

Microservicios con Spring

© JMA 2020. All rights reserved

Contenidos

- Arquitectura de microservicios
 - Definiciones y Antecedentes
 - Arquitectura Monolítica vs Arquitectura Microservicios
 - Características, ventajas y desventajas
 - Buenas prácticas
 - Implementación y Despliegue
- Clean Architecture
 - Principios SOLID
 - Antecedentes: Hexagonal Architecture y Onion Architecture
 - Principios y Características de una arquitectura limpia
 - Patrón arquitectónico MVC
 - Patrón arquitectónico MVVM
 - Arquitectura N-Capas con Orientación a Dominio (DDD)
 - Patrones de Inversión de Control e Inyección de Dependencias
 - Test Driven Development (TDD)
 - Refactorización
- Spring con Spring Boot
- IoC con Spring Core
- Gestión de datos
 - Patrón: Base de datos por servicio
 - Spring Data
 - MongoDB
 - Redis
 - Patrón: Saga
 - Patrón: API de Composición
 - Patrón: CORS
 - Patrón: Event sourcing
 - Patrón: Eventos de aplicación
- Estilos de comunicación
 - Patrón: Invocación a procedimiento remoto (RPI)
 - Spring Web MVC vs Spring WebFlux
- Documentar Servicios REST
 - Anotación del modelo
 - Anotación del servicio
 - Enfoque API First
 - OpenAPI (Swagger)
 - Swagger Editor, Swagger UI, Swagger Codegen
- Registro/Descubrimiento
 - Patrón: Registro de servicios
 - Patrón: Auto registro
 - Patrón: Registro de terceros
 - Eureka
 - Patrón: Descubrimiento del lado del cliente
 - Spring Cloud LoadBalancer
 - RestTemplate, WebClient y Feign
 - Patrón: API Gateway
 - Spring Cloud Gateway
- Seguridad
 - Spring Cloud Config
 - CORS
 - Spring Security
 - Patrón: token de acceso
 - JWT: JSON Web Tokens
- Monitorización y Resiliencia
 - Patrones de Observabilidad
 - Patrón: Health Check API
 - Spring Boot 2.x Actuator
 - Spring Boot Admin
- Pruebas
 - Automatización de pruebas
 - JUnit, DBUnit y Spring Boot Test
 - SoapUI, Postman y JMeter
- Despliegue
 - Patrones de despliegue
 - Docker
 - Kubernetes
 - Patrón: Sidecar
 - Service Mesh: Istio
- Micrometer, Prometheus, Grafana
- Patrón: Circuit Breaker
- Spring Cloud Circuit Breaker con Resilience4j
- Patrón: Distributed tracing
- Spring Cloud Sleuth y Zipkin
- Patrón: Log aggregation
- ELK: Elasticsearch, Logstash y Kibana

© JMA 2020. All rights reserved

INTRODUCCIÓN

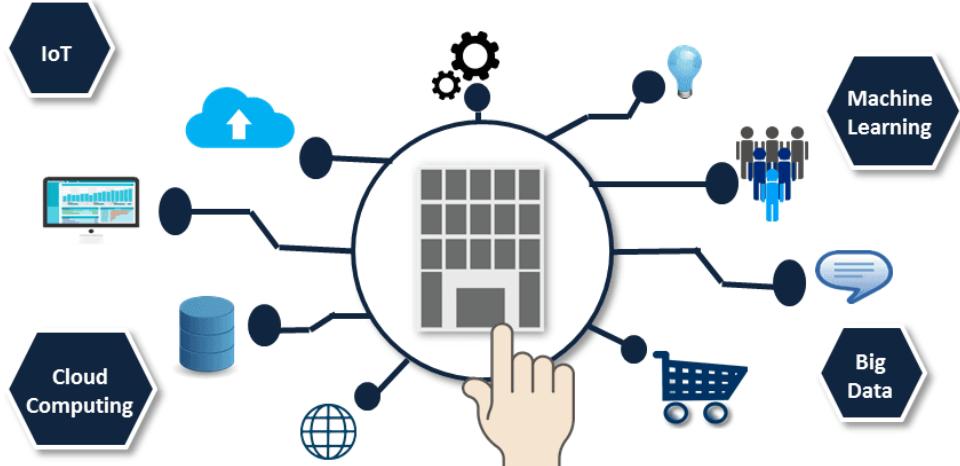
© JMA 2020. All rights reserved

La transformación digital

- Los clientes buscan establecer relaciones con las empresas a través de diferentes dispositivos y canales. La transformación digital permite a las empresas responder con agilidad a las demandas de interrelación con clientes y otras partes interesadas. Las APIs (Application Programming Interface) son el mecanismo en el que se basa esta capacidad de respuesta. Mediante una API el desarrollador es capaz de construir una aplicación que dé respuesta a las necesidades de un grupo de clientes o que permita la comunicación directa entre organizaciones.
- Las APIs actuales tienen su origen en los primeros intentos de integración de sistemas donde las organizaciones comunicaban diferentes sistemas en aras de automatizar procesos manuales de obtener una información global de sus sistemas. Estos sistemas actualmente han logrado contar con interfaces sencillos, con mecanismos de comunicación estándar y manejo de información de forma simple y desacoplada. Todo lo cual facilita su utilización de forma ágil por aplicaciones móviles, IoT u otras aplicaciones dentro o fuera de una organización.
- Igualmente, al basar su comunicación en el protocolo HTTP, las APIs facilitan la comunicación con o desde sistemas en la nube. Abriendo de esta forma el abanico de servicios con el que una organización puede dar servicio a sus clientes o los canales a través de los cuales dota dichos servicios. El paradigma de comunicaciones y consumo de servicios digitales actual conlleva que para garantizar el éxito en la adopción de un producto las organizaciones no solo deban proporcionar el mejor producto si no la mejor forma de integrarse digitalmente con el servicio de las APIs.
- La transformación digital se puede definir como la integración de las nuevas tecnologías en todas las áreas de una organización u otros ámbitos para cambiar su forma de funcionar.

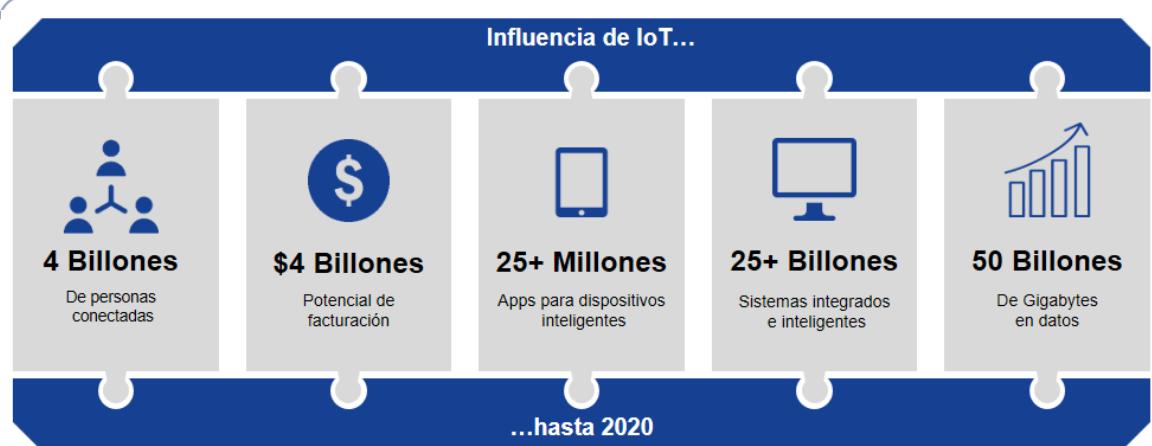
© JMA 2020. All rights reserved

Transformación digital



© JMA 2020. All rights reserved

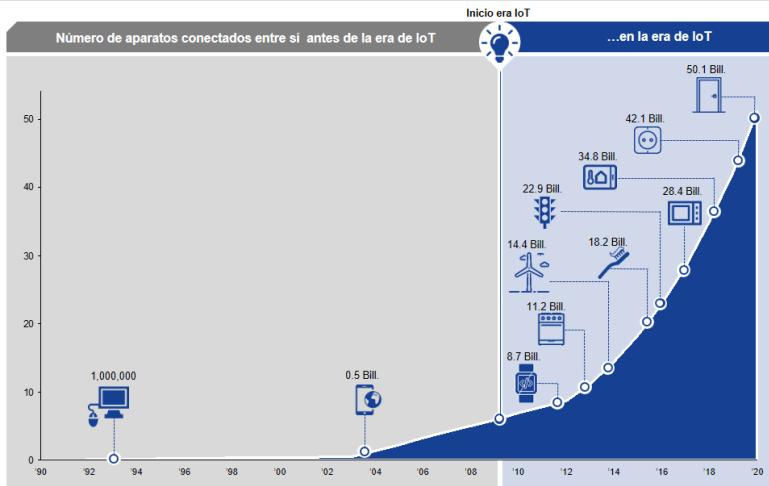
Influencia de IoT



© JMA 2020. All rights reserved

<https://www.fostec.com/wp-content/uploads/Internet-de-las-cosas-1-ES.png>

Desarrollo explosivo de IoT



© JMA 2020. All rights reserved

<https://www.fostec.com/wp-content/uploads/Internet-de-las-cosas-2-ES.png>

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.

© JMA 2020. All rights reserved

Cambio de paradigma

Local tradicional

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

Nube moderna

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

© JMA 2020. All rights reserved

¿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.

© JMA 2020. All rights reserved

Estilos de arquitectura

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

© JMA 2020. All rights reserved

Arquitectura basada en eventos

- Una arquitectura basada en eventos consta de productores de eventos que generan un flujo de eventos, y consumidores de eventos que escuchan los eventos. Los eventos se entregan casi en tiempo real, de modo que los consumidores pueden responder inmediatamente a los eventos cuando se producen. Los productores se desconectan de los consumidores, no saben si los consumidores están escuchando. Los consumidores también se desconectan entre sí, y cada consumidor ve todos los eventos.
- Una arquitectura basada en eventos puede usar un modelo pub/sub o un modelo de flujo de eventos.
 - Pub/sub: la infraestructura de mensajería mantiene el seguimiento de las suscripciones. Cuando se publica un evento, se envía el evento a cada suscriptor. Después de que se consume un evento, no se puede reproducir y los nuevos suscriptores no ven el evento.
 - Streaming de eventos: los eventos se escriben en un registro. Los eventos siguen un orden estricto (dentro de una partición) y son duraderos. Los clientes no se suscriben al flujo, sino que un cliente puede leer desde cualquiera de sus partes y es responsable de avanzar su posición en el flujo. Por lo que un cliente puede unirse en cualquier momento y puede reproducir los eventos.

© JMA 2020. All rights reserved

Arquitectura basada en eventos

- En el lado del consumidor, hay algunas variaciones comunes:
 - Procesamiento sencillo de eventos: Un evento desencadena inmediatamente una acción en el consumidor.
 - Procesamiento de eventos complejos: El consumidor procesa una serie de eventos, en busca de patrones en los datos de eventos.
 - Procesamiento de flujo de eventos: Los procesadores de flujos sirven para procesar o transformar el flujo. Puede haber varios procesadores de flujo para diferentes subsistemas de la aplicación.
- El origen de los eventos puede ser externo con respecto al sistema, como dispositivos físicos en una solución de IoT. En ese caso, el sistema debe ser capaz de ingerir los datos según el volumen y el rendimiento que requiere el origen de datos. En algunos sistemas, como IoT, los eventos se deben ingerir en volúmenes muy altos.
- En la práctica, es habitual tener varias instancias de un consumidor para evitar hacer que el consumidor se convierta en un único punto de error en el sistema. También podrían ser necesarias varias instancias para administrar el volumen y la frecuencia de los eventos.

