example.com")
    ),
    externalDocs = @ExternalDocumentation(description = "Documentación del proyecto", url = "https://github.com/jmagit/curso")
)
public class PrincipalApplication implements CommandLineRunner {

@Configuration
public class OpenApiConfiguration {
    @Bean
    public OpenAPI springShopOpenAPI() {
        return new OpenAPI()
            .info(new Info().title("Microservicio: Demos")
                .description("**Demos** de Microservicios.").version("1.0")
                .license(new License().name("Apache 2.0").url("https://www.apache.org/licenses/LICENSE-2.0.html"))
                .contact(new Contact().name("Javier Martín").url("https://github.com/jmagit").email("support@example.com"))
            ).externalDocs(new ExternalDocumentation().description("Documentación del proyecto").url("https://github.com/jmagit/curso"));
    }
}
```

Configuración (swagger-ui)

- Para ordenar los servicios y sus operaciones, en el fichero application.properties

```
springdoc.swagger-ui.tagsSorter=alpha
springdoc.swagger-ui.operationsSorter=alpha
springdoc.swagger-ui.docExpansion=none
```

- Para ordenar los modelos del esquema:

```
@Bean
public OpenApiCustomiser sortSchemasAlphabetically() {
   return openApi -> {
      var schemas = openApi.getComponents().getSchemas();
      openApi.getComponents().setSchemas(new TreeMap<>(schemas));
   };
}
```


Archivo de propiedades

- Hay que crear en resources un archivo de propiedades, por ejemplo, openapi.properties
- Insertar los mensajes deseados como pares clave-valor donde la clave se usará como marcador de posición:

```
person.id.value = Identificador único de la persona
```

- En lugar del texto en la anotación, se inserta un marcador de posición:

```
@Parameter(description = "${person.id.value}", required = true)
```

- Hay que registrar el archivo de propiedades de la configuración a nivel de clase:

```
@PropertySource ("classpath: openapi.properties")
```

Seguridad

- @SecurityScheme: Permite definir los esquemas de seguridad, una por definición. La definición es global y complementan a la anotación @OpenAPIDefinition.

```
@SecurityScheme(name = "bearerAuth", type = SecuritySchemeType.HTTP, scheme =
"bearer", bearerFormat = "JWT")

return new OpenAPI()
        .components(new Components()
        .addSecuritySchemes("bearerAuth",
        new SecurityScheme().type(SecurityScheme.Type.HTTP)
            .scheme("bearer").bearerFormat("JWT")));
```

- @SecurityRequirement: A nivel de operación, indica que requiere autorización y el esquema de autenticación:

```
@SecurityRequirement(name = "bearerAuth")
```


Que debe incluir

- Una explicación clara de lo que hace el método / recurso.
- Una lista de los parámetros utilizados en este recurso / método,
- Posibles respuestas, que comparten información importante con los desarrolladores, incluidas advertencias y errores
- Descripción de los tipos, formatos especial, reglas y restricciones.
- Una invocación y una respuesta de ejemplo, incluido los cuerpos con los media-type correspondientes.
- Ejemplos de código para varios lenguajes, incluido todo el código necesario (por ejemplo, Curl con PHP, así como ejemplos para Java, .Net, Ruby, etc.)
- Ejemplos de SDK (si se proporcionan SDK) que muestren cómo acceder al recurso / método utilizando el SDK para los lenguajes en que se suministra.
- Experiencias interactivas para probar las llamadas API.
- Preguntas frecuentes / escenarios con ejemplos de código
- Enlaces a recursos adicionales (otros ejemplos, blogs, etc.)
- Una sección de comentarios donde los usuarios pueden compartir / discutir el código.

CONSULTAS ENTRE SERVICIOS

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.

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/

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();
```

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.

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();
```

RestTemplate

- La RestTemplate proporciona un API de nivel superior sobre las bibliotecas de cliente HTTP y facilita la invocación de los endspoint 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;
```


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.

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.

RestTemplate

- Para recuperar uno:

```
PersonaDTO rslt = 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> rslt = response.getBody();
```

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);
```


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
  - UnknownHttpStatusCodeException: en caso de un estado HTTP desconocido.
- Para vigilar las excepciones:

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

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();
```


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.

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;
```


@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.

@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);
}
```


@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 des serializado
  - 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);
}
```

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 basada 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

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 le proceso de equilibrio de carga antes de enviarla.

Clientes 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 {
```


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.

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}
```


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.

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/*, /
```


Estilos de comunicación

MENSAJERÍA

Mensajería y contextos de dominio

- En términos de integración, cuando se construyen estructuras de comunicación entre los diferentes procesos, es común ver productos y enfoques que ponen el énfasis de la inteligencia en el mecanismo de comunicación en sí.
- Normalmente los productos de ESB (Enterprise Service Bus) son un ejemplo donde generalmente se incluyen sofisticadas estructuras para el ruteo de mensajes, la coreografía, la transformación, la aplicación de reglas de negocio, etc.
- En términos de diseño, también es común ver que a la hora de definir dónde colocamos cierto código que representa reglas de negocio, a veces, decidimos hacerlo dentro del ESB por conveniencia o rapidez. Entonces, convertimos al ESB en un elemento de integración clave para el funcionamiento del proceso en su totalidad, y esa excesiva inteligencia puesta en un único componente (la tubería), torna al sistema en algo más frágil que antes.

Mensajería y contextos de dominio

- Las aplicaciones creadas con microservicios pretenden ser tan disociadas y cohesivas como sean posible, ellas poseen su propia lógica de dominio y actúan más como filtros en el clásico sentido Unix:
  - recibe una solicitud, aplica la lógica apropiada y produce una respuesta.
  - estos pasos son coordinados utilizando protocolos REST simples en lugar de protocolos complejos WS-BPEL o la coordinación por una herramienta central.
- Los dos protocolos que se utilizan con mayor frecuencia son:
  - Petición/respuesta de HTTP con recursos API
  - Mensajería liviana, como puede ser RabbitMQ o ActiveMQ.
- Estos principios y protocolos hacen que los equipos de microservicios, a la hora de integrar servicios mas complejos, prefieran el concepto de coreografía, en lugar de orquestación.
- En orquestación, contamos con un cerebro central para orientar y conducir el proceso, al igual que el director de una orquesta.
- Con coreografía, le informamos a cada parte del sistema de su trabajo y se les deja trabajar en los detalles, como los bailarines en un ballet que se encuentran en su camino y reaccionan ante otros a su alrededor.

Mensajería y contextos de dominio

- La desventaja del enfoque de orquestación es que el orquestador se puede convertir en una autoridad de gobierno central demasiado fuerte y pasar a ser un punto central donde toda la lógica gira alrededor de él.
- En cambio, con un enfoque coreografiado, podríamos tener un servicio emitiendo un mensaje asincrónico, los servicios interesados sólo se suscriben a esos eventos y reaccionan en consecuencia.
- Este enfoque es significativamente más desacoplado. Si algún nuevo servicio está interesado en recibir los mensajes, simplemente tiene que subscribirse a los eventos y hacer su trabajo cuando sea necesario.
- La desventaja de la coreografía es que la vista explícita del proceso de negocio del modelado de proceso ahora sólo se refleja de manera implícita en el sistema.
- Esto implica que se necesita de trabajo adicional para asegurarse de que alguien pueda controlar y realizar un seguimiento de que hayan sucedido las cosas correctas.
- Para solucionar este problema, normalmente es necesario crear un sistema de monitoreo que refleje explícitamente la vista del proceso de negocio del modelado de procesos, pero que a su vez, haga un seguimiento de lo que cada uno de los servicios realiza como entidad independiente que le permite ver excepciones mapeadas al flujo de proceso más explícito.

Comandos vs Eventos

- Los mensajes se pueden clasificar en dos categorías principales:
  - Si el productor espera una acción del consumidor, ese mensaje es un comando.
  - Si el mensaje informa a los consumidor de que se ha realizado una acción, dicho mensaje es un evento.
- El productor envía un comando con la intención de que un consumidor realicen una operación dentro del ámbito de una transacción de negocio. Un comando es un mensaje de alto valor y debe entregarse al menos una vez. Si se pierde un comando, puede producirse un error en toda la transacción de negocio. Además, un comando no debe procesarse más de una vez. Si eso sucede, podría producirse una transacción errónea. Un cliente puede recibir pedidos o cargos duplicados. Los comandos se suelen usar para administrar el flujo de trabajo de una transacción de negocio de varios pasos. En función de la lógica de negocios, el productor puede esperar que el consumidor confirme el mensaje e informe de los resultados de la operación. En función de ese resultado, el productor puede elegir el plan de acción adecuado.
- Un evento es un tipo de mensaje que el productor genera para anunciar datos. El productor (conocido como publicador en este contexto) no espera que los eventos produzcan ninguna acción. Los consumidores interesados pueden suscribirse, escuchar eventos y tomar medidas en función de su escenario de consumo. Los eventos pueden tener varios suscriptores o ninguno. Dos suscriptores diferentes pueden reaccionar a un evento con diferentes acciones sin detectar al otro. El productor y el consumidor están acoplados de forma flexible y se administran de forma independiente. No se espera que el consumidor confirme el evento para el productor. Un consumidor que ya no está interesado en los eventos, puede finalizar la suscripción. El consumidor se quita de la canalización sin afectar al productor ni a la funcionalidad general del sistema.

Colas de mensajes

- En determinadas ocasiones, los servicios deben integrarse con otros actores, componentes o sistemas internos y externos, siendo necesario aportar o recibir información de ellos. En la mayoría de los casos, estas comunicaciones tienen que estar permanentemente disponibles, ser rápidas, seguras, asíncronas y fiables entre otros requisitos.
- Las colas de mensajes (MQ) solucionan estas necesidades, actuando de intermediario entre emisores y destinatarios, o en un contexto más definido, productores y consumidores de mensajes.
- Se pueden usar para reducir las cargas y los tiempos de entrega por parte de los servicios, ya que las tareas, que normalmente tardarían bastante tiempo en procesarse, se pueden delegar a un tercero cuyo único trabajo es realizarlas.
- El uso de colas de mensajes también es bueno cuando se desea distribuir un mensaje a múltiples destinatarios. Además aportan otros beneficios como:
  - Garantía de entrega y orden: Los mensajes se consumen, en el mismo orden que se llegaron a la cola, y son consumidos una única vez
  - Redundancia: Las colas persisten los mensajes hasta que son procesados por completo
  - Desacoplamiento: Siendo capas intermedias de comunicación entre procesos, aportan la flexibilidad en la definición de arquitectura de cada uno de ellos de manera separada, siempre que se mantenga una interfaz común
  - Escalabilidad: Con más unidades de procesamiento, las colas balancean su respectiva carga

RabbitMQ

- RabbitMQ (https://www.rabbitmq.com/) es el intermediario de mensajes de código abierto más ampliamente implementado. Dicho de forma simple, es un software donde se pueden definir colas de mensajes, las aplicaciones se pueden conectar a dichas colas y transferir/leer mensajes en ellas.
- RabbitMQ es ligero y fácil de implementar en las instalaciones y en la nube. Es compatible con múltiples protocolos de mensajería. RabbitMQ se puede implementar en configuraciones distribuidas y federadas para cumplir con los requisitos de alta disponibilidad y alta escalabilidad.
- RabbitMQ se ejecuta en muchos sistemas operativos y entornos de nube, y proporciona una amplia gama de herramientas para desarrolladores en los lenguajes más populares.
- La arquitectura básica de una cola de mensajes es simple. Hay aplicaciones clientes, llamadas productores, que crean mensajes y los entregan al intermediario (la cola de mensajes). Otras aplicaciones, llamadas consumidores, se conectan a la cola y se suscriben a los mensajes que se procesarán. Un mensaje puede incluir cualquier tipo de información.