© JMA 2020. All rights reserved

Arquitectura basada en eventos

- Esta arquitectura se debe utilizar para: Procesamiento de los mismos eventos en varios subsistemas , Procesamiento en tiempo real con retardo mínimo, Procesamiento de eventos complejos (como coincidencia de patrones o agregación durante ventanas de tiempo) y Procesamiento de un gran volumen y alta velocidad de datos (como IoT).
- Ventajas
 - Se desvinculan productores y consumidores.
 - No existen integraciones de punto a punto. Es fácil agregar nuevos consumidores al sistema.
 - Los consumidores pueden responder a eventos inmediatamente a medida que llegan.
 - Muy escalable y distribuida.
 - Los subsistemas tienen vistas independientes del flujo de eventos.
- Desafíos
 - Entrega garantizada: En algunos sistemas, especialmente en escenarios de IoT, es fundamental garantizar la entrega de los eventos.
 - Procesamiento de eventos en orden o exactamente solo una vez: Cada tipo de consumidor normalmente se ejecuta en varias instancias, a fin de conseguir resiliencia y escalabilidad. Esto puede suponer un desafío si se deben procesar los eventos en orden (dentro de un tipo de consumidor) o si la lógica de procesamiento no es idempotente.

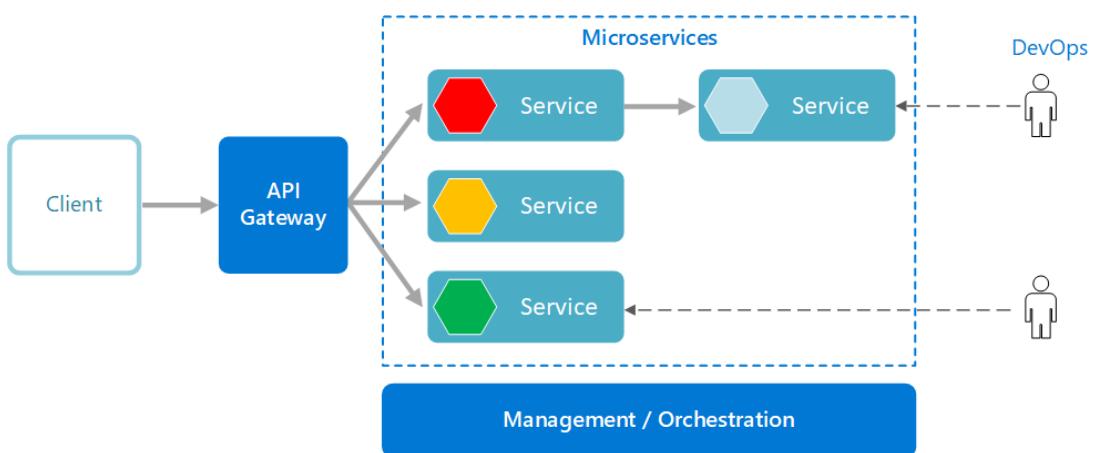
© JMA 2020. All rights reserved

Arquitectura de microservicios

- Una arquitectura de microservicios consta de una colección de servicios autónomos y pequeños. Los servicios son independientes entre sí y cada uno debe implementar una funcionalidad de negocio individual.
- Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa. Un único equipo reducido de programadores puede escribir y mantener un servicio.
- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.
- Los servicios se comunican entre sí mediante APIs bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- No es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos de trabajo.
- El estilo de microservicio surge como alternativa al estilo monolítico.

© JMA 2020. All rights reserved

Arquitectura de microservicios



© JMA 2020. All rights reserved

Arquitectura de microservicios

- Además de los propios servicios, hay otros componentes que aparecen en una arquitectura típica de microservicios:
 - **Administración e implementación:** Este componente es el responsable de la colocación de servicios en los nodos, la identificación de errores, el reequilibrio de servicios entre nodos, etc. Normalmente, este componente es una tecnología estándar, como Kubernetes, en lugar de algo creado de forma personalizada.
 - **Puerta de enlace de API:** Es el punto de entrada para los clientes. En lugar de llamar a los servicios directamente, los clientes llaman a la puerta de enlace de API, que reenvía la llamada a los servicios apropiados en el back-end. Entre las ventajas de usar una puerta de enlace de API se encuentran las siguientes:
 - Desacoplan los clientes de los servicios. Los servicios pueden cambiar de versión o refactorizarse sin necesidad de actualizar todos los clientes.
 - Los servicios pueden utilizar los protocolos de mensajería que no son fáciles de usar para un servicio web, como AMQP.
 - La puerta de enlace de API puede realizar otras funciones transversales como la autenticación, el registro, la terminación SSL y el equilibrio de carga.

© JMA 2020. All rights reserved

Ventajas de los microservicios

- **Agilidad:** Dado que microservicios se implementan de forma independiente, resulta más fácil de administrar las correcciones de errores y las versiones de características. Puede actualizar un servicio sin volver a implementar toda la aplicación y revertir una actualización si algo va mal.
- **Equipos pequeños y centrados:** Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo de características lo pueda compilar, probar e implementar. Los equipos pequeños favorecen la agilidad.
- **Base de código pequeña:** En las aplicaciones monolíticas, con el paso del tiempo se da la tendencia de que las dependencias del código se acaben enredarse, por lo que para agregar una nueva característica, es preciso tocar el código en muchos lugares. Al no compartir el código ni los almacenes de datos, la arquitectura de microservicios minimiza las dependencias y resulta más fácil agregar nuevas características.
- **Mezcla de tecnologías:** Los equipos pueden elegir la tecnología que mejor se adapte al servicio de una combinación de pilas de tecnología, según corresponda.
- **Aislamiento de defectos mejorado:** Si un microservicio individual no está disponible o genera problemas, no interrumpe toda la aplicación, siempre que los microservicios de nivel superior estén diseñados para controlar los errores correctamente (por ejemplo, circuit break).
- **Escalabilidad:** Los servicios se pueden escalar de forma independiente, lo que permite escalar horizontalmente los subsistemas que requieren más recursos, sin tener que escalar horizontalmente toda la aplicación. Mediante un orquestador como Kubernetes o Service Fabric se puede empaquetar una mayor densidad de servicios en un solo host, lo que aumenta la eficacia en el uso de los recursos.
- **Aislamiento de los datos:** Al afectar a un solo microservicio, facilita realizar actualizaciones del esquema.

© JMA 2020. All rights reserved

Inconvenientes de los microservicios

- **Complejidad:** La arquitectura de microservicios es un estilo de programación distribuida. Cada servicio es más sencillo, pero el sistema como un todo es más complejo.
- **Desarrollo y pruebas:** Sobrecarga a los desarrolladores que deben implementar el mecanismo de comunicación entre servicios. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios. La refactorización en los límites del servicio puede resultar difícil. También supone un desafío probar las dependencias de los servicios, especialmente cuando la aplicación está evolucionando rápidamente. La prueba es más difícil, requiere un mayor peso en las pruebas de integración.
- **Falta de gobernanza:** El enfoque descentralizado para la generación de microservicios tiene ventajas, pero también puede causar problemas, se puede acabar con tantos lenguajes y marcos de trabajo diferentes que la aplicación puede ser difícil de mantener.
- **Congestión y latencia de red:** El uso de muchos servicios pequeños y detallados puede dar lugar a más comunicación interservicios. Además, si la cadena de dependencias del servicio se hace demasiado larga, la latencia adicional puede constituir un problema. Tendrá que diseñar las APIs con atención. Hay que evitar que las APIs se comuniquen demasiado, pensar en formatos de serialización y buscar lugares para utilizar patrones de comunicación asíncrona.
- **Integridad de datos:** Cada microservicio es responsable de la conservación de sus propios datos y, como consecuencia, mantener la coherencia de los datos es un problema. Implementar casos de uso que abarcan múltiples servicios sin usar transacciones distribuidas es difícil.
- **Administración:** El mayor consumo de recursos, la complejidad del despliegue y la complejidad operativa de implementar y administrar un sistema que comprende muchos tipos componentes y servicios diferentes requiere una cultura de DevOps consolidada. El registro correlacionado entre servicios puede resultar un desafío.
- **Control de versiones:** Las actualizaciones de un servicio no deben interrumpir servicios que dependen de él. Es posible que varios servicios se actualicen en cualquier momento; por lo tanto, sin un cuidadoso diseño, podrían surgir problemas con la compatibilidad con versiones anteriores o posteriores.
- **Conjunto de habilidades:** Cada microservicio requiere una implementación completa, incluyendo interfaz de usuario, almacenamiento persistente y colaboración externa. En consecuencia, los equipos deben ser interdisciplinares, incluyendo toda la gama de habilidades.

© JMA 2020. All rights reserved

Estilos de comunicación

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

© JMA 2020. All rights reserved

Estilos arquitectónicos

- Precursoros:**

- RPC: Llamadas a Procedimientos Remotos
- Binarios: CORBA, Java RMI, .NET Remoting
- XML-RPC: Precursor del SOAP

- Actuales:**

- Servicios Web XML o Servicios SOAP
- Servicios Web REST o API REST
 - WebHooks
- Servicios GraphQL
- Servicios gRPC

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

© JMA 2020. All rights reserved

Estilos arquitectónicos

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

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 más 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.

© JMA 2020. All rights reserved

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, ...

© JMA 2020. All rights reserved

Servicios REST

- Inconvenientes:
 - Uso de HTTP: Aunque es la infraestructura más 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 establece 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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 el 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 más empinada que la de REST.

© JMA 2020. All rights reserved

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 esperar a nuevas solicitudes. Signalr usa WebSocket cuando está disponible o recurre a las otras cuando es necesario.

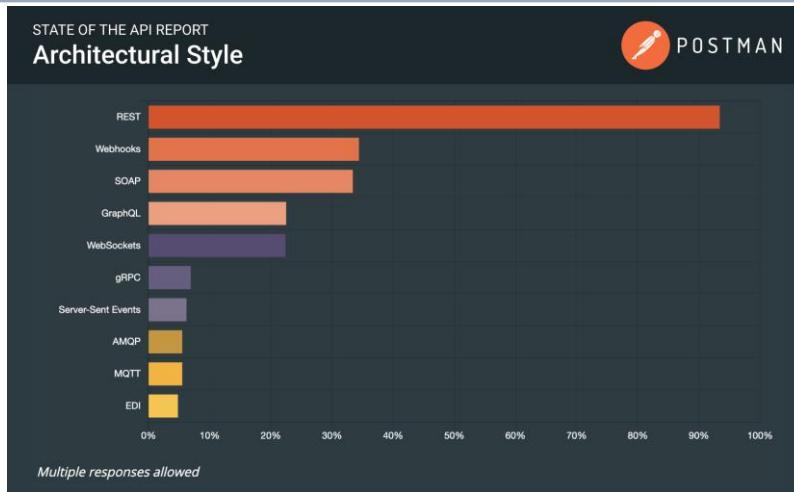
© JMA 2020. All rights reserved

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 electrónicos de marketing para notificar al servidor que debe enviar un nuevo correo electrónico porque el usuario ha abierto el mensaje.
- Pueden considerarse una versión especializada y simplificada de los servicios REST (solo POST).

© JMA 2020. All rights reserved

Estilos arquitectónicos más utilizados



© JMA 2020. All rights reserved

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

Tipos de API por propósito

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

© JMA 2020. All rights reserved

Tipos de API por estrategias de gestión

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

© JMA 2020. All rights reserved

Preocupaciones transversales

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

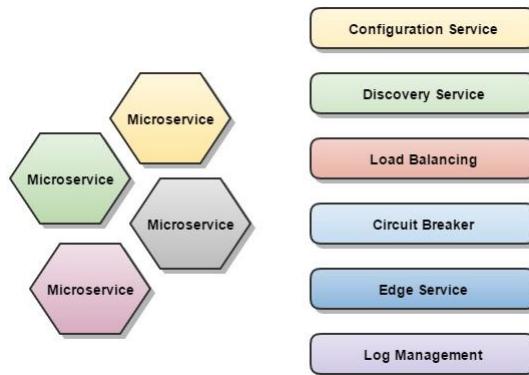
© JMA 2020. All rights reserved

Implantación

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

© JMA 2020. All rights reserved

Modelo de referencia



© JMA 2020. All rights reserved

Modelo de referencia

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

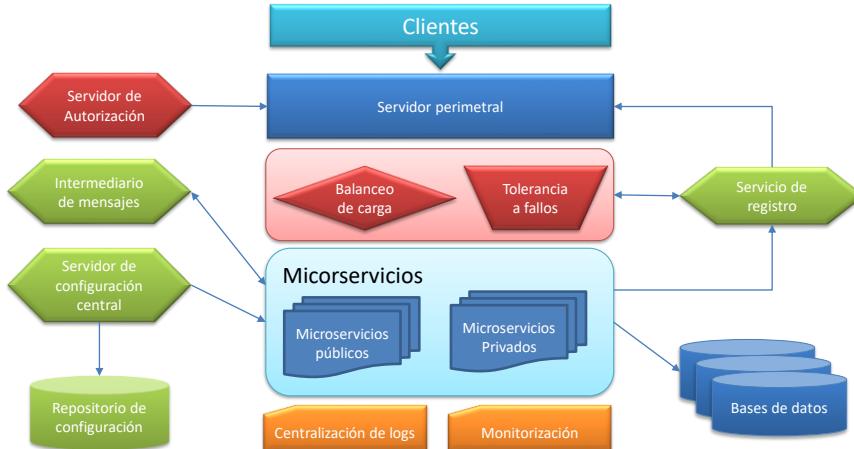
© JMA 2020. All rights reserved

Modelo de referencia

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

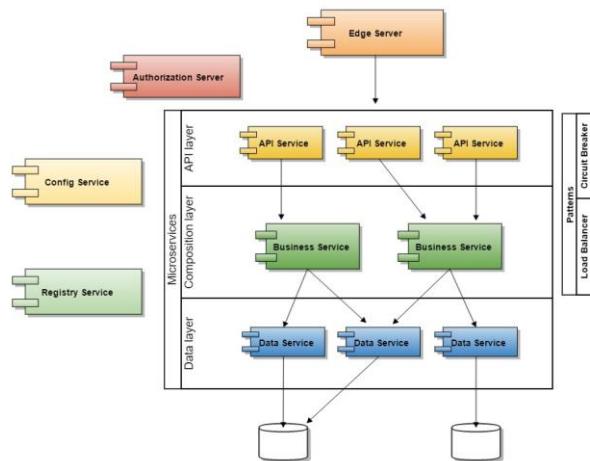
© JMA 2020. All rights reserved

Modelo de referencia



© JMA 2020. All rights reserved

Modelo de referencia



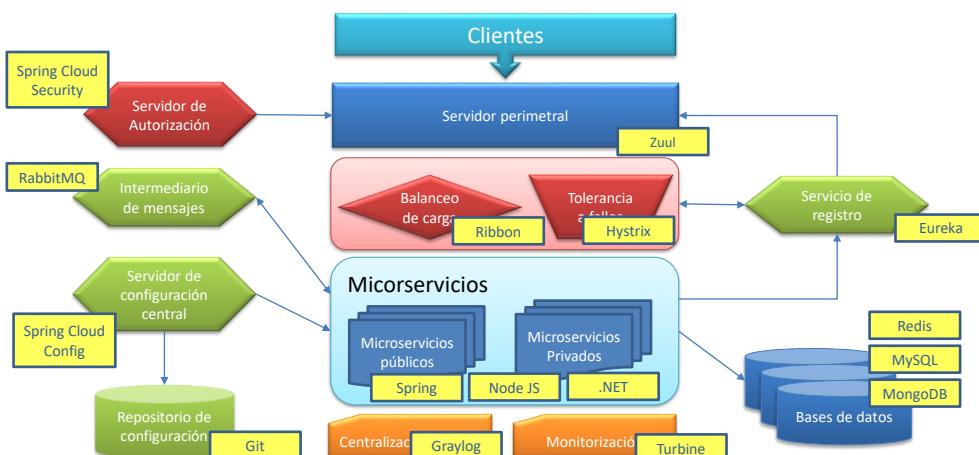
© JMA 2020. All rights reserved

Modelo de implementación (Netflix OSS)

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

© JMA 2020. All rights reserved

Modelo de implementación (Netflix OSS)



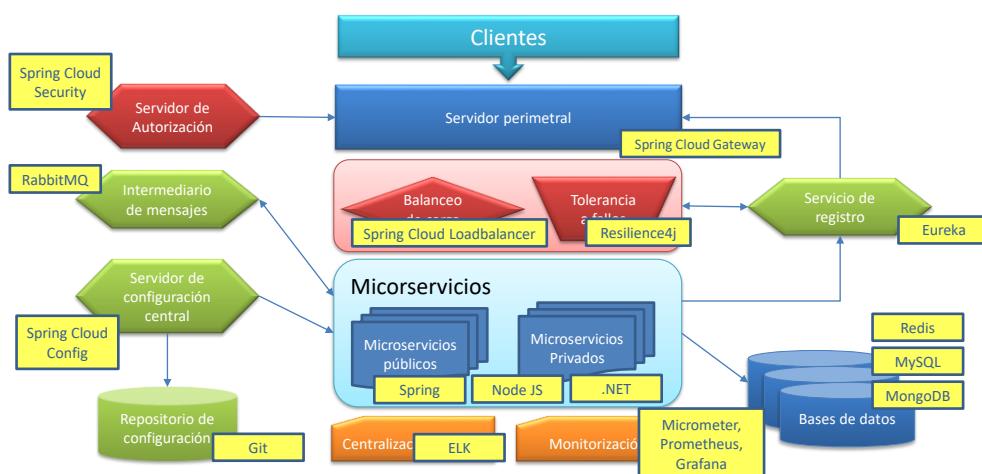
© JMA 2020. All rights reserved

Modelo de implementación (Spring Cloud)

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

© JMA 2020. All rights reserved

Modelo de implementación (Spring Cloud)



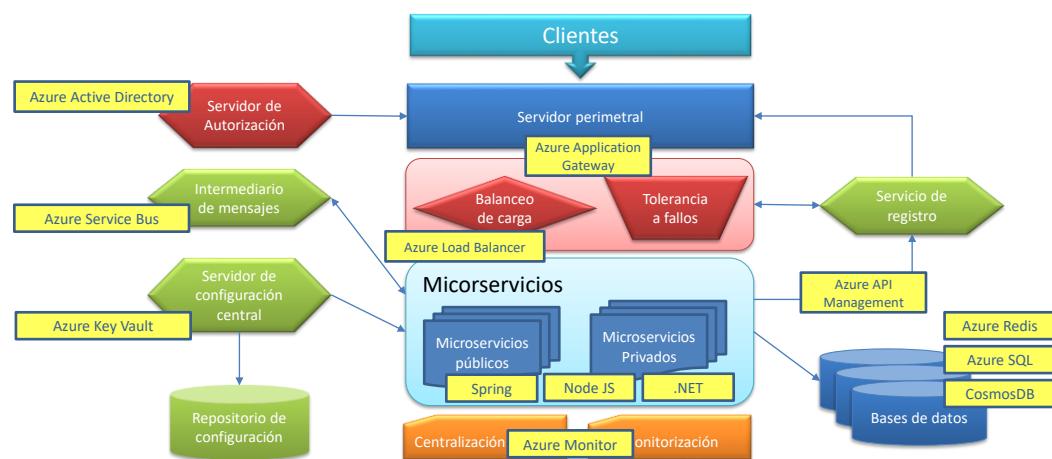
© JMA 2020. All rights reserved

Modelo de implementación (Azure)

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

© JMA 2020. All rights reserved

Modelo de implementación (Azure)



© JMA 2020. All rights reserved

Modelo de despliegue

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

© JMA 2020. All rights reserved

API Economy

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

© JMA 2020. All rights reserved

API Strategy

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

© JMA 2020. All rights reserved

Componentización a través de APIs

- Un componente es una unidad de software que es reemplazable y actualizable de manera independientemente.
- Definimos las librerías como componentes que están vinculados a un programa y se llaman mediante llamadas a función en memoria, mientras que los APIs son componentes fuera de proceso que se comunican con un mecanismo como una solicitud de servicio web o una llamada a procedimiento remoto.
- Las arquitecturas de APIs usarán librerías, pero su manera primaria de componentización y reutilización es dividir en servicios.
- La razón principal para usar servicios como componentes (en lugar de bibliotecas) es que los servicios son desplegables de forma independiente.
- Otra consecuencia es una interfaz de componentes más explícita.

© JMA 2020. All rights reserved

Organización de equipos

- Cuando se busca dividir un sistema grande en partes, a menudo la administración de equipos de trabajo se centra en la capa tecnológica, lo que lleva a tener equipos de interfaz de usuario, equipos de lógica de servidor y equipos de bases de datos. Cuando los equipos están separados de esta manera, incluso los cambios más simples pueden conducir a proyectos cruzados entre equipos que suponen tiempo y coste.
- La estrategia API es diferente, dividiendo los equipos por servicios organizados alrededor de la capacidad empresarial.
- Cada servicio requiere una implementación completa de software, incluyendo interfaz de usuario, almacenamiento persistente y colaboración externa.
- En consecuencia, los equipos son interdisciplinares, incluyendo toda la gama de habilidades necesarias para el desarrollo: experiencia de usuario, base de datos y gestión de proyectos.

© JMA 2020. All rights reserved

Productos y Gobernanza

• Productos no Proyectos

- La mayoría de los esfuerzos de desarrollo de aplicaciones que vemos utilizan un modelo de proyecto: donde el objetivo es entregar algún software que se considera completado.
- Al terminar el software se entrega a una organización de mantenimiento y el equipo de proyecto que lo construyó se disuelve.
- La estrategia API tiende a evitar este modelo, prefiriendo en cambio la noción de que un equipo debe poseer un producto durante toda su vida útil.

• Gobernanza descentralizada

- Una de las consecuencias de la gobernanza centralizada es la tendencia a estandarizar las plataformas con tecnología única. La experiencia demuestra que este enfoque es limitante.
- La estrategia API debe permitir usar la herramienta adecuada para cada caso pero manteniendo unas directrices comunes y establecer una cultura anti-burocracia: libertad con responsabilidad.

© JMA 2020. All rights reserved

Puntos finales inteligentes y conexiones tontas

- Al construir estructuras de comunicación entre diferentes procesos, hemos visto muchos productos y enfoques que ponen énfasis en el mecanismo de comunicación.
- Un buen ejemplo de esto es el Enterprise Service Bus (ESB), donde los productos de ESB a menudo incluyen sofisticadas instalaciones para enrutamiento de mensajes, coordinación, transformación y aplicación de reglas de negocio.
- La estrategia API favorece un enfoque alternativo: **puntos finales inteligentes y conexiones tontas**.
- Las aplicaciones construidas a partir de APIs tienen como objetivo estar tan desacopladas y cohesivas como sea posible (poseen su propia lógica de dominio y actúan más como filtros) recibiendo una petición, aplicando la lógica según corresponda y produciendo una respuesta.
- Estos son coordinados utilizando simples protocolos REST (HTTP/HTTPS, WebSockets o AMQP) en lugar de complejos protocolos como WS o BPEL u orquestación de una herramienta central.

© JMA 2020. All rights reserved

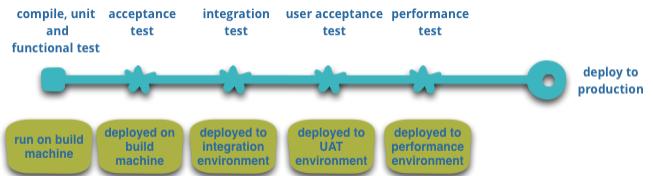
Gestión descentralizada de datos

- La descentralización de la gestión de datos se presenta de diferentes maneras.
- En el nivel más abstracto, significa que el modelo conceptual diferirá entre sistemas. Este es un problema común cuando se necesita hacer integración en una gran empresa.
- Una forma útil de pensar sobre esto es la noción de Contexto delimitado del Diseño Dirigido por Dominio (DDD). DDD divide un dominio complejo en múltiples contextos acotados y mapea las relaciones entre ellos.
- Antiguamente se recomendaba construir un modelo unificado de toda la empresa, pero hemos aprendido que “la unificación total del modelo de dominio para un sistema grande no será factible ni rentable”.
- Dicha unificación a conllevado la preponderancia del modelo relacional frente a modelos no-sql mas apropiados en determinados escenarios.
- Este proceso es útil tanto para arquitecturas monolíticas como para una estrategia API.
- La estrategia API prefieren dejar que cada servicio administre su propia fuente de datos, ya sea diferentes instancias de la misma tecnología de base de datos, o sistemas de base de datos completamente diferentes, un enfoque llamado Polyglot Persistence.