RabbitMQ

- Un software puede ser un productor, consumidor, o productor y consumidor de mensajes simultáneamente. Los mensajes colocados en la cola se almacenan hasta que el consumidor los recupera y procesa.
- Los mensajes no se publican directamente en una cola, en lugar de eso, el productor envía mensajes a un exchange. Los exchanges son agentes de enrutamiento de mensajes, definidos por virtual host dentro de RabbitMQ. Un exchange es responsable del enrutamiento de los mensajes a las diferentes colas: acepta mensajes del productor y los dirige a colas de mensajes con ayuda de atributos de cabeceras, bindings y routing keys.
  - Un binding es un «enlace» que se configura para vincular una cola a un exchange
  - La routing key es un atributo del mensaje. El exchange podría usar esta clave para decidir cómo enrutar el mensaje a las colas (según el tipo de exchange)
- Los exchanges, las conexiones y las colas pueden configurarse con parámetros tales como durable, temporary y auto delete en el momento de su creación. Los exchanges declarados como durable sobrevivirán a los reinicios del servidor y durarán hasta que se eliminen explícitamente. Aquellos de tipo temporary existen hasta que RabbitMQ se cierre. Por último, los exchanges configurados como auto delete se eliminan una vez que el último objeto vinculado se ha liberado del exchange.

Instalación

- Descargar la última versión estable para Windows desde:
  - https://www.rabbitmq.com/download.html
- Para arrancar el servicio:
  - rabbitmqctl.bat start
- Para parar el servicio:
  - rabbitmqctl.bat stop
- Instalación del complemento de administración:
  - rabbitmq-plugins enable rabbitmq_management
- Para acceder al complemento de administración:
  - http://localhost:15672
  - El agente crea un usuario “guest” con contraseña “guest”.

RabbitMQ: Spring Boot

- Añadir al proyecto:
  - Messaging > Spring for RabbitMQ
- Configurar:

```
spring.rabbitmq.host=localhost
spring.rabbitmq.port=5672
spring.rabbitmq.username=guest
spring.rabbitmq.password=guest
#spring.rabbitmq.template.retry.enabled=true
#spring.rabbitmq.template.retry.initial-interval=2s
```

- AmqpTemplate y AmqpAdmin se configuran de forma automáticamente.
- Anotaciones:
  - @Queue: definición de una cola.
  - @Exchange: definición de un enrutador
  - @QueueBinding: Define una cola, el intercambio al que debe vincularse y una clave de enlace opcional, utilizada con @RabbitListener.
  - @RabbitListener: indica que es un método de escucha de una cola.

Manejar una cola

- Crear cola si no existe:

```
@Bean
public Queue myQueue() {
        return new Queue("mi-cola");
}
```

- Enviar un mensaje:

```
@Autowired
private AmqpTemplate amqp;

public void send(String cola, MessageDTO outMsg) {
        amqp.convertAndSend(cola, outMsg);
}
```

- Recibir mensajes:

```
@RabbitListener(queues = "mi-cola")
public void recive(MessageDTO inMsg) {
        // Procesar el mensaje recibido: inMsg
}
```

Formato del Mensaje

- En la configuración, crear el formateador de los mensajes:

```
@Bean
public MessageConverter jsonConverter() () {
      return new Jackson2JsonMessageConverter();
}
```

- En la configuración, crear la versión personalizada del RabbitTemplate con el formateador recién creado:

```
@Bean
public RabbitTemplate rabbitTemplate(final ConnectionFactory connectionFactory) {
   RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
   rabbitTemplate.setMessageConverter(jsonConverter());
   return rabbitTemplate;
}
```


Eventos

- Un evento es una notificación que indica un cambio en un entorno, una aplicación o un servicio.
- En una arquitectura basada en eventos, los productores de eventos generan un flujo de eventos que los consumidores de eventos escuchan. Los productores están desconectados entre sí y de los consumidores, no saben si los consumidores están escuchando. A diferencia de los mensajes, un evento puede ser consumido por múltiples consumidores o por ninguno. Dado que un evento puede no ser consumido, la carga útil suele estar reducida al máximo en comparación con los mensajes.
- La transmisión de eventos (event streaming) es la práctica de:
  - capturar datos en tiempo real de las fuentes de eventos (como bases de datos, sensores, dispositivos móviles, IoT, servicios en la nube y aplicaciones de software) en forma de flujos de eventos
  - almacenar estos flujos de eventos de forma duradera para su posterior recuperación
  - manipular, procesar y reaccionar a los flujos de eventos en tiempo real y retrospectivamente
  - enrutar los flujos de eventos a diferentes tecnologías de destino según sea necesario.
- La transmisión de eventos garantiza un flujo continuo y una interpretación de los datos para que la información correcta esté en el lugar correcto, en el momento correcto.
- Los eventos se entregan casi en tiempo real, de modo que los consumidores pueden responder inmediatamente a los eventos cuando se producen. Un consumidor puede unirse en cualquier momento y capturar tantos eventos anteriores como desee.

Apache Kafka

- Apache Kafka es una popular plataforma de streaming de eventos y de código abierto, que sirve para recoger, procesar y almacenar datos de eventos de streaming o datos sin principio ni final concretos. Kafka posibilita aplicaciones distribuidas que pueden escalar para gestionar miles de millones de eventos de streaming cada minuto y casi en tiempo real.
- Al estar diseñado específicamente para la transmisión de registros en tiempo real, Apache Kafka es ideal para las aplicaciones que requieren:
  - Intercambio de datos fiables entre diferentes componentes
  - La capacidad de dividir las cargas de trabajo de mensajería a medida que cambian los requisitos de la aplicación
  - Transmisión en tiempo real para el procesamiento de datos
  - Soporte nativo para la reproducción de datos/mensajes

Conceptos

- Un **evento** registra el hecho de que "algo sucedió" en el mundo o en el negocio. También se le llama registro o mensaje en la documentación. Cuando se leen o escriben datos en Kafka, se hace en forma de eventos. Conceptualmente, un evento tiene una clave, un valor, una marca de tiempo y encabezados de metadatos opcionales.
- Los **productores** son aquellas aplicaciones cliente que publican (escriben) eventos en Kafka, y los **consumidores** son los que se suscriben (leen y procesan) estos eventos. En Kafka, los productores y los consumidores están totalmente desvinculados y son independientes entre sí, lo cual es un elemento de diseño clave para lograr la alta escalabilidad por la que Kafka es conocido.
- Los eventos se organizan y almacenan de forma duradera en temas con nombre (**topics**), similar a las colas de mensajes. Los temas en Kafka son siempre multiproductor y multisuscriptor: un tema puede tener cero, uno o muchos productores que escriben eventos en él, así como cero, uno o muchos consumidores que se suscriben a estos eventos. Los eventos de un tema se pueden leer tantas veces como sea necesario; a diferencia de los sistemas de mensajería tradicionales, los eventos no se eliminan después del consumo. En su lugar, se define a través de la configuración del tema durante cuánto tiempo Kafka debe retener los eventos, después del cual se descartarán los eventos antiguos. El rendimiento de Kafka es efectivamente constante con respecto al tamaño de los datos, por lo que almacenar datos durante mucho tiempo está perfectamente bien.

API de Kafka

- **Admin API**: para administrar e inspeccionar temas, intermediarios y otros objetos de Kafka.
- **Producer API**: para publicar (escribir) un flujo de eventos en uno o más temas de Kafka.
- **Consumer API**: para suscribirse a (leer) uno o más temas y procesar el flujo de eventos producidos para ellos.
- **Kafka Streams API**: para implementar microservicios y aplicaciones de procesamiento de flujo. Proporciona funciones de nivel superior para procesar flujos de eventos, incluidas transformaciones, operaciones con estado como agregaciones y uniones, ventanas, procesamiento basado en el tiempo del evento y más. La entrada se lee de uno o más temas para generar una salida a uno o más temas, transformando efectivamente los flujos de entrada en flujos de salida.
- **Kafka Connect API**: para crear y ejecutar conectores de importación/exportación de datos reutilizables que consumen (leen) o producen (escriben) flujos de eventos desde y hacia sistemas y aplicaciones externos para que puedan integrarse con Kafka, hay cientos de conectores listos para usar.
- Herramientas de línea de comandos para tareas de gestión y administración.

Instalación

- Hay varias opciones disponibles para instalar y configurar el entorno. Se pueden instalar los componentes necesarios en la máquina host de forma tradicional, pero es mas conveniente utilizar contenedores Docker dado que tiene la ventaja de ser limpia, independiente del sistema y fácil de configurar.
- Los componentes imprescindibles para operar con Apache Kafka son:
  - Kafka Brokers: Cada uno de los servicios intermediarios de Kafka que conforman el clúster. Almacenan y distribuyen los datos, que se organizan en topics.
  - Zookeeper: Gestiona los brokers de Kafka y les envía notificaciones en caso de cambio como creación de topics, borrado de topics, caída de broker, recuperación de broker …
  - UI for Apache Kafka (opcional): Es una interfaz web de usuario gratuita y de código abierto para monitorizar y administrar clústeres de Apache Kafka.