© JMA 2020. All rights reserved

Automatización de Infraestructura

- Las técnicas de automatización de la infraestructura han evolucionado enormemente en los últimos años: la evolución de la nube y de AWS o Azure en particular ha reducido la complejidad operativa de la creación, implementación y operación de las APIs.
- Muchos de los productos o sistemas que se están construyendo con APIs están siendo construidos por equipos con amplia experiencia en Entrega Continua y su precursor, la Integración Continua. Los equipos que crean software de esta manera hacen un uso extensivo de las técnicas de automatización de infraestructura.



- Para realizar el proceso con garantías:
 - Requiere exhaustivas pruebas automatizadas.
 - La promoción del software en funcionamiento "hacia arriba" implica automatizar la implementación en cada nuevo entorno.

© JMA 2020. All rights reserved

Diseño tolerante a fallos

- Una consecuencia del uso de APIs como componentes distribuidos, es que las aplicaciones deben diseñarse de manera que puedan tolerar el fallo de los servicios. Cualquier llamada a un servicio podría fallar debido a la falta de disponibilidad del proveedor.
- Esto es una desventaja en comparación con un diseño monolítico ya que introduce complejidad adicional para manejarlo.
- Dado que los servicios pueden fallar en cualquier momento, es importante poder detectar los fallos rápidamente y restaurar automáticamente el servicio si es posible.
- La estrategia API debe poner mucho énfasis en el monitoreo en tiempo real de la aplicación, comprobando los elementos arquitectónicos (cuántas solicitudes por segundo tiene la base de datos) y métricas relevantes para el negocio (por ejemplo, cuántas órdenes por minuto se reciben).
- El monitoreo semántico puede proporcionar un sistema de alerta temprana de algo que va mal, lo que indicará a los equipos de desarrollo que deben investigarlo.

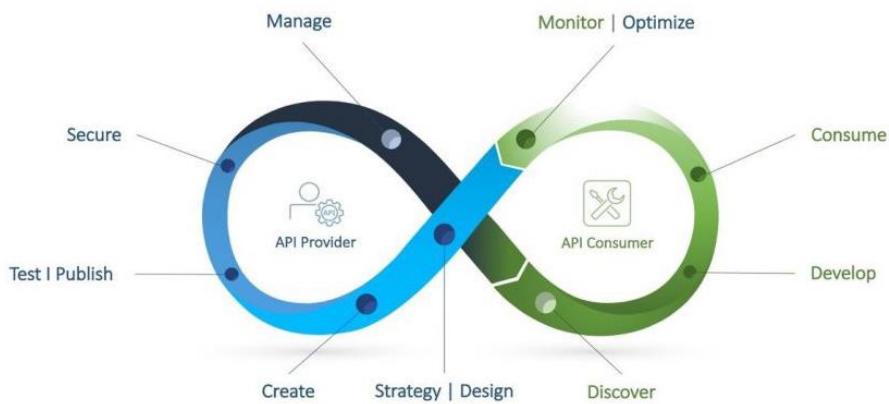
© JMA 2020. All rights reserved

Diseño Evolutivo

- La implementación de componentes en APIs añade una oportunidad para una planificación de entrega más granular.
- Con un monolito, cualquier cambio requiere una compilación completa y despliegue de toda la aplicación.
- Con la estrategia API, sin embargo, sólo es necesario volver a implementar el servicio que se modificó. La propiedad clave de un componente es la noción de reemplazo y la capacidad de actualización independientes, lo que implica que buscamos puntos en los que podamos imaginar reescribir un componente sin afectar a sus colaboradores.
- Esto puede simplificar y acelerar el proceso de entrega.
- En una estrategia API, solo se necesita volver a implementar los servicios que se modificaron. Esto puede simplificar y acelerar el proceso de lanzamiento aunque el inconveniente es que hay que preocuparse por si los cambios en un servicio rompan a sus consumidores.
- El enfoque de integración tradicional es tratar de abordar este problema utilizando el control de versiones, pero la preferencia en las estrategias API es utilizar sólo el versionado como último recurso: deberemos diseñar los servicios para que sean lo más tolerantes posible a los cambios en sus proveedores.

© JMA 2020. All rights reserved

Ciclo de vida



© JMA 2020. All rights reserved

Ciclo de vida

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

© JMA 2020. All rights reserved

Ciclo de vida



© JMA 2020. All rights reserved

CLEAN ARCHITECTURE

© JMA 2020. All rights reserved

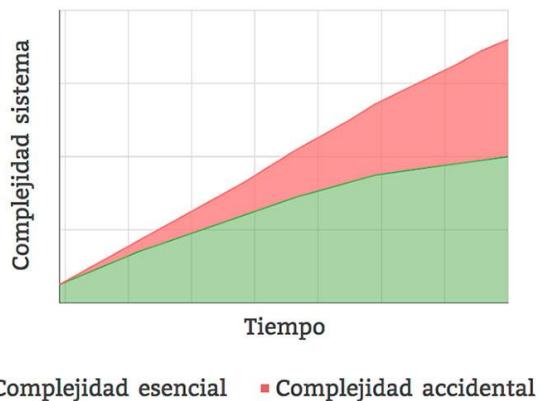
Arquitectura ágil

- Es económicamente factible realizar cambios radicales si la estructura del software separa sus aspectos de forma eficaz y dispone de las pruebas unitarias adecuadas de la arquitectura del sistema.
- Es posible iniciar un proyecto de software con una arquitectura simple pero bien desconectada, y ofrecer historias funcionales de forma rápida, para después aumentar la infraestructura.
- Los estándares facilitan la reutilización de ideas y componentes, reclutar personas con experiencia relevante, encapsulan buenas ideas y conectan componentes. Sin embargo, el proceso de creación de estándares a veces puede llevar demasiado tiempo para que la industria espere, y algunos estándares pierden contacto con las necesidades reales de los adoptantes a los que están destinados a servir. Hay que usar estándares cuando añadan un valor demostrable.

© JMA 2020. All rights reserved

Complejidad

- La complejidad esencial está causada por el problema a resolver y nada puede eliminarla; si los usuarios quieren que un programa haga 30 cosas diferentes, entonces esas 30 cosas son esenciales y el programa debe hacer esas 30 cosas diferentes.
- La complejidad accidental se relaciona con problemas que los desarrolladores crean y pueden solucionar; por ejemplo, los detalles de escribir y optimizar el código o las demoras causadas por el procesamiento por lotes.
- Con el tiempo, la complejidad accidental ha disminuido sustancialmente, los programadores de hoy dedican la mayor parte de su tiempo a abordar la complejidad esencial.



© JMA 2020. All rights reserved

Arquitectura Limpia

- En los últimos años, hemos visto una amplia gama de ideas sobre la arquitectura de los sistemas. Éstos incluyen:
 - Arquitectura Hexagonal (también conocida como Puertos y Adaptadores) por Alistair Cockburn y adoptada por Steve Freeman y Nat Pryce en su libro *Growing Object Oriented Software*
 - Arquitectura de cebolla (Onion Architecture) de Jeffrey Palermo
 - Screaming Architecture de Robert Martin
 - DCI de James Coplien y Trygve Reenskaug.
 - BCE por Ivar Jacobson de su libro *Ingeniería de software orientada a objetos: un enfoque basado en casos de uso*.
- Aunque todas estas arquitecturas varían algo en sus detalles, son muy similares. Todos tienen el mismo objetivo, que es la separación de conceptos. Todos logran esta separación dividiendo el software en capas. Cada uno tiene al menos una capa para las reglas de negocio y otra para las interfaces.

© JMA 2020. All rights reserved

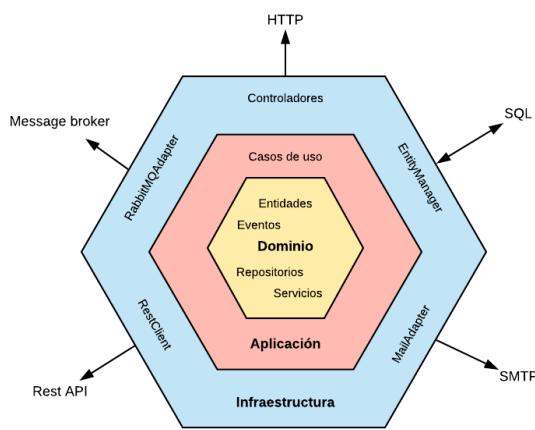
Arquitecturas multicapas

- Cada capa se apoya en la capa subsiguiente (capas horizontales) que depende de las capas debajo de ella, que a su vez dependerá de alguna infraestructura común y servicios públicos.
- El gran inconveniente de esta arquitectura en capas de arriba hacia abajo es el acoplamiento que crea.
- Hay sistemas estrictos (strict layered systems) y relajados (relaxed layered systems) que determinaran el nivel de acoplamiento de las dependencias entre capas.
- Las preocupaciones transversales provocan la aparición de capas verticales.
- Todo esto crea una complejidad accidental innecesaria.



© JMA 2020. All rights reserved

Arquitectura Hexagonal

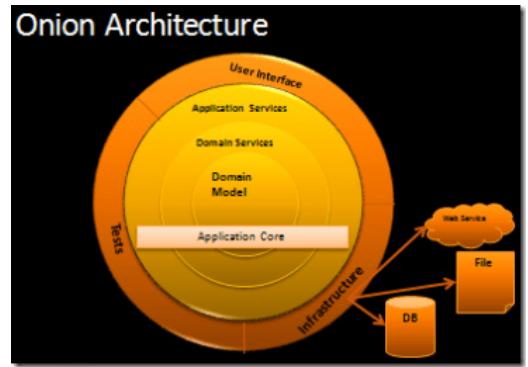


- Propone que nuestro dominio sea el núcleo de las capas y que este no se acople a nada externo. Mediante el principio de inversión de dependencias, en vez de hacer uso explícito, nos acoplamos a contratos (interfaces o puertos) y no a implementaciones concretas.
- También llamada arquitectura de puertos y adaptadores:
 - Puerto: definición de una interfaz pública.
 - Adapter: especialización de un puerto para un contexto concreto.

© JMA 2020. All rights reserved

Onion Architecture

- El término Onion Architecture o Arquitectura cebolla fue acuñada por Jeffrey Palermo, es un patrón arquitectónico basado en capas concéntricas, de fuera a dentro (núcleo), y en mecanismos de inyección de dependencias, para disminuir el acoplamiento entre las capas.



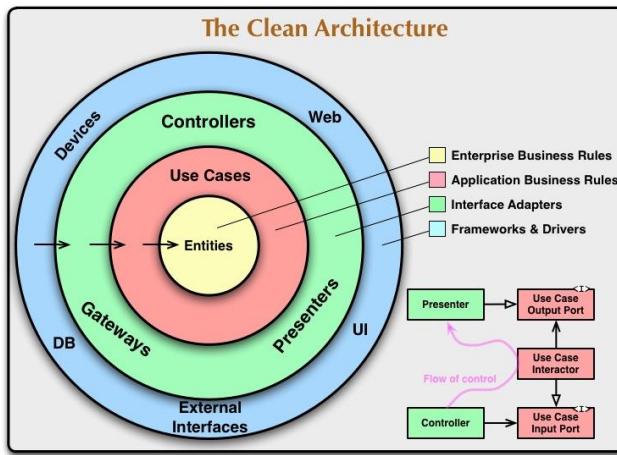
© JMA 2020. All rights reserved

Arquitectura Limpia

- Independiente de los marcos: La arquitectura no depende de la existencia de alguna biblioteca de software cargado de funciones. Esto permite utilizar dichos marcos como herramientas, en lugar de tener que ajustar el sistema a las limitaciones impuestas.
- Comprobable: Las reglas de negocio se pueden probar sin la interfaz de usuario, la base de datos, el servidor web o cualquier otro elemento externo.
- Independiente de la IU: La interfaz de usuario puede cambiar fácilmente, sin cambiar el resto del sistema. Una interfaz de usuario web podría reemplazarse por una interfaz de usuario de consola, por ejemplo, sin cambiar las reglas de negocio.
- Independiente de la base de datos: Puede cambiar Oracle o SQL Server por Mongo, BigTable, CouchDB u otra cosa. Las reglas de negocio no están vinculadas a la base de datos.
- Independiente de cualquier dependencia externa: De hecho, las reglas de negocio simplemente no saben nada sobre el mundo exterior.

© JMA 2020. All rights reserved

Arquitectura Limpia



<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

© JMA 2020. All rights reserved

La regla de la dependencia

- El diagrama presenta una arquitectura basada en “capas de cebolla”, enfoque propuesto por Jeffrey Palermo.
- Los círculos concéntricos representan diferentes áreas del software. En general, cuanto más avanza, mayor nivel se vuelve el software. Los círculos exteriores son mecanismos. Los círculos internos son políticas.
- La regla primordial que hace que esta arquitectura funcione es la regla de dependencia. Esta regla dice que las dependencias del código fuente solo pueden apuntar hacia adentro. Nada en un círculo interno puede saber nada sobre algo en un círculo externo. En particular, el nombre de algo declarado en un círculo exterior no debe ser mencionado por el código en un círculo interior. Eso incluye funciones, clases, variables o cualquier otra entidad de software nombrada.
- Del mismo modo, los formatos de datos utilizados en un círculo exterior no deben ser utilizados por un círculo interior, especialmente si esos formatos son generados por un marco en un círculo exterior. No queremos que nada en un círculo externo impacte en los círculos internos.

© JMA 2020. All rights reserved

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaría que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

© JMA 2020. All rights reserved

Casos de uso

- Esta capa contiene las reglas de negocio específicas de la aplicación. Encapsula e implementa todos los casos de uso del sistema. Éstos orquestan el flujo de datos hacia-desde las entidades y hacen que las entidades usen su lógica de negocio para conseguir el objetivo del caso de uso.
- No esperamos que los cambios en esta capa afecten a las entidades. Tampoco esperamos que esta capa se vea afectada por cambios en las externalidades como la base de datos, la interfaz de usuario o cualquiera de los marcos comunes. Esta capa está aislada de tales preocupaciones.
- Sin embargo, si es esperable que los cambios en el funcionamiento de la aplicación o en las reglas del negocio vayan a afectar a los casos de uso y, por lo tanto, a esta capa. Es decir, si los detalles de un caso de uso cambian, algún código de esta capa se verá afectado y habrá que cambiarlo.

© JMA 2020. All rights reserved

Adaptadores de interfaz

- Esta capa contiene un conjunto de adaptadores que convierten los datos desde el formato más conveniente para el caso de uso y entidades al formato más conveniente o aceptado por elementos externos como la base de datos o la UI. Por ejemplo, esta capa es la que contendrá por completo la arquitectura MVC de una GUI.
- Los presentadores, vistas y controladores pertenecen a esta capa. Los modelos son sólo estructuras que se pasan desde los controladores a los casos de uso y de vuelta desde los casos de uso a los presentadores y/o vistas.
- Del mismo modo, los datos son convertidos en esta capa, desde la forma de las entidades y casos de uso a la forma conveniente para la herramienta de persistencia de datos.
- En este círculo se encuentran también los demás adaptadores encargados de convertir datos obtenidos de una fuente externa, como un servicio externo, al formato interno usado por los casos de uso y las entidades.

© JMA 2020. All rights reserved

Frameworks y drivers

- El círculo más externo se compone generalmente de frameworks y herramientas como la base de datos, el framework web, etc.
- Generalmente en esta capa sólo se escribe código que actúa de “pegamento” con los círculos interiores.
- En esta capa es donde van todos los detalles. La web es un detalle. La base de datos es un detalle. Se mantienen estas cosas afuera, donde no pueden hacer mucho daño.

© JMA 2020. All rights reserved

Cruzando fronteras

- En la parte inferior derecha del diagrama hay un ejemplo de cómo se cruzan las fronteras del círculo. Muestra cómo los controladores y presentadores se comunican con los casos de uso del círculo interno.
- El control del flujo empieza en el controlador, atraviesa el caso de uso y finaliza en el presentador. La dependencia a nivel de código fuente apunta hacia adentro, hacia los casos de uso.
- Se resuelve esta supuesta contradicción usando el principio de inversión de dependencia, es decir, desarrollando interfaces y relaciones de herencia de tal manera que las dependencias de código fuente se oponen al flujo de control en sólo algunos puntos a través de la frontera.
- Considerando que los casos de uso necesitan llamar al presentador, esta llamada no debe ser directa ya que se violaría el principio de la regla de dependencia. Ninguna implementación de un círculo exterior puede ser llamado por un círculo interior. En esta situación el caso de uso llama a una interfaz definida en el círculo interno y hace que el presentador implemente dicha interfaz en el círculo exterior.

© JMA 2020. All rights reserved

Datos que cruzan los límites

- Las estructuras de datos que se pasan a través de las fronteras deben ser simples.
- No se debería pasar entidades o filas de base de datos. Muchos frameworks de base de datos devuelven los datos en el formato de respuesta de una query. Por ejemplo, un RowStructure. Pasar los datos en ese formato violaría la regla de dependencia obligando a un círculo interno saber algo sobre un círculo exterior.
- Las estructuras de datos que tienen cualquier tipo de dependencia con capas exteriores viola la regla de dependencia.
- Los datos que cruzan las fronteras deben ser estructuras de datos simples. Se puede utilizar estructuras básicas u objetos Data Transfer. O los datos pueden ser argumentos en las llamadas a funciones. O bien, almacenar en un diccionario o hash. Cuando se pasan datos a través de una frontera, siempre debe ser en la forma más conveniente para el círculo interior.

© JMA 2020. All rights reserved

Beneficios

- Cumplir con estas simples reglas no es difícil y ahorrará muchos dolores de cabeza en el futuro.
- Al separar el software en capas y cumplir con la regla de dependencia, obliga a que nuestro dominio no esté acoplado a nada externo mediante el uso de interfaces propias del dominio que son implementadas por elementos externos (Alta reutilización: Principio de Responsabilidad Única (SRP) de SOLID).
- Esto creará un sistema que es intrínsecamente comprobable, con todos los beneficios que ello implica (Alta testabilidad: Aplicación del Principio de Inversión de Dependencias (DIP) de SOLID para la interacción del dominio con el resto de elementos)
- Permite estar preparado para cambiar los detalles de implementación, cuando alguna de las partes externas del sistema se vuelve obsoleta, como la base de datos o el marco web, se pueden reemplazar con un mínimo de esfuerzo (Alta tolerancia al cambio: Principio de Abierto/Cerrado (OCP) de SOLID derivado de la aplicación del DIP).
- Permite postergar decisiones importantes.

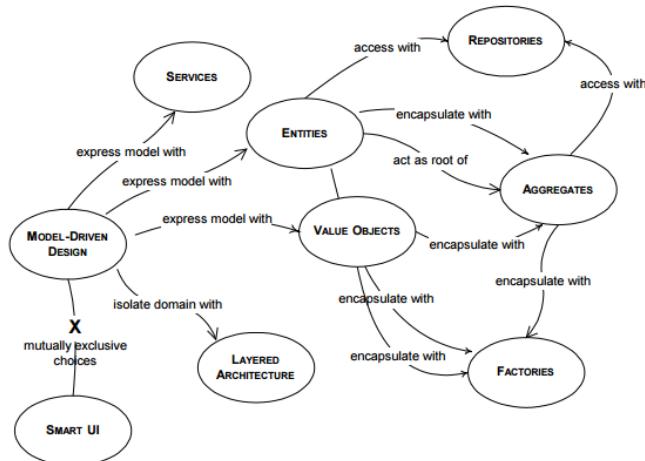
© JMA 2020. All rights reserved

Postergar decisiones importantes

- Una buena arquitectura nos permite postergar decisiones importantes, esto no significa que estemos obligados a hacerlo. Sin embargo, al poder postergarlas tenemos muchísima flexibilidad.
- Cuanto más tarde tomemos una decisión, más conocimiento tendremos del negocio y del entorno técnico:
 - Soluciones más simples/sencillas
 - Soluciones más pequeñas y manejables
 - Mayor productividad, minimiza el hacer cosas que no aporte valor real al final, retrabajos y adaptaciones
- Nuestro objetivo es tener mucho juego de cintura, reaccionar bien y rápido a cualquier cosa y sin echarnos las manos a la cabeza. Cuanto más postponemos, añadimos menos cosas innecesarias, menos llenamos la mochila y es más difícil coger lastre, por lo que podemos viajar ligeros. Como cualquiera sabe: una mudanza se hace fácil si tenemos pocas cosas.

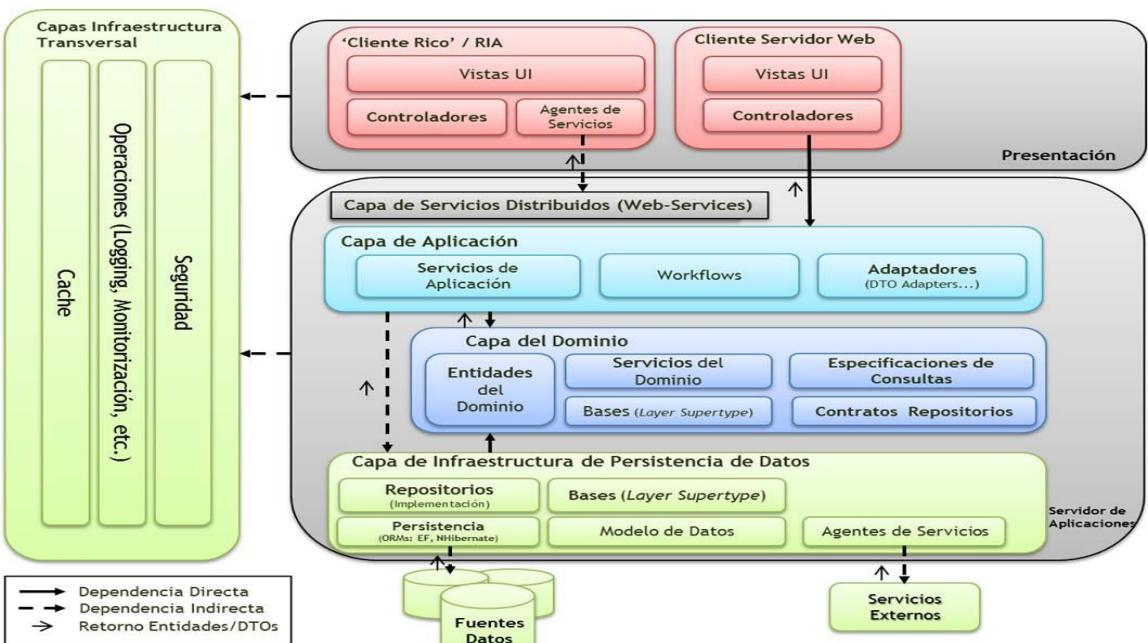
© JMA 2020. All rights reserved

Domain Driven Design



© JMA 2020. All rights reserved

Arquitectura N-Capas con Orientación al Dominio



Capas de DDD

- Interface de usuario (User Interface)
 - Responsable de presentar la información al usuario, interpretar sus acciones y enviarlas a la aplicación.
- Aplicación (Application)
 - Responsable de coordinar todos los elementos de la aplicación. No contiene lógica de negocio ni mantiene el estado de los objetos de negocio. Es responsable de mantener el estado de la aplicación y del flujo de esta.
- Dominio (Domain)
 - Contiene la información sobre el Dominio. Es el núcleo de la parte de la aplicación que contiene las reglas de negocio. Es responsable de mantener el estado de los objetos de negocio. La persistencia de estos objetos se delega en la capa de infraestructura.
- Infraestructura (Infrastructure)
 - Esta capa es la capa de soporte para el resto de capas. Provee la comunicación entre las otras capas, implementa la persistencia de los objetos de negocio y las librerías de soporte para las otras capas (Interface, Comunicación, Almacenamiento, etc..)
- Dado que son capas conceptuales, su implementación puede ser muy variada y en una misma aplicación, tendremos partes o componentes que formen parte de cada una de estas capas.

© JMA 2020. All rights reserved

Entidades

- Las entidades encapsulan los conceptos del negocio. Una entidad puede ser un objeto con métodos o puede ser un conjunto de funciones y estructuras de datos. No importa siempre que las entidades puedan ser utilizadas por muchas aplicaciones diferentes en la empresa.
- Estas entidades son los objetos de dominio de la aplicación. Encapsulan las reglas más generales y de alto nivel.
- Son los menos propensos a cambiar cuando algo externo cambia. Por ejemplo, no esperaría que estos objetos se vieran afectados por un cambio en la navegación de la página o la seguridad.
- Ningún cambio operativo en una aplicación en particular debería afectar la capa de la entidad.