Docker Compose

```
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
    ports:
      - 2181:2181

  kafka:
    image: confluentinc/cp-kafka:latest
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - 9092:9092
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS:
PLAINTEXT://kafka:29092,PLAINTEXT_HOST://localhost:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1

  kafka-ui:
    image: provectuslabs/kafka-ui
    container_name: kafka-ui
    depends_on:
      - kafka
    ports:
      - 9091:8080
    environment:
      - KAFKA_CLUSTERS_0_NAME=local
      - KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS=kafka:29092
      - KAFKA_CLUSTERS_0_ZOOKEEPER=localhost:2181
```


Apache Kafka: Spring Boot

- Añadir al proyecto:
  - Messaging > Spring for Apache Kafka
- Configurar:

```
spring.kafka.bootstrap-servers=${KAFKA_BROKER:http://localhost:9092}
```

- KafkaTemplate<String, String> se configuran de forma automáticamente.

```
@Autowired
private KafkaTemplate<String, String> kafkaTemplate;
```

- Para crear un tema si no existe:

```
@Bean
public NewTopic topicLocation() {
    return new NewTopic("mi-tema", 1, (short) 1);
}
```

Enviar un evento

```
// la clave (key) es opcional
public void sendMessage(String topic, String key, String message) {
        kafkaTemplate.send(topic, key, message)
            .thenAccept(result -> LOG.info(String.format("TOPIC: %s, KEY: %s,
                MESSAGE: %s, OFFSET: %s", topic, key, message,
                result.getRecordMetadata().offset())))
            .exceptionally(ex -> {
                LOG.error(String.format("TOPIC: %s, KEY: %s, MESSAGE: %s, ERROR:
                    %s", topic, key, message, ex.getMessage()));
                return null;
            });
}
```


Recibir eventos

- Recibir los mensajes de los eventos:

```
@KafkaListener(topics = "${topic.name}", groupId = "nombre-consumidor")
public void listenTopic(String message) {
    LOG.info("Mensaje: " + message);
}
```

- Extraer los metadatos de los eventos:

```
@KafkaListener(topics = "${topic.name}", topicPattern = "${topic.name}", groupId = "nombre-
    consumidor")
public void listenWithHeaders(@Header(KafkaHeaders.RECEIVED_KEY) String key, @Payload String
    message, @Header(KafkaHeaders.OFFSET) String offset) {
    LOG.info(String.format("KEY: %s, MESSAGE: %s, OFFSET: %s", key, message, offset));
}
```

- Para recibir todos los eventos anteriores del tema:
  - spring.kafka.consumer.auto-offset-reset=earliest

ActiveMQ (JMS)

- Apache ActiveMQ es un intermediario de mensajes basado en Java, multiprotocolo y de código abierto, que implementa plenamente la especificación de Java Message Service 1.1 (JMS).
- Es compatible con los protocolos estándar de la industria (AMQP, STOMP sobre websockets, MQTT y JMS) para que los usuarios obtengan los beneficios de las opciones del cliente en una amplia gama de lenguajes y plataformas (JavaScript, C, C++, Python, .Net y más).
- Actualmente hay dos "sabores" de ActiveMQ disponibles: el conocido como broker "clásico" y el broker de "próxima generación" cuyo nombre en código es Artemis. Una vez que Artemis alcance un nivel suficiente de paridad de funciones con el broker "clásico", se convertirá en la versión principal de ActiveMQ.
- El broker "Classic" ofrece características empresariales tales como JMS 1.1 con implementación de cliente completa que incluye JNDI, alta disponibilidad mediante almacenamiento compartido, modelo familiar de direccionamiento basado en JMS, "Red de intermediarios" para la distribución de carga y KahaDB o JDBC para persistencia.
- Artemis ofrecerá JMS 1.1 y 2.0 + Jakarta Messaging 2.0 y 3.0 con implementaciones de cliente completas (incluido JNDI), alta disponibilidad mediante almacenamiento compartido o replicación de red, modelo simple y potente de direccionamiento agnóstico de protocolo, agrupación flexible para distribuir la carga, implementaciones avanzadas del diario para persistencia de baja latencia (así como JDBC), duplicación asíncrona para recuperación ante desastres y equilibrio de carga basado en datos.
- ActiveMQ "Artemis" dispondrá de una alta paridad de características con ActiveMQ "Classic" para facilitar la migración.

Instalación

- Descargar la última versión estable para Windows desde:
  - https://activemq.apache.org/components/classic/download/
- Extraer los archivos del archivo ZIP en el directorio de instalación.
- Para arrancar el servicio, desde una ventana de consola, cambiar al directorio de instalación y ejecutar:
  - bin\activemq start
- Los directorios de trabajo se crean en relación con el directorio actual. Para crear directorios de trabajo en el lugar adecuado, ActiveMQ debe iniciarse desde su directorio de inicio/instalación.
- Para acceder al complemento de administración, desde el navegador:
  - URL: http://127.0.0.1:8161/admin/
    - Iniciar sesión: admin
    - Contraseña: admin
  - Ir a "Queues"
  - Agregar un nombre de cola y hacer clic en crear
  - Enviar mensaje de prueba haciendo clic en "Enviar a"
  - Consultar con atom o rss

Conceptos

- Soporta dos modelos/dominios de mensajería:
  - Punto a Punto (Queues), en la que un mensaje se consume por un único consumidor.
  - Publicación/Subscripción (Topics), en la que un mensaje se consume por muchos consumidores.
- Soporta el modelo de mensaje unificado de JMS que especifica que todos los mensajes deben contar con cabeceras, propiedades, y cuerpo. Las cabeceras ofrecen metadatos sobre el mensaje utilizado por ambos clientes y el proveedor JMS. Las propiedades son campos opcionales dentro del mensaje para añadir información adicional al mensaje. El cuerpo puede contener tanto texto como datos binarios mediante los diferentes tipos de mensajes.
- JMS define seis tipos de cuerpo para los mensajes, también conocidos tipos de mensajes:
  - Message: Tipo de mensaje base. Se utiliza para enviar un mensaje sin cuerpo, solo cabeceras y propiedades. Normalmente se utiliza para notificación simple de eventos.
  - MapMessage: Compuesto de un conjunto de pares {nombre,valor}. El tipo de los nombres es String, y los valores tipos primitivos Java. A los nombres (que no están ordenados) podemos acceder de forma secuencial mediante un enumerador, o por acceso directo por nombre.
  - BytesMessage: Contiene un array de bytes sin interpretar. Se utiliza para hacer coincidir el cuerpo con un formato de mensaje existente (legacy).
  - StreamMessage: El cuerpo es un flujo de tipos primitivos Java, cuya lectura y escritura se realiza de modo secuencial.
  - TextMessage: Un mensaje cuya carga es un String. Se suele utilizar para enviar texto simple y datos XML o JSON.
  - ObjectMessage: La carga es un objeto Java Serializable. Normalmente se utiliza para trabajar con objetos Java complejos.

Protocolos

- Apache ActiveMQ es un intermediario de mensajes que admite múltiples protocolos de nivel de conexión para una máxima interoperabilidad.
  - Core-UI (port: 8161)
  - REST (port: 8161)
  - RSS y Atom (port: 8161)
  - AMQP (port: 5672)
  - MQTT (port: 1883)
  - WebSocket (port: 61614)
  - Stomp (port: 61613)
  - JMS (port: 61616)
  - OpenWire (port: 61616)
  - ~~XMPP (port: 61222)~~

ActiveMQ: Spring Boot

- Para añadir al proyecto ActiveMQ 5:
  - Messaging > Spring for Apache ActiveMQ 5
- Configurar:

```
#spring.activemq.broker-url=tcp://localhost:61616
#spring.activemq.user=admin
#spring.activemq.password=admin
spring.activemq.in-memory=false
```

- Para añadir al proyecto ActiveMQ Artemis:
  - Messaging > Spring for Apache ActiveMQ Artemis
- Configurar:

```
#spring.artemis.mode=native
spring.artemis.host=localhost
spring.artemis.port=61616
#spring.artemis.user=admin
#spring.artemis.password=admin
```


Template y Anotaciones

- Spring proporciona un marco de integración de JMS que simplifica el uso del API de JMS de la misma manera que lo hace para JDBC.
- JMS se puede dividir aproximadamente en dos áreas de funcionalidad, a saber, la producción y el consumo de mensajes.
- La clase JmsTemplate se utiliza para la producción de mensajes y la recepción de mensajes sincrónicos.
- Para una recepción asincrónica similar al estilo de bean controlado por mensajes de Jakarta EE, Spring proporciona una serie de contenedores de escucha de mensajes que puede usar para crear POJO controlados por mensajes (MDP). Spring también proporciona una forma declarativa con anotaciones de crear oyentes de mensajes.

Queues: enviar y recibir

- Enviar un mensaje:

```
@Autowired
JmsTemplate jms;

@GetMapping(path = "/saludo/{nombre}")
public String saluda(@PathVariable String nombre) {
        String msg = "Hola " + nombre;
        jms.convertAndSend("saludos", new MessageDTO(msg, origen));
        return "SEND COLA: " + msg;
}
```

- Recibir mensajes:

```
@JmsListener(destination = "saludos")
public void listenQueue(MessageDTO in) {
        Store.addQueue(new Message(in));
        log.warn("MENSAJE RECIBIDO: " + in);
}
```


Topics: enviar y recibir

- Se puede establecer la Publicación/Subscripción como mecanismo por defecto en application.properties:

```
spring.jms.pub-sub-domain=true
```

- En caso de convivir los dos sistemas hay que crear uno o varios contenedores personalizados para asignarlos a los listener:

```
@Bean
JmsListenerContainerFactory<?> myPubSubFactory(ConnectionFactory connectionFactory,
        DefaultJmsListenerContainerFactoryConfigurer configurer) {
    DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
    configurer.configure(factory, connectionFactory);
    factory.setPubSubDomain(true);
    return factory;
}
```

Topics: enviar y recibir

- Enviar un mensaje:

```
@Autowired
JmsTemplate jms;
@GetMapping(path = "/despedida/{nombre}")
public String despedida(@PathVariable String nombre) {
        String msg = "Adios " + nombre;
        jms.setPubSubDomain(true);  // Si por defecto spring.jms.pub-sub-domain=false
        jms.convertAndSend("despedidas", new MessageDTO(msg, origen));
        return "SEND TEMA: " + msg;
}
```

- Recibir mensajes:

```
@JmsListener(destination = "despedidas", containerFactory = "myPubSubFactory")
public void listen(MessageDTO in) {
        Store.addTopic(new Message(in));
        log.warn("TEMA RECIBIDO: " + in);
}
```


Formato del Mensaje

- Los objetos ObjectMessage dependen de la serialización de Java de la carga útil del objeto marshal/unmarshal. Este proceso generalmente se considera inseguro ya que una carga útil maliciosa puede explotar el sistema host. Es por eso que a partir de las versiones 5.12.2 y 5.13.0 , ActiveMQ obliga a incluir explícitamente en la lista blanca de los paquetes que se pueden intercambiar.

  -Dorg.apache.activemq.SERIALIZABLE_PACKAGES=*

- Para facilitar el envío de objetos del modelo de dominio, JmsTemplate tiene varios métodos de envío que toman un objeto Java como argumento para el contenido de datos de un mensaje. Los métodos sobrecargados convertAndSend() y receiveAndConvert() delegan el proceso de conversión a una instancia de la interfaz MessageConverter. Esta interfaz define un contrato simple para convertir entre objetos Java y mensajes JMS. La implementación predeterminada (SimpleMessageConverter) admite la conversión entre String a TextMessage, byte[] a BytesMessage y java.util.Map a MapMessage.

Formato del Mensaje

- Los objetos ObjectMessage dependen de la serialización de Java de la carga útil del objeto marshal/unmarshal. Este proceso generalmente se considera inseguro ya que una carga útil maliciosa puede explotar el sistema host. Es por eso que a partir de las versiones 5.12.2 y 5.13.0 , ActiveMQ obliga a incluir explícitamente en la lista blanca de los paquetes que se pueden intercambiar.

  -Dorg.apache.activemq.SERIALIZABLE_PACKAGES=*
- Para facilitar el envío de objetos del modelo de dominio, JmsTemplate tiene varios métodos de envío que toman un objeto Java como argumento para el contenido de datos de un mensaje. Los métodos sobrecargados convertAndSend() y receiveAndConvert() delegan el proceso de conversión a una instancia de la interfaz MessageConverter. Esta interfaz define un contrato simple para convertir entre objetos Java y mensajes JMS. La implementación predeterminada (SimpleMessageConverter) admite la conversión entre String a TextMessage, byte[] a BytesMessage y java.util.Map a MapMessage.

Formato del Mensaje

- Para establecer un conversor predeterminado:

```
@Bean // Serialize message content to json using TextMessage
MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_type");
        return converter;
}
```

- Para acomodar la configuración de las propiedades, los encabezados y el cuerpo de un mensaje que no se pueden encapsular genéricamente dentro del convertidor, la interfaz MessagePostProcessor brinda acceso al mensaje después de que se haya convertido pero antes de que se envíe.

```
public void sendWithConversion(Map map) {
   jmsTemplate.convertAndSend("testQueue", map, new MessagePostProcessor() {
      public Message postProcessMessage(Message message) throws JMSException {
         message.setIntProperty("AccountID", 1234);
         message.setJMSCorrelationID("123-00001");
         return message;
      }
   });
}
```

SEGURIDAD

Patrón: Externalized configuration

- Motivación:
  - Se has aplicado la arquitectura de microservicios. Los microservicios utilizan una infraestructura (servidores de bases de datos, registro de servicios, intermediarios de mensajes, …) y servicios de terceros (procesamiento de pagos, correo electrónico, open authorization, …) que requieren configuración.
- Intención:
  - ¿Cómo permitir que un microservicio comparta su configuración entre sus diferentes instancias o con otros microservicios y se ejecute en múltiples entornos sin modificaciones?
- Requisitos:
  - Un microservicio debe contar con datos de configuración que le indiquen cómo conectarse a la infraestructura y los servicios de terceros.
  - Un microservicio debe ejecutarse en múltiples entornos (desarrollo, prueba, producción, …) sin modificaciones ni re compilación.
  - Los diferentes entornos tienen diferentes instancias de infraestructura o servicios de terceros con sus correspondientes configuraciones.
  - Los microservicios se despliegan en múltiples ubicaciones.

Patrón: Externalized configuration

- Solución:
  - Externalizar toda la configuración de la aplicación, incluidas las credenciales y ubicaciones de red. Al iniciarse, un microservicio lee la configuración de una fuente externa, por ejemplo, variables de entorno del sistema operativo, …
- Implementación:
  - Crear un servidor de configuración con Spring Cloud Config
- Consecuencias:
  - Este patrón tiene los siguientes beneficios:
    - Los servicios se ejecuta en múltiples entornos sin modificación y/o recompilación..
  - Este patrón tiene los siguientes problemas:
    - ¿Cómo garantizar que cuando se implementa un servicio la configuración proporcionada coincida con lo esperado?
    - ¿Cómo garantizar la confidencialidad de los secretos?
- Patrones relacionados:
  - Los patrones de registro/descubrimiento de servicios resuelven el problema relacionado de cómo un servicio conoce la ubicación de red de otros servicios.

Spring Cloud Config

- Para conectarse con recursos protegidos y otros servicios, las aplicaciones típicamente necesitan usar cadenas de conexión, contraseñas u otras credenciales que contengan información confidencial.
- Estas partes de información sensible se llaman secretos.
- Es una buena práctica no incluir secretos en el código fuente y, sobre todo, no almacenar secretos en el sistema de control de versiones.
- En su lugar, debería utilizar el modelo de configuración para leer los secretos desde ubicaciones más seguras.
- Se deben separar los secretos para acceder a los recursos de desarrollo y pre-producción (staging) de los que se usan para acceder a los recursos de producción, porque diferentes individuos necesitarán acceder a esos conjuntos diferentes de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en una Key Vault.
- Los servidores de configuración de Spring Cloud soportan los siguientes orígenes (backends): GIT, Vault y JDBC
- Los recursos con los nombres de archivos application*(application.properties, application.yml, application-*.properties, etc.) son compartidos entre todas las aplicaciones cliente.

Spring Cloud Config: Servidor

- Añadir proyecto:
  - Spring Cloud Config > Config Server
- Anotar la aplicación:
  @EnableConfigServer
- Crear repositorio (local):
  - Crear directorio
  - Desde la consola de comandos posicionada en el directorio: git init
  - Crear un fichero que se llame como el spring.application.name del cliente que va a solicitar los datos y extensión **.properties**, con la configuración. Se pueden incluir perfiles añadiéndoselos al nombre: -production.properties
  - Añadir el fichero al repositorio: git add mi-service.properties ó git add .
  - Realizar un commit del fichero: git commit -m "Comentario a la versión"
- Configurar:

```
server.port= ${PORT:8888}
spring.cloud.config.server.git.uri=file://C:/mi/configuration-repository
#spring.cloud.config.server.git.uri=https://github.com/jmagit/mi-config.git
```


Spring Cloud Config: Cliente

- Añadir proyecto:
  - Spring Cloud Config > Config Client
- Para poder refrescar la configuración en caliente, se añadirá el starter Actuator
- Configurar:

```
server.port= ${PORT:8001}
spring.application.name=mi-service
spring.config.import=optional:configserver:${CONFIG_URI:http://localhost:8888}
#spring.profiles.active=production
management.endpoints.web.exposure.include=refresh
```

- Eclipse: Run Configurations → Arguments → Program Arguments: --spring.profiles.active=production
- De forma predeterminada, los valores de configuración solo se leen en el inicio del cliente. Puede forzar a un bean a que actualice su configuración (vuelva a leer), para ello debe anotarse con @RefreshScope (clase o método).
- Para refrescar la configuración en caliente después de realizar un commit al repositorio hay que hacer un POST a:
  - http://localhost:8001/actuator/refresh

Spring Cloud Config: Cliente

- Para recuperar un valor de la configuración:

```
@Value("${mi.valor}")
String miValor;
```

- Para recuperar y crear un componente:

```
// En el fichero .properties
// rango.min=1
// rango.max=10
@Data
@Component
@ConfigurationProperties("rango")
public class Rango {
   private int min;
   private int max;
}
@Autowired
private Rango rango;
```


Autenticación y Autorización

- La autenticación es un proceso en el que un usuario o una aplicación se identifica proporcionando credenciales que después se comparan con las almacenadas en un sistema operativo, base de datos, aplicación o recurso para validar que es realmente quién asegura ser. La autenticación puede crear una o varias identidades para el usuario o aplicación autenticado.
- La autorización se refiere al proceso que determina lo que una identidad puede hacer en función a los permisos otorgados a una identidad concreta sobre un recurso concreto. La autorización es ortogonal e independiente de la autenticación. Sin embargo, la autorización requiere un mecanismo de autenticación. La autorización puede utilizar un modelo basado en roles, sencillo y declarativo, o un modelo avanzado basado en directivas y evidencias.

Spring Security

- Spring Security es un framework de apoyo al marco de trabajo Spring, que dota al mismo de una serie servicios de seguridad aplicables para sistemas basados en la arquitectura JEE, enfocado particularmente sobre proyectos construidos usando Spring Framework. De esta dependencia, se minimiza la curva de aprendizaje si ya se conoce Spring.
- Los procesos de seguridad están destinados principalmente, a comprobar la identidad del usuario mediante la autenticación y los permisos asociados al mismo mediante la autorización. La autorización, basada en roles, es dependiente de la autenticación ya que se produce posteriormente a su proceso.
- Por regla general muchos de estos modelos de autenticación son proporcionados por terceros o son desarrollados por estándares importantes como el IETF. Adicionalmente, Spring Security proporciona su propio conjunto de características de autenticación:
  - In-Memory, JDBC, LDAP, OAuth 2.0, Kerberos, SAML …
- El proceso de autorización se puede establecer a nivel de recurso individual o mediante configuración que cubra múltiples recursos.

Spring Boot

- Si Spring Security está en la ruta de clase, las aplicaciones web están protegidas de forma predeterminada. Spring Boot se basa en la estrategia de negociación de contenido de Spring Security para determinar si se debe usar httpBasic o formLogin.
- Para agregar seguridad a nivel de método a una aplicación web, también puede agregar @EnableMethodSecurity en la configuración que desee.
- El valor predeterminado del UserDetailsService tiene un solo usuario. El nombre del usuario es “user” y la contraseña se genera aleatoriamente al arrancar y se imprime como INFO:
  - Using generated security password: e4918bc4-d8ac-4179-9916-c37825c7eb55
- Puede cambiar el nombre de usuario y la contraseña proporcionando un spring.security.user.name y spring.security.user.password en application.properties.
- Las características básicas predeterminadas en una aplicación web son:
  - Un bean UserDetailsService con almacenamiento en memoria y un solo usuario con una contraseña generada.
  - Inicio de sesión basado en formularios o seguridad básica HTTP (según el tipo de contenido) para toda la aplicación (incluidos los endpoints).
  - Un DefaultAuthenticationEventPublisher para la publicación de eventos de autenticación.

Seguridad MVC

- La configuración de seguridad predeterminada se implementa en SecurityAutoConfiguration y UserDetailsServiceAutoConfiguration. SecurityAutoConfiguration importa SpringBootWebSecurityConfiguration para la seguridad web y UserDetailsServiceAutoConfiguration configura la autenticación, que también es relevante en aplicaciones no web.
- Para desactivar completamente la configuración de seguridad de la aplicación web predeterminada,se puede agregar un bean de tipo WebSecurityConfigurerAdapter (al hacerlo, no se desactiva la configuración UserDetailsService).
- Para cambiar la configuración del UserDetailsService, se puede añadir un bean de tipo UserDetailsService, AuthenticationProvider o AuthenticationManager.
- Las reglas de acceso se pueden anular agregando una personalización de WebSecurityConfigurerAdapter, que proporciona métodos de conveniencia que se pueden usar para anular las reglas de acceso para los puntos finales del actuador y los recursos estáticos.
- EndpointRequest se puede utilizar para crear un RequestMatcher que se basa en la propiedad management.endpoints.web.base-path. PathRequest se puede usar para crear recursos RequestMatcher en ubicaciones de uso común

Elementos principales

- SecurityContextHolder contiene información sobre el contexto de seguridad actual de la aplicación, que contiene información detallada acerca del usuario que está trabajando actualmente con la aplicación. Utiliza el ThreadLocal para almacenar esta información, por lo que el contexto de seguridad siempre está disponible para la ejecución de los métodos en el mismo hilo de ejecución (Thread). Para cambiar eso, se puede utilizar un método estático SecurityContextHolder.setStrategyName (estrategia de cadena).
- SecurityContext contiene un objeto de autenticación, es decir, la información de seguridad asociada con la sesión del usuario.
- Authentication es, desde punto de vista Spring Security, un usuario (Principal)
- GrantedAuthority representa la autorización dada al usuario de la aplicación.
- UserDetails estandariza la información del usuario independientemente del sistema de autenticación.
- UserDetailsService es la interfaz utilizada para crear el objeto UserDetails.

Proceso de Autenticación

- Para poder tomar decisiones sobre el acceso a los recursos, es necesario que el participante se identifique para realizar las comprobaciones necesarias sobre su identidad. Mediante la interfaz Authentication, se pueden acceder a tres objetos bien diferenciados:
  - principal, normalmente hace referencia al nombre del participante
  - credenciales (del usuario) que permiten comprobar su identidad, normalmente su contraseña, aunque también puede ser otro tipo de métodos como certificados, etc...
  - autorizaciones, un lista de los roles asociados al participante.
- Si un usuario inicia un proceso de autenticación, se crea un objeto Authentication, con los elementos Principal y Credenciales. Si realiza la autenticación mediante el empleo de contraseña y nombre usuario, se crea un objeto UsernamePasswordAuthenticationToken. El framework Spring Security aporta un conjunto de clases que permite que esta autenticación se realice mediante nombre de usuario y contraseña. Para ello, utiliza la autenticación que proporciona el contenedor o utiliza un servicio de identificación basado en Single Sign On (sólo se identifica una vez).