© JMA 2020. All rights reserved

Patrón Agregado (Aggregate)

- Una Agregación es un grupo de entidades asociadas que deben tratarse como una unidad a la hora de manipular sus datos.
- El patrón Agregado es ampliamente utilizado en los modelos de datos basados en Diseños Orientados al Dominio (DDD).
- Proporciona una forma de encapsular nuestras entidades y los accesos y relaciones que se establecen entre las mismas de manera que se simplifique la complejidad del sistema en la medida de lo posible.
- Cada Agregación cuenta con una Entidad Raíz (root) y una Frontera (boundary):
 - La Entidad Raíz es una Entidad contenida en la Agregación de la que colgarán el resto de entidades del agregado y será el único punto de entrada a la Agregación.
 - La Frontera define qué está dentro de la Agregación y qué no.
- La Agregación es la unidad de persistencia, se recupera toda y se almacena toda.

© JMA 2020. All rights reserved

Servicio

- Los servicios representan operaciones, acciones o actividades que no pertenecen conceptualmente a ningún objeto de dominio concreto. Los servicios no tienen ni estado propio ni un significado más allá que la acción que los definen. Se anotan con @Service.
- Podemos dividir los servicios en tres tipos diferentes:
 - Domain services
 - Son responsables del comportamiento más específico del dominio, es decir, realizan acciones que no dependen de la aplicación concreta que estemos desarrollando, sino que pertenecen a la parte más interna del dominio y que podrían tener sentido en otras aplicaciones pertenecientes al mismo dominio.
 - Application services
 - Son responsables del flujo principal de la aplicación, es decir, son los casos de uso de nuestra aplicación. Son la parte visible al exterior del dominio de nuestro sistema, por lo que son el punto de entrada-salida para interactuar con la funcionalidad interna del dominio. Su función es coordinar entidades, value objects, domain services e infrastructure services para llevar a cabo una acción.
 - Infrastructure services
 - Declaran comportamiento que no pertenece realmente al dominio de la aplicación pero que debemos ser capaces de realizar como parte de este.

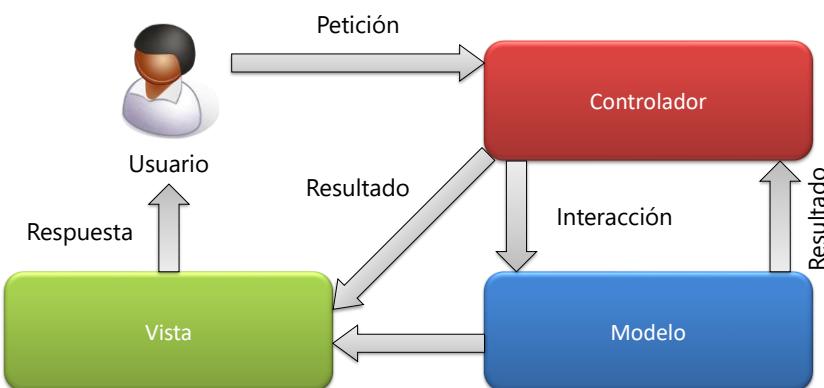
© JMA 2020. All rights reserved

El patrón MVC

- 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, Móvil, ...) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net ...)

© JMA 2020. All rights reserved

El patrón MVC



© JMA 2020. All rights reserved

El patrón MVC



- Representación de los **datos del dominio**
- Lógica de **negocio**
- Mecanismos de **persistencia**



- **Interfaz** de usuario
- Incluye elementos de **interacción**

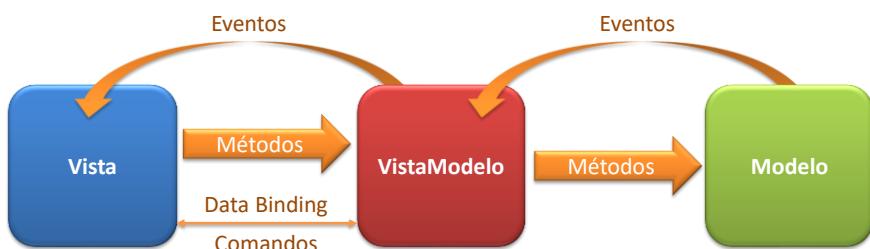


- **Intermediario** entre Modelo y Vista
- **Mapea acciones** de usuario → acciones del Modelo
- **Selecciona** las vistas y les **suministra** información

© JMA 2020. All rights reserved

Model View ViewModel (MVVM)

- El **Modelo** es la entidad que representa el concepto de negocio.
- La **Vista** es la representación gráfica del control o un conjunto de controles que muestran el Modelo de datos en pantalla.
- La **VistaModelo** es la que une todo. Contiene la lógica del interfaz de usuario, los comandos, los eventos y una referencia al Modelo.



© JMA 2020. All rights reserved

¿Cuáles son los beneficios del patrón MVVM?

- Separación de vista / presentación.
- Permite las pruebas unitarias: como la lógica de presentación está separada de la vista, podemos realizar pruebas unitarias sobre la VistaModelo.
- Mejora la reutilización de código.
- Soporte para manejar datos en tiempo de diseño.
- Múltiples vistas: la VistaModelo puede ser presentada en múltiples vistas, dependiendo del rol del usuario por ejemplo.

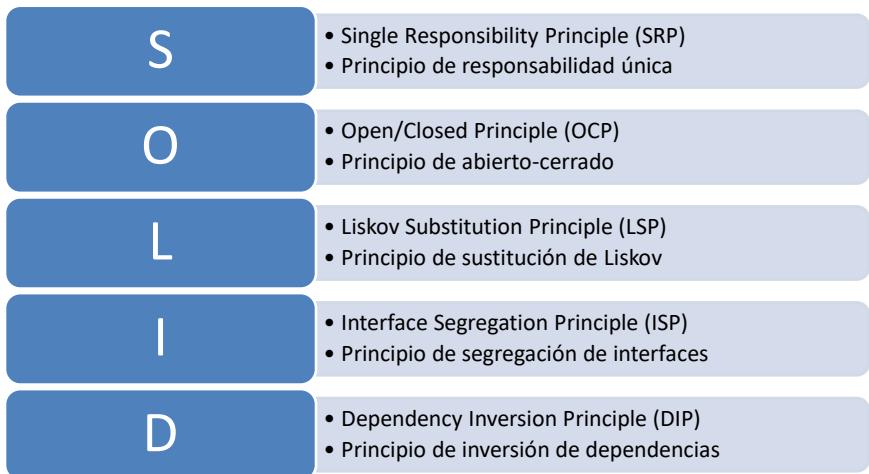
© JMA 2020. All rights reserved

S.O.L.I.D.

- SOLID es el acrónimo que acuñó Michael Feathers, basándose en los 5 principios de la programación orientada a objetos que Robert C. Martin había recopilado en el año 2000 en su artículo “Design Principles and Design Patterns”.
- Los objetivos de estos 5 principios a la hora de escribir código son:
 - Crear un software eficaz: que cumpla con su cometido y que sea robusto y estable.
 - Escribir un código limpio y flexible ante los cambios: que se pueda modificar fácilmente según necesidad, que sea reutilizable y mantenible.
 - Permitir la escalabilidad: que acepte ser ampliado con nuevas funcionalidades de manera ágil.
- La aplicación de los principios SOLID está muy relacionada con la comprensión y el uso de patrones de diseño, que permitirán minimizar el acoplamiento (grado de interdependencia que tienen dos unidades de software entre sí) y maximizar la cohesión (grado en que elementos diferentes de un sistema permanecen unidos para alcanzar un mejor resultado que si trabajaran por separado).

© JMA 2020. All rights reserved

S.O.L.I.D.



© JMA 2020. All rights reserved

S.O.L.I.D.

• Principio de Responsabilidad Única

“A class should have one, and only one, reason to change.”

- La S del acrónimo del que hablamos hoy se refiere a Single Responsibility Principle (SRP). Según este principio “una clase debería tener una, y solo una, razón para cambiar”. Es esto, precisamente, “razón para cambiar”, lo que Robert C. Martin identifica como “responsabilidad”.
- El principio de Responsabilidad Única es el más importante y fundamental de SOLID, muy sencillo de explicar, pero el más difícil de seguir en la práctica.
- El propio Bob resume cómo hacerlo: “Reúne las cosas que cambian por las mismas razones. Separa aquellas que cambian por razones diferentes”.

© JMA 2020. All rights reserved

S.O.L.I.D.

- **Principio de Abierto/Cerrado**

“You should be able to extend a classes behavior, without modifying it.”

- El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.
- El principio Open/Closed se suele resolver utilizando polimorfismo.
- Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías, frameworks, microservicios.

© JMA 2020. All rights reserved

S.O.L.I.D.

- **Principio de Sustitución de Liskov**

“Base classes must be substitutable for their derived classes.”

- La L de SOLID alude al apellido de quien lo creó, Barbara Liskov, y dice que “las clases base deben poder sustituirse por sus clases derivadas”.
- Esto significa que los objetos deben poder ser reemplazados por instancias de sus subtipos sin alterar el correcto funcionamiento del sistema o lo que es lo mismo: si en un programa utilizamos cierta clase, deberíamos poder usar cualquiera de sus subclases sin interferir en la funcionalidad del programa.
- Según Robert C. Martin incumplir el Liskov Substitution Principle (LSP) implica violar también el principio de Abierto/Cerrado.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Segregación de la Interfaz

“Make fine grained interfaces that are client specific.”

- En el cuarto principio de SOLID, el tío Bob sugiere: “Haz interfaces que sean específicas para un tipo de cliente”, es decir, para una finalidad concreta.
- En este sentido, según el Interface Segregation Principle (ISP), es preferible contar con muchas interfaces especializadas que definan unos pocos métodos que tener una interface generalista que fuerce a implementar muchos métodos a los que no se dará uso.

© JMA 2020. All rights reserved

S.O.L.I.D.

- Principio de Inversión de Dependencias

“Depend on abstractions, not on concretions.”

- Llegamos al último principio: “Depende de abstracciones, no de clases concretas”.
- Así, Robert C. Martin recomienda:
 - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
 - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
- El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases. Las abstracciones se logran mediante los interfaces.

© JMA 2020. All rights reserved

Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instanciación en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.

© JMA 2020. All rights reserved

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto con unas técnicas de programación:
 - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
 - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir. Técnicas de implementación:
 - Service Locator: es un componente (contenedor) que contiene referencias a los servicios y encapsula la lógica que los localiza dichos servicios.
 - Inyección de dependencias.

© JMA 2020. All rights reserved

Inyección de Dependencias

- Las dependencias son expresadas en términos de interfaces en lugar de clases concretas y se resuelven dinámicamente en tiempo de ejecución.
- La Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java o .NET, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del Reflection y crear una instancia de dicha clase inyectándole sus dependencias.

© JMA 2020. All rights reserved

<http://spring.io>

SPRING CON SPRING BOOT

© JMA 2020. All rights reserved

Spring

- 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 pueden ser .NET, Silverlight, Windows Phone, etc.)
- 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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Proyectos



© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Spring Boot

- 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 si centre en el desarrollo de la solución, olvidándose por completo de la compleja configuración que actualmente tiene Spring Core para poder funcionar.
 - 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 nada.
 - 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.
 - 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.
 - 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 LiveReload y Auto Restart, funcionan en su IDE favorito: Spring Tool Suite, IntelliJ IDEA y NetBeans.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Spring Tools

- <https://spring.io/tools>
- 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
- Spring Tools 4 es la próxima generación de herramientas Spring para los entorno sde 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

© JMA 2020. All rights reserved

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)

© JMA 2020. All rights reserved

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
    }
}
```

© JMA 2020. All rights reserved

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**

© JMA 2020. All rights reserved

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 settings
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%
logging.level.org.hibernate.SQL=debug
```
- Repetir con src/test/resources/application.properties
- Eclipse: Run Configurations → Arguments → VM Arguments: -DPORT=8888

© JMA 2020. All rights reserved

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 maa/sakila:latest
- 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 -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 rabbitmq:3-management

© JMA 2020. All rights reserved

<https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.htm>

IOC CON SPRING CORE

© JMA 2020. All rights reserved

Construcción vs Uso

- Los sistemas de software deben separar el proceso de inicio, en el que se instancian los objetos de la aplicación y se conectan las dependencias, de la lógica de ejecución que utilizan las instancias. La separación de conceptos es una de las técnicas de diseño más antiguas e importantes.
- El mecanismo mas potente es la Inyección de Dependencias, la aplicación de la Inversión de Control a la gestión de dependencias. Delega la instancia en un mecanismo alternativo, que permite la personalización, responsable de devolver instancias plenamente formadas con todas las dependencias establecidas. Permite la creación de instancias bajo demanda de forma transparente al consumidor.