Proceso de Autenticación

- Una vez se ha obtenido el objeto Authentication se envía al AuthenticationManager. Una vez aquí, se realiza una comprobación del contenido de los elementos del objeto principal y las credenciales. Se comprueban que concuerdan con las esperadas, añadiéndole al objeto Authentication las autorizaciones asociadas a esa identidad o generandon una excepción de tipo AuthenticationException.
- El propio framework ya tiene implementado un gestor de autenticación que es válido para la mayoría de los casos, el ProviderManager. El bean AuthenticationManager es del tipo ProviderManager, lo que significa que actúa de proxy con el AuthenticationProvider.
- Este es el encargado de realizar la comprobación de la validez del nombre de usuario/contraseña asociada y de devolver las autorizaciones permitidas a dicho participante (roles asociados).
- Esta clase delega la autenticación en una lista que engloba a los proveedores y que, por tanto, es configurable. Cada uno de los proveedores tiene que implementar el interfaz AuthenticationProvider.

Proceso de Autenticación

- Cada aplicación web tendrá una estrategia de autenticación por defecto. Cada sistema de autenticación tendrá su AuthenticationEntryPoint propio, que realiza acciones como enviar avisos para la autenticación.
- Cuando el navegador decide presentar sus credenciales de autenticación (ya sea como formulario HTTP o HTTP header) tiene que existir algo en el servidor que "recoja" estos datos de autenticación. A este proceso se le denomina "mecanismo de autenticación". Una vez que los detalles de autenticación se recogen en el agente de usuario, un objeto "solicitud de autenticación" se construye y se presenta a un AuthenticationProvider.
- El último paso en el proceso de autenticación de seguridad es un AuthenticationProvider. Es el responsable de tomar un objeto de solicitud de autenticación y decidir si es o no válida. El Provider decide si devolver un objeto de autenticación totalmente lleno o una excepción.
- Cuando el mecanismo de autenticación recibe de nuevo el objeto de autenticación, si se considera la petición válida, debe poner la autenticación en el SecurityContextHolder, y hacer que la solicitud original se ejecute. Si, por el contrario, el AuthenticationProvider rechazó la solicitud, el mecanismo de autenticación mostrara un mensaje de error.

Proceso de Autenticación

- El DaoAuthenticationProvider es una implementación de la interfaz de autenticación centrada en el acceso a los datos que se encuentran almacenados dentro de una base de datos. Este proveedor específico requiere una atención especial.
- Esta implementación delega a su vez en un objeto de tipo UserDetailsService, un interfaz que define un objeto de acceso a datos con un único método loadUserByUsername que permite obtener la información de un usuario a partir de su nombre de usuario devolviendo un UserDetails que estandariza la información del usuario independientemente del sistema de autenticación.
- El UserDetails contiene el nombre de usuario, contraseña, los flags isAccountNonExpired, isAccountNonLocked, isCredentialsNonExpired, IsEnabled y los roles del usuario.
- Los roles de usuario son cadenas que por defecto llevan el prefijo de “ROLE_”.

Cifrado de claves

- Nunca se debe almacenar las contraseñas en texto plano, uno de los procesos básicos de seguridad contra robo de identidad es el cifrado de las claves de usuario.
- Spring Security ofrece algoritmos de encriptación que se pueden aplicar de forma rápida al resto de la aplicación.
- Para esto hay que utilizar una clase que implemente la interfaz PasswordEncoder, que se utilizará para cifrar la contraseña introducida a la hora de crear el usuario.
- Además, hay que pasárselo al AuthenticationManagerBuilder cuando se configura para que cifre la contraseña recibida antes de compararla con la almacenada.
- Spring suministra BCryptPasswordEncoder que es una implementación del algoritmo BCrypt, que genera una hash segura como una cadena de 60 caracteres.

```
@Autowired private PasswordEncoder passwordEncoder;

String encodedPass = passwordEncoder.encode(userDTO.getPassword());
```


Configuración de Autenticación

- Para realizar la configuración se crea una clase, anotada con @Configuration y @EnableWebSecurity, que extienda a WebSecurityConfigurerAdapter. La sobreescritua del método configure(AuthenticationManagerBuilder) permite fijar el UserDetailsService y el PasswordEncoder.

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   @Autowired
   UserDetailsService userDetailsService;
   @Bean
   public PasswordEncoder passwordEncoder() {  return new BCryptPasswordEncoder();  }
   @Autowired
   public void configure(AuthenticationManagerBuilder auth) throws Exception {
      auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
   }
}
```

UserDetailsService

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
	@Autowired
	private PasswordEncoder passwordEncoder;
	@Override
	public UserDetails loadUserByUsername(final String username) throws UsernameNotFoundException {
		switch(username) {
		case "user":		return this.userBuilder(username, passwordEncoder.encode("user"), "USER");
		case "manager":		return this.userBuilder(username, passwordEncoder.encode("manager"), "MANAGER");
		case "admin":		return this.userBuilder(username, passwordEncoder.encode("admin"), "USER", "MANAGER", "ADMIN");
		default: throw new UsernameNotFoundException("Usuario no encontrado");
		}
	}
	private User userBuilder(String username, String password, String... roles) {
		List<GrantedAuthority> authorities = new ArrayList<>();
		for (String role : roles) {
			authorities.add(new SimpleGrantedAuthority("ROLE_" + role));
		}
		return new User(username, password, /* enabled */ true, /* accountNonExpired */ true,
				/* credentialsNonExpired */ true, /* accountNonLocked */ true, authorities);
	}}
```


InMemoryAuthentication

```
@Autowired
  public void configureAuth(AuthenticationManagerBuilder auth) throws Exception {
   auth.inMemoryAuthentication()
       .withUser("user")
           .password("user").roles("USER")
       .and()
       .withUser("manager")
           .password("manager").roles("MANAGER")
       .and()
       .withUser("admin")
           .password("admin").roles("USER", "ADMIN");
  }
```

Autorización

- El AccessDecisionManager es la interfaz que atiende la llamada AbstractSecurityInterceptor producida tras interceptar una petición. Esta interfaz es la responsable final de la toma de decisiones sobre el control de acceso.
- AccessDecisionManager delega la facultad de emitir votos en objetos de tipo AccessDecisionVoter. Se proporcionan dos implementaciones de éste último interfaz:
  - RoleVoter, que comprueba que el usuario presente un determinado rol, comprobando si se encuentra entre sus autorizaciones (authorities).
  - BasicAclEntryVoter, que a su vez delega en una jerarquía de objetos que permite comprobar si el usuario supera las reglas establecidas como listas de control de acceso.
- El acceso por roles se puede fijar para:
  - URLs, permitiendo o denegando completamente
  - Servicios, controladores o métodos individuales

Configuración

- La sobreescritua del método configure(HttpSecurity) permite configurar el http.authorizeRequests():

  .requestMatchers("/static/**").permitAll() acceso a los recursos

  .anyRequest().authenticated() se requiere estar autenticado para todas las peticiones.

  .requestMatchers("/**").permitAll() equivale a anyRequest()

  .requestMatchers("/privado/**", "/config/**").authenticated() equivale a @PreAuthorize("authenticated")

  .requestMatchers("/admin/**").hasRole("ADMIN") equivale a @PreAuthorize("hasRole('ROLE_ADMIN')")
- El método .and() permite concatenar varias definiciones.

Seguridad: Configuración

```
@Configuration
@EnableWebSecurity
@EnableMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
   // …
   @Override
   protected void configure(HttpSecurity http) throws Exception {
      http.csrf().disable()
         .authorizeRequests()
            .requestMatchers("/**").permitAll()
            .requestMatchers("/privado/**").authenticated()
            .requestMatchers("/admin/**").hasRole("ADMIN")
         .and().formLogin().loginPage("/login").permitAll()
         .and().logout().permitAll();
   }
}
```


Basada en anotaciones

- Desde la versión 2.0 en adelante, Spring Security ha mejorado sustancialmente el soporte para agregar seguridad a los métodos de capa de servicio proporcionando soporte para la seguridad con anotación JSR-250, así como la anotación original @Secured del marco. A partir de la 3.0 también se puede hacer uso de nuevas anotaciones basadas en expresiones: @PreAuthorize y @PostAuthorize.
- Se puede habilitar la seguridad basada anotaciones en cualquier instancia @Configuration utilizando la anotación @EnableMethodSecurity:
  - prePostEnabled habilita las anotaciones previas y posteriores
  - secureEnabled determina si las anotaciones @Secured deben estar habilitadas
  - jsr250Enabled permite usar las anotaciones @RoleAllowed
- **@Secured**: Anotación para definir una lista de atributos de configuración de seguridad para métodos de un servicio y se puede utilizar como una alternativa a la configuración XML.

```
@Secured({ "ROLE_USER" }) public void create(Contact contact) {
@Secured({ "ROLE_USER", "ROLE_ADMIN" }) public void update(Contact contact) {
@Secured({ "ROLE_ADMIN" }) public void delete(Contact contact){
```

Basada en anotaciones

- **@PreAuthorize**: Anotación para especificar una expresión de control de acceso al método que se evaluará para decidir si se permite o no una invocación del método.

```
@PreAuthorize("isAnonymous()")
@PreAuthorize("isAuthenticated()")
@PreAuthorize("hasAuthority('ROLE_TELLER')")
@PreAuthorize("hasRole('USER') or hasRole('ROLE_EDITOR')")
@PreAuthorize("hasPermission(#contact, 'admin')")
```

- Se puede usar un argumento del método como parte de la expresión:

```
@PreAuthorize("#item.username == authentication.name")
public void putProfile(@RequestBody User item) { … }
```

- **@PostAuthorize**: Anotación para especificar una expresión de control de acceso al método que se evaluará después de que se haya invocado un método.

```
@PostAuthorize("returnObject.username == authentication.principal.nickName")
public CustomUser loadUserDetail(String username) {
   return userRoleRepository.loadUserByUserName(username);
}
```

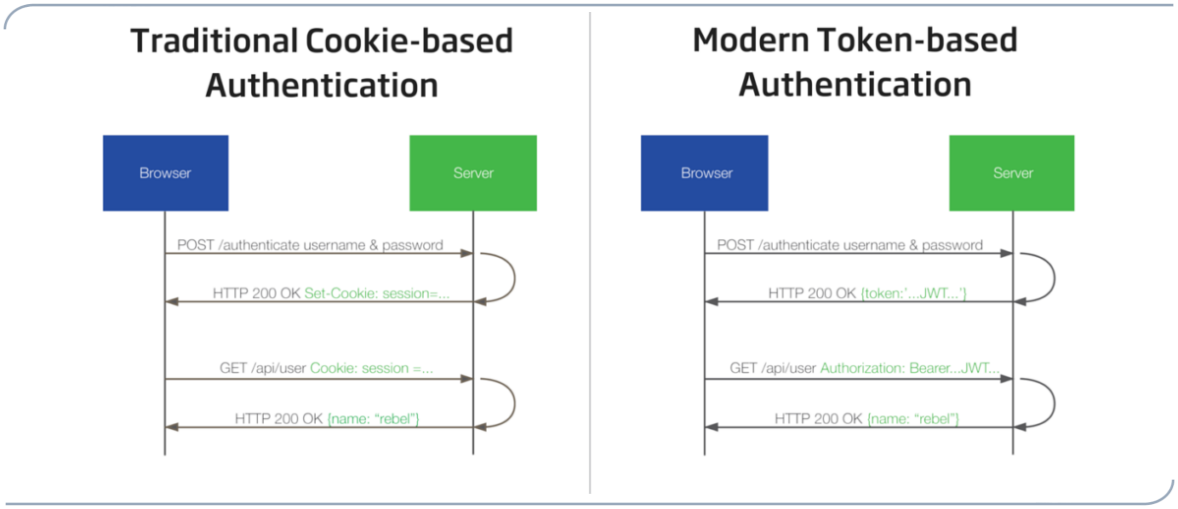

Control de acceso basado en expresiones

Expresión	Descripción
hasRole([role])	Devuelve true si el principal actual tiene el rol especificado. De forma predeterminada, si el rol proporcionado no comienza con 'ROLE_' se agregará. Esto se puede personalizar modificando el defaultRolePrefix en DefaultWebSecurityExpressionHandler.
hasAnyRole([role1,role2])	Se devuelve true si el principal actual tiene alguno de los roles proporcionados (lista de cadenas separadas por comas).
hasAuthority([authority])	Devuelve true si el principal actual tiene la autoridad especificada.
hasAnyAuthority([authority1,authority2])	Se devuelve true si el principal actual tiene alguna de las autorizaciones proporcionadas (se proporciona como una lista de cadenas separadas por comas)
principal	Permite el acceso directo al objeto principal que representa al usuario actual.
authentication	Permite el acceso directo al objeto Authentication actual obtenido del SecurityContext

Control de acceso basado en expresiones

Expresión	Descripción
permitAll	Siempre se evalúa a true
denyAll	Siempre se evalúa a false
isAnonymous()	Devuelve true si el principal actual es un usuario anónimo
isRememberMe()	Devuelve true si el principal actual es un usuario de recordarme
isAuthenticated()	Devuelve true si el usuario no es anónimo
isFullyAuthenticated()	Se devuelve true si el usuario no es un usuario anónimo o recordado
hasPermission(Object target, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(domainObject, 'read')
hasPermission(Object targetId, String targetType, Object permission)	Devuelve true si el usuario tiene acceso al objetivo proporcionado para el permiso dado. Por ejemplo, hasPermission(1, 'com.example.domain.Message', 'read')

Autenticación



Patrones de diseño

- Patrón: Token de acceso
  - Ha aplicado la arquitectura de microservicios y los patrones de API Gateway . La aplicación consta de numerosos servicios. La puerta de enlace API es el único punto de entrada para las solicitudes de los clientes. Autentica las solicitudes y las reenvía a otros servicios, que a su vez podrían invocar otros servicios. ¿Cómo comunicar la identidad del solicitante a los servicios que tramitan la solicitud? El API Gateway autentica la solicitud y pasa un token de acceso (por ejemplo, JSON Web Token) que identifica de forma segura al solicitante en cada solicitud a los servicios. Un servicio puede incluir el token de acceso en las solicitudes que realiza a otros servicios.
- Patrón: Valet Key
  - Usa un token que proporciona a los clientes acceso directo restringido a un recurso específico, con el fin de descargar la transferencia de datos desde la aplicación. Esto es especialmente útil en aplicaciones que usan sistemas o colas de almacenamiento hospedado en la nube, ya que puede minimizar los costes y maximizar la escalabilidad y el rendimiento.
- Patrón: Federated Identity
  - La autenticación se delega a un proveedor de identidad externo. Esto puede simplificar el desarrollo, minimizar los requisitos de administración de usuarios y mejorar la experiencia del usuario de la aplicación.

Autenticación sin estado basada en tokens

- Para solucionar los problemas de sobrecarga y escalabilidad provocados la autenticación basada en sesiones y cookies, surge la autenticación sin estado (stateless). Esto significa que el servidor no va a almacenar ninguna información, ni tampoco la sesión.
- Cuando el usuario se autentica con sus credenciales o cualquier otro método, en la respuesta recibe un token (access token) y, opcionalmente, un refresh token. El token es una cadena encriptada firmada, para evitar alteraciones y ser confiable, con una fecha de expiración corta para evitar vulnerabilidades de seguridad.
- A partir de ese momento, todas las peticiones que se hagan al API llevarán este token en una cabecera HTTP de modo que el servidor pueda identificar qué usuario hace la petición y, una vez verificado el token, confiar en las credenciales suministradas sin necesidad de buscar en base de datos ni en ningún otro sistema de almacenamiento o agente externo.
- Con este enfoque, la aplicación pasa a ser escalable, ya que es el propio cliente el que almacena su información de autenticación, y no el servidor. Así las peticiones pueden llegar a cualquier instancia del servidor y podrá ser atendida sin necesidad de sincronizaciones. Así mismo, diferentes plataformas podrán usar el mismo API. Además se incrementa la seguridad, evitando vulnerabilidades CSRF, al no existir sesiones.
- El refresh token es una identidad verificada y se usa para generar un nuevo access token cuando este expira. Típicamente, si el access token tiene fecha de expiración corta, una vez que caduca, el usuario tendría que autenticarse de nuevo para obtener un nuevo access token. Con el refresh token, que identifica al usuario y tiene una expiración mas generosa, este paso se puede saltar y con una petición al API obtener un nuevo access token que permita al usuario seguir accediendo de forma transparente a los recursos de la aplicación.

Patrón Single Sign On (SSO)

- El Single SingOn (SSO o inicio de sesión única) es un patrón arquitectónico para permitir a los usuarios el acceso a varias aplicaciones mediante una solo autenticación, utilizando para ello, un proveedor de autenticación en común. SSO permite delegar el proceso de autenticación a una entidad externa, la cual tiene como única responsabilidad autenticar que el usuario es quien dice ser, una vez autenticado recibe un token de identidad.
- Esto quieres decir que un usuario podría entrar a varias aplicaciones sin necesidad de tener que autenticarse en cada una, en su lugar, solo requerirá autenticarse la primera vez en cualquiera de las aplicaciones y posteriormente podrá acceder al resto si pasar por el proceso de autenticación.
- Hay varios tipos principales de SSO, también llamados reduced sign-on systems ("sistemas de autenticación reducida").
  - Enterprise SSO (E-SSO), también llamado legacy sso, funciona para una autenticación primaria, interceptando los requisitos de login presentados por las aplicaciones secundarias para completar los mismos con el usuario y contraseña. Los sistemas E-SSO permiten interactuar con sistemas que pueden deshabilitar la presentación de la pantalla de login.
  - Web SSO (Web-SSO), también llamado gestión de acceso web (web access management, Web-AM o WAM) trabaja solamente con aplicaciones y recursos accedidos vía web.
  - Kerberos es un método popular de externalizar la autenticación de los usuarios. Los usuarios se registran en el servidor Kerberos y reciben un tique, luego las aplicaciones cliente lo presentan para obtener acceso.
  - Identidad federada es una nueva manera de enfrentar el problema de la autenticación, también para aplicaciones Web. Utiliza protocolos basados en estándares para habilitar que las aplicaciones puedan identificar los clientes sin necesidad de autenticación redundante.
  - OpenID es un proceso de SSO distribuido y descentralizado donde la identidad se compila en un Localizador Uniforme de Recursos (URL) que cualquier aplicación o servidor puede verificar.

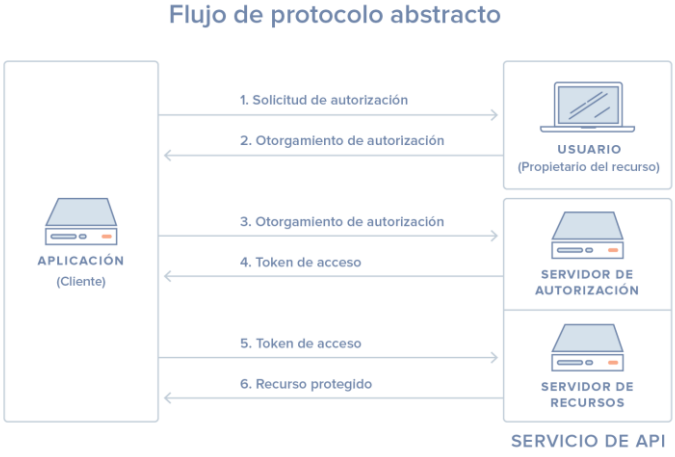
Ventajas y Desventajas del SSO

- Ventajas
  - Acceso rápido a las aplicaciones: Con el SSO, los usuarios pueden acceder rápidamente a múltiples aplicaciones y servicios con una sola autenticación.
  - Simplificación de la experiencia del usuario.
  - Mayor seguridad: Al utilizar el SSO, se puede implementar una autenticación más robusta y segura.
  - Administración simplificada: El SSO simplifica la administración de usuarios y contraseñas en un entorno empresarial. Los administradores pueden gestionar de manera centralizada las cuentas de usuario y los permisos de acceso, lo que facilita la incorporación y desactivación de usuarios en los
- Desventajas
  - Vulnerabilidad única: Si el SSO se ve comprometido, todas las aplicaciones y servicios vinculados a él también pueden estar en riesgo.
  - Dependencia de la disponibilidad del sistema SSO: Si el sistema SSO experimenta una interrupción o se vuelve inaccesible, los usuarios podrían perder el acceso a todas las aplicaciones y servicios vinculados. Es un potencial cuello de botellas.
  - Complejidad de implementación: La implementación de un sistema SSO puede ser compleja, especialmente en entornos empresariales con múltiples aplicaciones y sistemas.
  - Privacidad y confianza: Al utilizar el SSO, los usuarios deben confiar en que su proveedor de SSO protegerá adecuadamente sus datos personales y de inicio de sesión.

OAuth 2

- OAuth 2 es un protocolo de autorización que permite a las aplicaciones obtener acceso limitado a los recursos de usuario en un servicio HTTP, como Facebook, GitHub y Google. Delega la autenticación del usuario al servicio que aloja la cuenta del mismo y autoriza a las aplicaciones de terceros el acceso a dicha cuenta de usuario. OAuth define cuatro roles:
  - Propietario del recurso: Una entidad capaz de otorgar acceso a un recurso protegido. Cuando el propietario del recurso es una persona, se le conoce como usuario final.
  - Servidor de recursos: El servidor que aloja los recursos protegidos, capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso.
  - Cliente: Una aplicación que realiza solicitudes de recursos protegidos en nombre del propietario del recurso y con su autorización. El término "cliente" no implica ninguna característica de implementación particular.
  - Servidor de autorizaciones: El servidor que emite tokens de acceso al cliente después de haber realizado correctamente la autenticación y validar la concesión del propietario del recurso.