© JMA 2020. All rights reserved

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es tanto un concepto como unas técnicas de programación:
 - en las que el flujo de ejecución de un programa se invierte respecto a los métodos de programación tradicionales,
 - en los que la interacción se expresa de forma imperativa haciendo llamadas a procedimientos (procedure calls) o funciones.
- Tradicionalmente el programador especifica la secuencia de decisiones y procedimientos que pueden darse durante el ciclo de vida de un programa mediante llamadas a funciones.
- En su lugar, en la inversión de control se especifican respuestas deseadas a sucesos o solicitudes de datos concretas, dejando que algún tipo de entidad o arquitectura externa lleve a cabo las acciones de control que se requieran en el orden necesario y para el conjunto de sucesos que tengan que ocurrir.

© JMA 2020. All rights reserved

Inyección de Dependencias

- Inyección de Dependencias (en inglés Dependency Injection, DI) es un patrón de arquitectura orientado a objetos, en el que se inyectan objetos a una clase en lugar de ser la propia clase quien cree el objeto, básicamente recomienda que las dependencias de una clase no sean creadas desde el propio objeto, sino que sean configuradas desde fuera de la clase.
- La inyección de dependencias (DI) procede del patrón de diseño más general que es la Inversión de Control (IoC).
- Al aplicar este patrón se consigue que las clases sean independientes unas de otras e incrementando la reutilización y la extensibilidad de la aplicación, además de facilitar las pruebas unitarias de las mismas.
- Desde el punto de vista de Java, un diseño basado en DI puede implementarse mediante el lenguaje estándar, dado que una clase puede leer las dependencias de otra clase por medio del API Reflection de Java y crear una instancia de dicha clase inyectándole sus dependencias.

© JMA 2020. All rights reserved

Inyección

- La Inyección de Dependencias proporciona:
 - Código más limpio
 - Desacoplamiento más eficaz, pues los objetos no deben de conocer donde están sus dependencias ni cuales son.
 - Facilidad en las pruebas unitaria e integración

© JMA 2020. All rights reserved

Spring Core IoC

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita injectar unos objetos sobre otros.
- Para ello, los objetos deberán ser simplemente JavaBeans.
- La inyección de dependencias será bien por constructor o bien por métodos setter.
- La configuración podrá realizarse bien por anotaciones Java o mediante un fichero XML (XMLBeanFactory).
- Para la gestión de los objetos tendrá la clase (BeanFactory).
- Todos los objetos serán creados como singlettons sino se especifica lo contrario.

© JMA 2020. All rights reserved

Modulo de dependencias

- Se crea el fichero de configuración applicationContext.xml y se guarda en el directorio src/META-INF.

```
<beans xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns="http://www.springframework.org/schema/beans"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">
    <context:component-scan base-package="es.miEspacio.ioc.services">
        </context:component-scan>
</beans>
```

© JMA 2020. All rights reserved

Beans

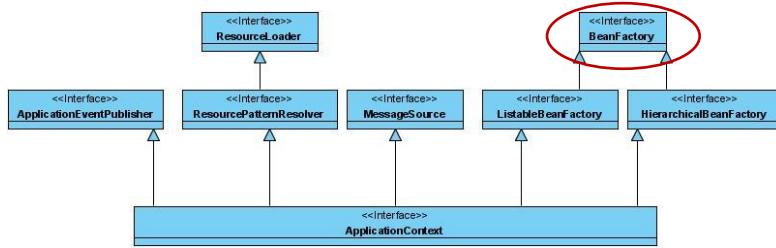
- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories, JMS Queues), etc.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">
    <!-- services -->
    <bean id="petStore" class="com.samples.PetStoreServiceImpl">
        <property name="accountDao" ref="accountDao"/>
        <property name="itemDao" ref="itemDao"/>
        <!-- additional collaborators and configuration for this bean go here -->
    </bean>
    <!-- more bean definitions for services go here -->
</beans>
```

© JMA 2020. All rights reserved

Bean factory

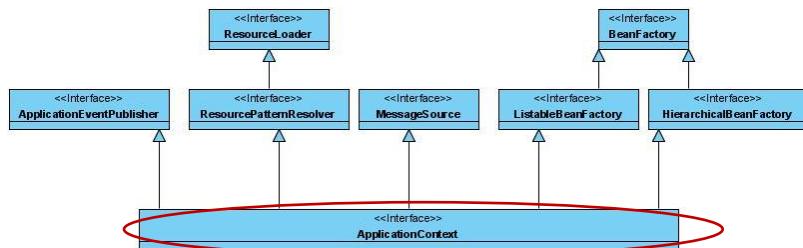
- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory) implementan el interface BeanFactory y algunos sub-interfaces para ampliar funcionalidades



© JMA 2020. All rights reserved

Application Context

- Spring también soporta una "fábrica de beans" algo más avanzado, llamado contexto de aplicación.
- Application Context, es una especificación de Bean Factory que implementa la interface ApplicationContext.
- En general, cualquier cosa que un Bean Factory puede hacer, un contexto de aplicación también lo puede hacer.



© JMA 2020. All rights reserved

Uso de la inyección de dependencias

- Se crea un inyector partiendo de un módulo de dependencias.
- Se solicita al inyector las instancias para que resuelva las dependencias.

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("META-INF/applicationContext.xml");
    // AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
    BeanFactory factory = context;
    Client client = (Client )factory.getBean("ID_Cliente");
    client.go();
}
• Muestra:
– Este es un servicio...
```

© JMA 2020. All rights reserved

Anotaciones IoC

- Autodescubrimiento
 - @Component
 - @Repository
 - @Service
 - @Controller
 - @Scope
- Personalización
 - @Configuration
 - @Bean
- Inyección
 - @Autowired (@Inject)
 - @Qualifier (@Named)
 - @Value
 - @PropertySource
 - @Required
 - @Resource
- Otras
 - @PostConstruct
 - @PreDestroy

© JMA 2020. All rights reserved

Esterotipos

- Spring define un conjunto de anotaciones core que categorizan cada uno de los componentes asociándoles una responsabilidad concreta.
 - `@Component`: Es el estereotipo general y permite anotar un bean para que Spring lo considere uno de sus objetos.
 - `@Repository`: Es el estereotipo que se encarga de dar de alta un bean para que implemente el patrón repositorio que es el encargado de almacenar datos en una base de datos o repositorio de información que se necesite. Al marcar el bean con esta anotación Spring aporta servicios transversales como conversión de tipos de excepciones.
 - `@Service` : Este estereotipo se encarga de gestionar las operaciones de negocio más importantes a nivel de la aplicación y aglutina llamadas a varios repositorios de forma simultánea. Su tarea fundamental es la de agregador.
 - `@Controller` : El último de los estereotipos que es el que realiza las tareas de controlador y gestión de la comunicación entre el usuario y el aplicativo. Para ello se apoya habitualmente en algún motor de plantillas o librería de etiquetas que facilitan la creación de páginas.
 - `@RestController` que es una especialización de controller que contiene las anotaciones `@Controller` y `@ResponseBody` (escribe directamente en el cuerpo de la respuesta en lugar de la vista).

© JMA 2020. All rights reserved

Alcance

- Un aspecto importante del ciclo de vida de los Beans es si el contenedor creara una única instancia o tantas como ámbitos sean necesarios.
 - `prototype`: No reutiliza instancias, genera siempre una nueva instancia.
`@Scope("prototype")`
 - `singleton`: (Predeterminado) Instancia única para todo el contenedor Spring IoC.
`@Scope("singleton") @Singleton`
 - Adicionalmente, en el contexto de un Spring Web ApplicationContext:
`@RequestScope @SessionScope @ApplicationScope`
 - `request`: Instancia única para el ciclo de vida de una sola solicitud HTTP. Cada solicitud HTTP tiene su propia instancia única.
 - `session`: Instancia única para el ciclo de vida de cada HTTP Session.
 - `application`: Instancia única para el ciclo de vida de un ServletContext.
 - `websocket`: Instancia única para el ciclo de vida de un WebSocket.

© JMA 2020. All rights reserved

Inyección

- La inyección se realiza con la anotación `@Autowired`:
 - En atributos:
`@Autowired
private MyBeans myBeans;`
 - En propiedades (setter):
`@Autowired
public void setMyBeans(MyBeans value) { ... }`
 - En constructores
- Por defecto la inyección es obligatoria, se puede marcar como opcional en cuyo caso si no encuentra el Bean inyectará un null.
`@Autowired(required=false)
private MyBeans myBeans;`
- Se puede completar `@Autowired` con la anotación `@Lazy` para inyectar un proxy de resolución lenta.

© JMA 2020. All rights reserved

Cualificación

- Con `@Qualifier (@Named)` se pueden agrupar o cualificar los beans asociándoles un nombre:
`public interface MyInterface { ... }

@Component
@Qualifier("old")
public class MyInterfaceImpl implements MyInterface { ... }

@Qualifier("new")
@Component
public class MyNewInterfaceImpl implements MyInterface { ... }

@Autowired(required=false)
@Qualifier("new")
private MyInterface srv;`

© JMA 2020. All rights reserved

Perfiles

- Un perfil es un grupo lógico con nombre de definiciones de beans que se registrarán en el contenedor solo si el perfil dado está activo, proporcionan un mecanismo en el contenedor central que permite el registro de diferentes beans en diferentes entornos.

```
@Service
@Profile("default")
public class ServicioImpl { ... }
```

```
@Service
@Profile("test")
public class ServicioMockImpl { ... }
```

```
@Bean
@Profile("test")
Servicio getServicio() { ... }
```

- Hay indicarle a Spring qué perfil está activo, o por defecto utilizará el perfil "default", con la anotación `@ActiveProfiles("test")`, el parámetro `-Dspring.profiles.active = "test"` o por código:

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext();
ctx.getEnvironment().setActiveProfiles("test");
ctx.refresh();
```

© JMA 2020. All rights reserved

Acceso a ficheros de propiedades

- Localización (fichero .properties, .yml, .xml):
 - Por defecto: `src/main/resources/application.properties`
 - En la carpeta de recursos `src/main/resources`:
`@PropertySource("classpath:my.properties")`
 - En un fichero local:
`@PropertySource("file://c:/cng/my.properties")`
 - En una URL:
`@PropertySource("http://myserver/application.properties")`
- Acceso directo:
`@Value("${spring.datasource.username}") private String name;`
- Acceso a través del entorno:
`@Autowired private Environment env;
env.getProperty("spring.datasource.username")`

© JMA 2020. All rights reserved

Ciclo de Vida

- Con la inyección el proceso de creación y destrucción de las instancias de los beans es administrada por el contendor.
- Para poder intervenir en el ciclo para controlar la creación y destrucción de las instancias se puede:
 - Implementar las interfaces InitializingBean y DisposableBean de devoluciones de llamada
 - Sobrescribir los métodos init() y destroy()
 - Anotar los métodos con @PostConstruct y @PreDestroy.
- Se pueden combinar estos mecanismos para controlar un bean dado.

© JMA 2020. All rights reserved

Configuración por código

- Hay que crear una (o varias) clase anotada con @Configuration que contendrá un método por cada clase/interfaz (sin estereotipo) que se quiera tratar como un Bean inyectable.
- El método irá anotado con @Bean, se debería llamar como la clase en notación Camel y devolver del tipo de la clase la instancia ya creada. Adicionalmente se puede anotar con @Scope y con @Qualifier.