OAuth 2



Bearer Authentication

- La autenticación de portador (también llamada token de autenticación) es un esquema de autenticación HTTP que involucra tokens de seguridad llamados tokens de portador (Bearer). El nombre "Autenticación de portador" puede entenderse como "dar acceso al portador de este token". El token portador es una cadena encriptada, generalmente generada por el servidor en respuesta a una solicitud de inicio de sesión (Access token). El cliente debe enviar este token en el encabezado Authorization al realizar solicitudes a recursos protegidos:

  Authorization: Bearer <token>
- El esquema de autenticación Bearer se creó originalmente como parte de OAuth 2.0 en RFC 6750, pero a veces también se usa solo. De manera similar a la autenticación básica, la autenticación de portador solo debe usarse a través de HTTPS (SSL).

JWT: JSON Web Tokens

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier dato entre servicios de nuestra aplicación y asegurarnos de que sean siempre válido. Por ejemplo, si tenemos un servicio de envío de email, otro servicio podría enviar una petición con un JWT junto al contenido del mail o cualquier otro dato necesario y que estemos seguros que esos datos no fueron alterados de ninguna forma.

https://jwt.io

Tokens

- Los tokens son una serie de caracteres cifrados y firmados con una clave compartida entre servidor OAuth y el servidor de recurso o para mayor seguridad mediante clave privada en el servidor OAuth y su clave pública asociada en el servidor de recursos, con la firma el servidor de recursos el capaz de comprobar la autenticidad del token sin necesidad de comunicarse con él.
- Se componen de tres partes separadas por un punto: una cabecera con el algoritmo hash utilizado y tipo de token, un documento JSON con datos y una firma de verificación.
- El hecho de que los tokens JWT no sea necesario persistirlos en base de datos elimina la necesidad de tener su infraestructura, como desventaja es que no es tan fácil de revocar el acceso a un token JWT y por ello se les concede un tiempo de expiración corto.

- La infraestructura requiere varios elementos configurables de diferentes formas:
  - El servidor OAuth que realiza la autenticación y proporciona los tokens.
  - El servicio al que se le envía el token, es el que decodifica el token y decide conceder o no acceso al recurso.
  - En el caso de múltiples servicios con múltiples recursos es conveniente un gateway para que sea el punto de entrada de todos los servicios, de esta forma se puede centralizar las autorizaciones liberando a los servicios individuales.

Servidor de Autenticación/Autorizacion

- Dependencias: Spring Web y Spring Security
- En pom.xml

```
<dependency>
   <groupId>com.auth0</groupId>
   <artifactId>java-jwt</artifactId>
   <version>4.4.0</version>
</dependency>
```

- Configurar:

```
server.port=8081
spring.application.name=autentication-service
autenticacion.clave.secreta=Una clave secreta al 99% segura
autenticacion.expiracion.min=10
```

API de autenticación y obtención del token

```
@RestController
public class UserResource {
	@Value("${jwt.secret}")
	private String SECRET;
	@Value("${jwt.expiracion.mim:10}")
	private int EXPIRES_IN_MINUTES = 10;
	@Autowired
	PasswordEncoder passwordEncoder;
	@Autowired
	UsuarioRepositoy dao;

	@PostMapping(path = "/login", consumes = "application/json")
	public AuthToken loginJSON(@Valid @RequestBody BasicCredential credential) {
		var item = dao.findById(credential.getUsername());
		if (item.isEmpty() || !passwordEncoder.matches(credential.getPassword(), item.get().getPassword()))
			return new AuthToken();
		var usr = item.get();
		String token = JWT.create().withIssuer("MicroserviciosJWT").withClaim("usr", usr.getIdUsuario()).withArrayClaim("roles", usr.getRoles().toArray(new
			String[0])).withIssuedAt(new Date(System.currentTimeMillis())).withExpiresAt(new Date(System.currentTimeMillis() + EXPIRES_IN_MINUTES *
			60_000)).sign(Algorithm.HMAC256(SECRET));
		return new AuthToken(true, "Bearer " + token, usr.getNombre());
	}
}
```


Filtro de decodificación del token

```
public class JWTAuthorizationFilter extends OncePerRequestFilter {
	private final String HEADER = "Authorization";
	private final String PREFIX = "Bearer ";
	private String secret;

	public JWTAuthorizationFilter(String secret) {
		super();
		this.secret = secret;
	}

	@Override
	protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain chain) throws ServletException, IOException {
		try {
			String authenticationHeader = request.getHeader(HEADER);
			if (authenticationHeader != null && authenticationHeader.startsWith(PREFIX)) {
				DecodedJWT token = JWT.require(Algorithm.HMAC256(secret)).withIssuer("MicroserviciosJWT").build().verify(authenticationHeader.substring(PREFIX.length()));
				List<GrantedAuthority> authorities = token.getClaim("roles").asList(String.class).stream().map(role -> new SimpleGrantedAuthority(role)).collect(Collectors.toList());
				UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(token.getClaim("usr").toString(), null, authorities);
				SecurityContextHolder.getContext().setAuthentication(auth);
			}
		} catch(JWTVerificationException ex) {
			response.sendError(403, ex.getMessage());
		} finally {
			chain.doFilter(request, response);
		}
	}
}
```

Configurar con el filtro

```
@Configuration
@EnableMethodSecurity(prePostEnabled = true, securedEnabled = true, jsr250Enabled = true)
public class WebSecurityConfig {
  @Value("${jwt.secret}")
  private String SECRET;

   @Bean
   SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      return http
            .cors(Customizer.withDefaults())
            .csrf((csrf) -> csrf.disable())
            .sessionManagement((session) -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
            .addFilterAfter(new JWTAuthorizationFilter(SECRET), UsernamePasswordAuthenticationFilter.class)
            .authorizeHttpRequests(requests -> requests
               .requestMatchers(HttpMethod.GET, "/publico").permitAll()
               .anyRequest().authenticated()
             )
            .build();
   }
}
```


CORS

- La ejecución de aplicaciones JavaScript puede suponer un riesgo para el usuario que permite su ejecución.
- Por este motivo, los navegadores restringen la ejecución de todo código JavaScript a un entorno de ejecución limitado.
- Las aplicaciones JavaScript no pueden establecer conexiones de red con dominios distintos al dominio en el que se aloja la aplicación JavaScript.
- Los navegadores emplean un método estricto para diferenciar entre dos dominios ya que no permiten ni subdominios ni otros protocolos ni otros puertos.
- Si el código JavaScript se descarga desde la siguiente URL: http://www.ejemplo.com
- Las funciones y métodos incluidos en ese código no pueden acceder a:
  - https://www.ejemplo.com/scripts/codigo2.js
  - http://www.ejemplo.com:8080/scripts/codigo2.js
  - http://scripts.ejemplo.com/codigo2.js
  - http://192.168.0.1/scripts/codigo2.js

CORS

- Un recurso hace una solicitud HTTP de origen cruzado cuando solicita otro recurso de un dominio distinto al que pertenece.
- XMLHttpRequest sigue la política de mismo-origen, por lo que, una aplicación usando XHR solo puede hacer solicitudes HTTP a su propio dominio. Para mejorar las aplicaciones web, los desarrolladores pidieron que se permitieran a XHR realizar solicitudes de dominio cruzado.
- El Grupo de Trabajo de Aplicaciones Web del W3C recomienda el nuevo mecanismo de Intercambio de Recursos de Origen Cruzado (CORS, Cross-origin resource sharing: https://www.w3.org/TR/cors). Los servidores deben indicar al navegador mediante cabeceras si aceptan peticiones cruzadas y con que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, Content-Type, Accept, Authorization, X-XSRF-TOKEN"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS"
  - "Access-Control-Allow-Credentials", "true"
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

CORS

- Para configurar CORS en la interfaz del repositorio

```
@CrossOrigin(origins = "http://myDomain.com", maxAge = 3600, methods={RequestMethod.GET,
RequestMethod.POST })
public interface PersonaRepository extends JpaRepository<Persona, Integer> {
```

- Para configurar CORS globalmente

```
@Configuration @EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
   @Override
   public void addCorsMappings(CorsRegistry registry) {
      registry.addMapping("/api/**")
         .allowedOrigins("*")
         .allowedMethods("GET", "POST", "PUT", "DELETE")
         .allowedHeaders("origin", "content-type", "accept", "authorization")
         .allowCredentials(true).maxAge(3600);
   }
}
```

MONITORIZACIÓN Y RESILIENCIA

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.

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.

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.

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 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/.

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.

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

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.

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.

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

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.

Micrometer, Prometheus, Grafana

- Para exponer un actuator especifico para es necesario instalar la dependencia:

```
<dependency>
  <groupId>io.micrometer</groupId>
  <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

- Se puede verificar accediendo a /actuator/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: '/actuator/prometheus'
    scrape_interval: 5s
    static_configs:
      - targets: ['192.168.1.11:8010','host.docker.internal:8011']
```

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 dashboard constan de paneles, cada uno de los cuales representa una parte de la historia que se desea que cuente el dashboard. Se pueden importar dashboard 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.

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).

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 subprocesos, 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 caiga en cascada a otros servicios?
- Solución:
  - La 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.

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

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:
  - Netfix 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)

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

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");
}
```


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
```

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());
}
```


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());
}
```

Micrometer Tracing y Zipkin

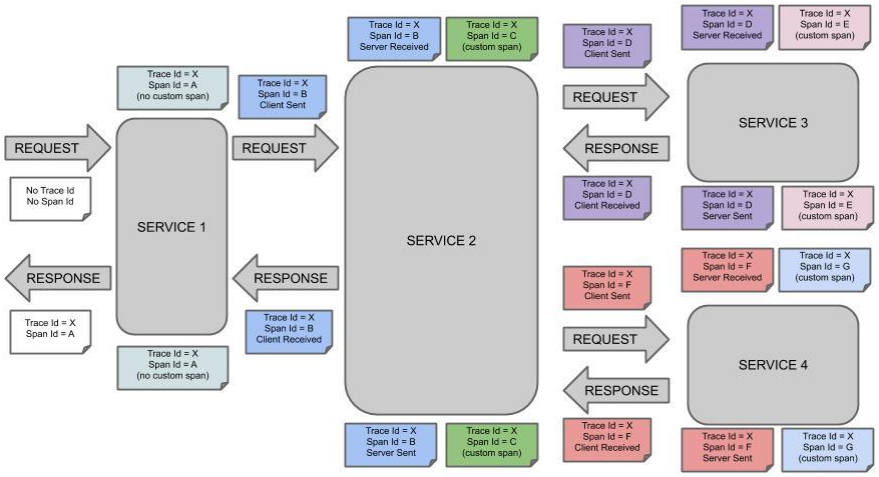
- Una de las funcionalidades esenciales en una aplicación distribuida es la trazabilidad de una petición, desde que entra por el API Gateway pasando por las diferentes peticiones que hacen los microservicios por la red o envío de mensajes. Es necesaria la funcionalidad que relacione las trazas de todos 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 transación. 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

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 diferencias 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.

Micrometer Tracing y Zipkin



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/

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();
}
```


Patrón: Log Aggregation

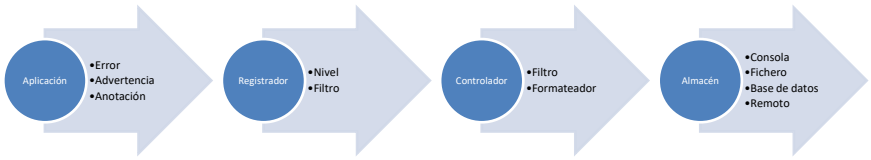
- Motivación:
  - Ha aplicado el patrón de arquitectura de microservicio. La aplicación consta de múltiples servicios e instancias de servicio que se ejecutan en varias máquinas. Las solicitudes suelen abarcar varias instancias de servicio.
  - Cada instancia de servicio genera información escrita sobre 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 la excepciones, se informa a un servicio de seguimiento de excepciones..

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.

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.



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)

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.

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.

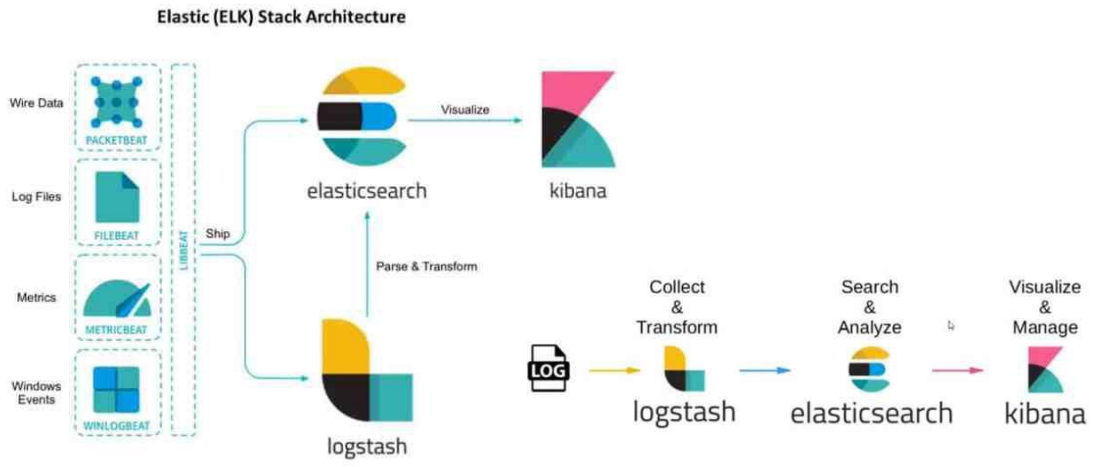
Realizar notificaciones

- 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
  ```

ELK: Elasticsearch, Logstash y Kibana

- "ELK" son las sigla 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 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.

ELK: Elasticsearch, Logstash y Kibana



Patrón: Service mesh

- Motivación:
  - Ha aplicado el patrón de arquitectura de microservicios y ha diseñado su sistema como un conjunto de servicios.
- Problema:
  - Se deben implementar numerosas preocupaciones transversales que incluyen:
    - Configuración externalizada: incluye credenciales y ubicaciones de red de servicios externos, como bases de datos y agentes de mensajes.
    - Registro: configuración de un marco de registro como log4j o logback
    - Comprobaciones de estado: extremo al que un servicio de supervisión puede "hacer ping" para determinar el estado.
    - Métricas: mediciones que brindan información sobre lo que está haciendo la aplicación y cómo se está desempeñando.
    - Seguimiento distribuido: servicios que asigna a cada solicitud externa un identificador único que se pasa entre servicios.
- Solución:
  - Utilizar una malla de servicios que medie entre toda la comunicación dentro y fuera de cada servicio.
- Implementación:
  - Istio, Linkerd, Envoy, …
- Patrones relacionados:
  - Una malla de servicios a menudo se implementa utilizando el patrón Sidecar.

DESPLIEGUE

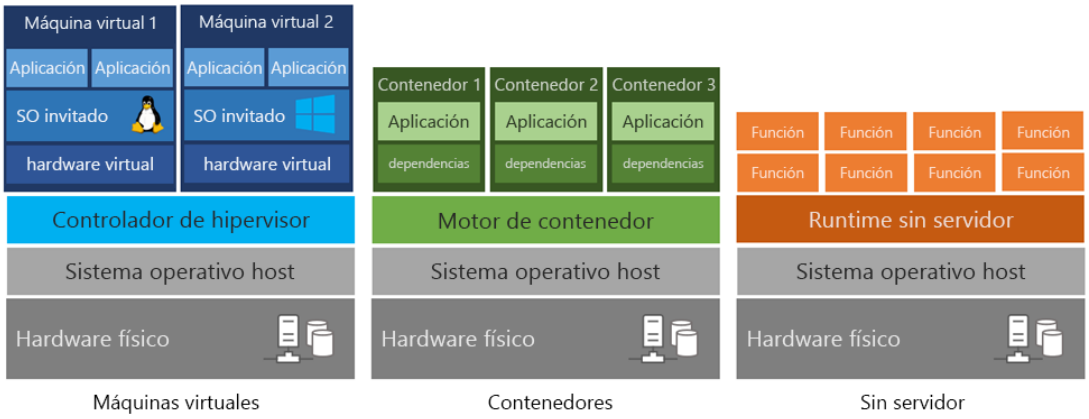
Modelo de despliegue

- 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.

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).

Patrones de despliegue

- Multiple service instances per host
  - Ejecutar varias instancias de diferentes servicios en un host (máquina física o virtual).
- Service instance per host
  - Implementar cada instancia de servicio individual en su propio host
- Service instance per VM
  - Empaquetar el servicio como una imagen de máquina virtual e implementar cada instancia del servicio como una VM separada
- Service instance per Container
  - Empaquetar el servicio como una imagen de contenedor (Docker) e implementar cada instancia del servicio como un contenedor
- Serverless deployment
  - Utilizar una infraestructura de implementación que oculte cualquier concepto de servidores (es decir, recursos reservados o preasignados), hosts físicos o virtuales, o contenedores. La infraestructura toma el código del servicio y lo ejecuta. Se factura por cada solicitud en función de los recursos consumidos.
- Service deployment platform (API Management)
  - Utilizar una plataforma de implementación, que es una infraestructura automatizada para la implementación de aplicaciones.

Despliegue

- Empaquetar la aplicación
  - mvnw clean package
- Crear el fichero "dockerfile":

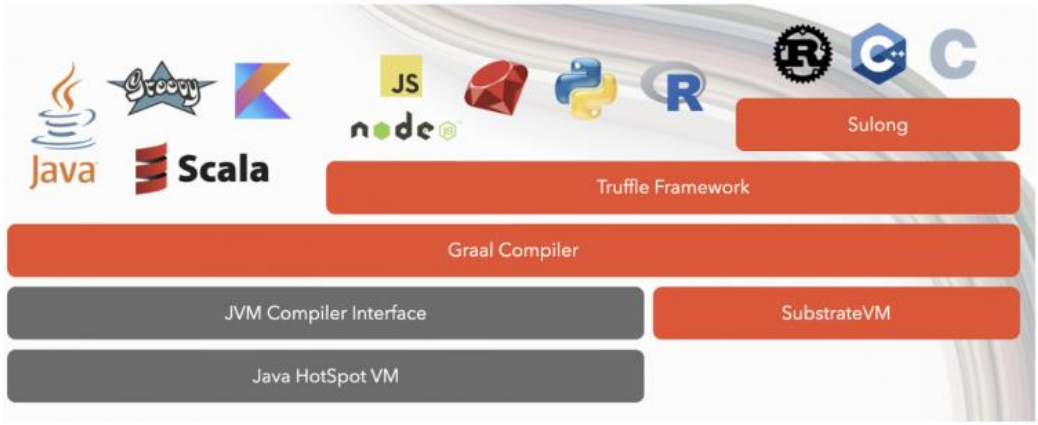
```
FROM openjdk:17
COPY target/ms.eureka-0.0.1-SNAPSHOT.jar /usr/app.jar
EXPOSE 8761
ENTRYPOINT ["java","-jar","/usr/app.jar"]
```

- Crear imagen:
  - docker build -t ms-eureka-server .
- Crear y ejecutar contenedor:
  - docker run -d --name ms-eureka-server -p 8761:8761 ms-eureka-server

GraalVM

- GraalVM, es una máquina virtual creada por Oracle Labs como podría serlo la tradicional JVM. No obstante, cabe destacar algunas mejoras. En primer lugar, es políglota, es decir, una máquina virtual capaz de ejecutar código en diversos lenguajes de programación. Por otro lado, incorpora un nuevo concepto de compilador de tipo AOT (ahead of time) que predice todo lo que nuestro código necesitará para ejecutarse lo que permite la creación de imágenes nativas.
- GraalVM es un conjunto de herramientas que puede ser utilizado de diversas formas:
  - Graal compiler puede entenderse como un compilador de código Java tradicional, con la diferencia que podemos “elegir” usar JIT (just in time) o el nuevo AOT (ahead of time).
  - SubstrateVM es un runtime necesario para ejecutar el AOT compiler de la JVM y generar las imágenes nativas.
  - Truffle es un framework usado dentro de GraalVM como interprete de otros lenguajes como Ruby, R, Python, entre otros. C/C++, Fortran y otros requieren de Sulong.
  - GraalVM: el paquete completo de tecnologías que puede ser utilizado para diversos casos de uso, ejecutar código Java compilado con JIT, con AOT, ejecutar otros lenguajes, ejecutar código mezclando lenguajes, etc.
- GraalVM está disponible como ediciones GraalVM Enterprise, basada en Oracle JDK y con optimizaciones adicionales además del soporte, y GraalVM Community , basada en OpenJDK, e incluyen soporte para Java 11 y Java 17. Están disponibles para Linux y macOS en sistemas x86 de 64 bits y ARM de 64 bits, y para Windows en sistemas x86 de 64 bits.

Arquitectura GraalVM



Native Image

- Native Image es una tecnología innovadora que compila código Java en un ejecutable nativo independiente o en una biblioteca compartida nativa. El código de bytes de Java que se procesa durante la compilación de un ejecutable nativo incluye todas las clases de aplicaciones, dependencias, bibliotecas dependientes de terceros y cualquier clase de JDK que se requiera. Un ejecutable nativo autónomo generado es específico para cada sistema operativo individual y arquitectura de máquina que no requiere una JVM.
- Para ello, la imagen nativa deberá generarse dentro de un contenedor Linux que tenga GraalVM y native-images. Después de generarla, sería tan sencillo como copiar este ejecutable a otro contenedor con la misma arquitectura y lo más minimalista posible.
- Las limitaciones de imágenes nativas son debidas a compilar asumiendo un escenario cerrado. Entre las limitaciones más conocidas se encuentran el uso de clases Java que usen reflection, escaneo de classpath y proxys dinámicos.

Spring Native

- Spring Native, disponible desde Spring 6, permite convertir proyectos Spring Bot 3 en imágenes nativas. Lo que permite una mejora considerable en el arranque y una menor sobrecarga de memoria en tiempo de ejecución si lo comparamos con la JVM.
- Hay que agregar la dependencia:
  - Developer Tools > GraalVM Native Support
- La aplicación nativa se puede construir de la siguiente manera:
  - mvnw spring-boot:build-image
- Para ejecutar la aplicación, se utiliza Docker de la forma habitual:
  - docker run -d --name eureka-server -p 8761:8761 ms.eureka.server:1.0.0

Spring Native

```
<plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
          <image>
            <createdDate>now</createdDate>
          </image>
        </configuration>
</plugin>
```