```
public class MyBean { ... }

@Configuration
public class MyConfig {
    @Bean
    @Scope("prototype")
    public MyBean myBean() { ... }
```

© JMA 2020. All rights reserved

Patrón: Doble herencia

- Se crea el interfaz con la funcionalidad deseada:

```
public interface Service {
    public void go();
}
```
- Se implementa la interfaz en una clase (por convenio se usa el sufijo Impl):

```
import org.springframework.stereotype.Service;
@Service
@Singleton
public class ServiceImpl implements Service {
    public void go() {
        System.out.println("Este es un servicio...");
    }
}
```

© JMA 2020. All rights reserved

Cliente

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("ID_Cliente")
public class Client {
    private final Service service;

    @Autowired
    public void setService(Service service){
        this.service = service;
    }

    public void go(){
        service.go();
    }
}

@Autowired establece que deben resolverse los parámetros mediante DI.
```

© JMA 2020. All rights reserved

Anotaciones estándar JSR-330

- A partir de Spring 3.0, Spring ofrece soporte para las anotaciones estándar JSR-330 (inyección de dependencia). Esas anotaciones se escanean de la misma manera que las anotaciones de Spring.
- Cuando trabaje con anotaciones estándar, hay que tener en cuenta que algunas características importantes no están disponibles.

Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

© JMA 2020. All rights reserved

GESTIÓN DE DATOS

© JMA 2020. All rights reserved

Spring Data

- Spring Framework ya proporcionaba soporte para JDBC, Hibernate, JPA o JDO, simplificando la implementación de la capa de acceso a datos, unificando la configuración y creando una jerarquía de excepciones común para todas ellas.
- Spring Data es un proyecto (subproyectos) de SpringSource cuyo propósito es unificar y facilitar el acceso a distintos tipos de tecnologías de persistencia, tanto a bases de datos relacionales como a las del tipo NoSQL.
- Spring Data viene a cubrir el soporte necesario para distintas tecnologías de bases de datos NoSQL integrándolas con las tecnologías de acceso a datos tradicionales, simplificando el trabajo a la hora de crear las implementaciones concretas.
- Con cada tipo de tecnología de persistencia, los DAOs (Data Access Objects) ofrecen las funcionalidades típicas de CRUD para objetos de dominio propios, métodos de búsqueda, ordenación y paginación.
- Spring Data proporciona interfaces genéricas para estos aspectos (CrudRepository, PagingAndSortingRepository) e implementaciones específicas para cada tipo de tecnología de persistencia.

© JMA 2020. All rights reserved

Entidades de Dominio

- Una entidad es cualquier objeto del dominio que mantiene un estado y comportamiento más allá de la ejecución de la aplicación y que necesita ser distinguido de otro que tenga las mismas propiedades y comportamientos.
- Es un tipo de clase dedicada a representar un modelo de dominio persistente que:
 - Debe ser pública (no puede ser estar anidada ni final o tener miembros finales)
 - Deben tener un constructor público sin ningún tipo de argumentos.
 - Para cada propiedad que queramos persistir debe haber un método get/set asociado.
 - Debe tener una clave primaria
 - Debería sobrescribir los métodos equals y hashCode
 - Debería implementar el interfaz Serializable para utilizar de forma remota

© JMA 2020. All rights reserved

Anotaciones JPA

Anotación	Descripción
@Entity	<ul style="list-style-type: none"> - Se aplica a la clase. - Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	<ul style="list-style-type: none"> - Se aplica a la clase e indica el nombre de la tabla de la base de datos donde se persistirá la clase. - Es opcional si el nombre de la clase coincide con el de la tabla.
@Id	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica que este atributo es la clave primaria.
@Column(name="Id")	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica el nombre de la columna de la base de datos en la que se persistirá la propiedad. - Es opcional si el nombre de la propiedad Java coincide con el de la columna de la base de datos.
@Column(...)	<ul style="list-style-type: none"> - name: nombre - length: longitud - precision: número total de dígitos - scale: número de dígitos decimales - unique: restricción valor único - nullable: restricción valor obligatorio - insertable: es insertable - updatable: es modificable
@Transient	<ul style="list-style-type: none"> - Se aplica a una propiedad Java e indica que este atributo no es persistente

© JMA 2020. All rights reserved

Anotaciones JPA

Anotación	Descripción
@Temporal	<ul style="list-style-type: none"> - Especifica el componente temporal a persistir en los tipos Date y Calendar. (TemporalType: DATE, TIME, TIMESTAMP)
@Lob	<ul style="list-style-type: none"> - Aplicado sobre una propiedad, indica que es un objeto grande (Large OBject)..
@Enumerated	<ul style="list-style-type: none"> - Indica que los valores de la propiedad van a estar dentro del rango de un objeto enumerador.
@Embeddable	<ul style="list-style-type: none"> - Define una clase cuyas instancias se almacenan como parte intrínseca de una entidad propietaria (columnas) y comparten la identidad de la entidad.
@Embedded	<ul style="list-style-type: none"> - Indica que la propiedad es de tipo Embeddable
@EmbeddedId	<ul style="list-style-type: none"> - Mapeada una clave primaria compuesta que es una clase incrustable.
@GeneratedValue	<ul style="list-style-type: none"> - Proporciona la especificación de estrategias de generación para los valores de claves primarias. <ul style="list-style-type: none"> - AUTO: Indica que el proveedor de persistencia debe elegir una estrategia adecuada para la base de datos en particular. - IDENTITY: Indica que el proveedor de persistencia debe asignar claves primarias mediante una columna de identidad de base de datos. - SEQUENCE: Indica que el proveedor de persistencia debe asignar claves primarias usando una secuencia de base de datos. - TABLE: Indica que el proveedor de persistencia debe asignar claves primarias mediante una tabla de base de datos subyacente para garantizar la exclusividad.

© JMA 2020. All rights reserved

Asociaciones

- Uno a uno (Unidireccional)
 - En la entidad fuerte se anota la propiedad con la referencia de la entidad.
 - `@OneToOne(cascade=CascadeType.ALL)`:
 - Esta anotación indica la relación uno a uno de las 2 tablas.
 - `@PrimaryKeyJoinColumn`:
 - Indicamos que la relación entre las dos tablas se realiza mediante la clave primaria.
- Uno a uno (Bidireccional)
 - Las dos entidades cuentan con una propiedad con la referencia a la otra entidad.

© JMA 2020. All rights reserved

Asociaciones

- Uno a Muchos
 - En Uno
 - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos:
 - List: Ordenada con repetidos
 - Set: Desordenada sin repetidos
 - `@OneToMany(mappedBy="propEnMuchos",cascade= CascadeType.ALL)`
 - mappedBy: contendrá el nombre de la propiedad en la entidad muchos con la referencia a la entidad uno.
 - `@IndexColumn (name="idx")`
 - Opcional. Nombre de la columna que en la tabla muchos para el orden dentro de la Lista.
 - En Muchos
 - Dispone de una propiedad con la referencia de la entidad uno.
 - `@ManyToOne`
 - Esta anotación indica la relación de Muchos a uno
 - `@JoinColumn (name="idFK")`
 - Indicaremos el nombre de la columna que en la tabla muchos contiene la clave ajena a la tabla uno.

© JMA 2020. All rights reserved

Asociaciones

- Muchos a muchos (Unidireccional)
 - Dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
 - `@ManyToMany(cascade=CascadeType.ALL)`:
 - Esta anotación indica la relación muchos a muchos de las 2 tablas.
- Muchos a muchos(Bidireccional)
 - La segunda entidad también dispone de una propiedad de tipo colección que contiene las referencias de las entidades muchos.
 - `@ManyToMany(mappedBy="propEnOtroMuchos")`:
 - `mappedBy`: Propiedad con la colección en la otra entidad para preservar la sincronicidad entre ambos lados

© JMA 2020. All rights reserved

Cascada

- El atributo cascade se utiliza en los mapeos de las asociaciones para indicar cuando se debe propagar la acción en una instancia hacia la instancias relacionadas mediante la asociación.
- Enumeración de tipo CascadeType:
 - ALL = {PERSIST, MERGE, REMOVE, REFRESH, DETACH}
 - DETACH (Separar)
 - MERGE (Modificar)
 - PERSIST (Crear)
 - REFRESH (Releer)
 - REMOVE (Borrar)
 - NONE


```
@OneToMany(mappedBy="profesor", cascade = CascadeType.ALL, orphanRemoval = true)
private List<AsignaturaProfesor> asignaturas = new ArrayList<>();
```
- Acepta múltiples valores:
 - `@OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})`

© JMA 2020. All rights reserved

Mapeo de Herencia

- Tabla por jerarquía de clases
 - Padre:
 - @Table("Account")
 - @Inheritance(strategy = InheritanceType.SINGLE_TABLE)
 - @DiscriminatorColumn(name="PAYMENT_TYPE")
 - Hija:
 - @DiscriminatorValue(value = "Debit")
- Tabla por subclases
 - Padre:
 - @Table("Account")
 - @Inheritance(strategy = InheritanceType.JOINED)
 - Hija:
 - @Table("DebitAccount")
 - @PrimaryKeyJoinColumn(name = "account_id")
- Tabla por clase concreta
 - Padre:
 - @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
 - Hija:
 - @Table("DebitAccount")

© JMA 2020. All rights reserved

Repositorio

- Un repositorio es una clase que actúa de mediador entre el dominio de la aplicación y los datos que le dan persistencia.
- Su objetivo es abstraer y encapsular todos los accesos a la fuente de datos.
- Oculta completamente los detalles de implementación de la fuente de datos a sus clientes.
- El interfaz expuesto por el repositorio no cambia aunque cambie la implementación de la fuente de datos subyacente (diferentes esquemas de almacenamiento).
- Se crea un repositorio por cada entidad de dominio que ofrece los métodos CRUD (Create-Read-Update-Delete), de búsqueda, ordenación y paginación.

© JMA 2020. All rights reserved

Repository

- Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para las entidades se simplifica porque solo vamos a necesitar un interfaz que extienda uno de los siguientes interfaces:
 - CrudRepository<T, ID>
 - count, delete, deleteAll, deleteById, existsById, findAll, findAllById, findById, save, saveAll
 - PagingAndSortingRepository<T, ID>
 - findAll(Pageable pageable), findAll(Sort sort)
 - JpaRepository<T, ID>
 - deleteAllInBatch, deleteInBatch, flush, getOne, saveAll, saveAndFlush
 - MongoRepository<T, ID>
 - findAll, insert, saveAll
- En el proceso de inyección Spring implementa la interfaz antes de inyectarla:


```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {}  
@Autowired  
private ProfesorRepository repository;
```

© JMA 2020. All rights reserved

Repository

- El interfaz puede ser ampliado con nuevos métodos que serán implementados por Spring:
 - Derivando la consulta del nombre del método directamente.
 - Mediante el uso de una consulta definida manualmente.
- La implementación se realizará mediante la decodificación del nombre del método, dispone de una sintaxis específica para crear dichos nombres:

```
List<Profesor> findByNombreStartingWiths(String nombre);  
List<Profesor> findByApellido1AndApellido2OrderByEdadDesc( String apellido1, String  
apellido2);  
List<Profesor> findByTipoIn(Collection<Integer> tipos);  
  
int deleteByEdadGreater Than(int valor);
```

© JMA 2020. All rights reserved

Repository

- Prefijo consulta derivada:
 - find (read, query, get), count, delete
- Opcionalmente, limitar los resultados de la consulta:
 - Distinct, TopNumFilas y FirstNumFilas
- Expresión de propiedad: ByPropiedad
 - Operador (Between, LessThan, GreaterThan, Like, ...) por defecto equal.
 - Se pueden concatenar varias con And y Or
 - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, OrderByPropiedadAsc para ordenar,
 - se puede sustituir Asc por Desc, admite varias expresiones de ordenación.
- Parámetros:
 - un parámetro por cada operador que requiera valor y debe ser del tipo apropiado
- Parámetros opcionales:
 - Sort → Sort.by("nombre", "apellidos").descending()
 - Pageable → PageRequest.of(0, 10, Sort.by("nombre"))

© JMA 2020. All rights reserved

Repository

Palabra clave	Muestra	Fragmento de JPQL
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname, findByFirstnameIs, findByFirstnameEquals	... where x.firstname = ?1
Between	findByStartDateBetween	... where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	... where x.age <= ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
GreaterThanOrEqual	findByAgeGreaterThanOrEqual	... where x.age >= ?1

© JMA 2020. All rights reserved

Repositorio

Palabra clave	Muestra	Fragmento de JPQL
After	findByStartDateAfter	... where x.startDate > ?1
Before	findByStartDateBefore	... where x.startDate < ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	... where x.firstname like ?1 (parámetro enlazado con % anexado)
EndingWith	findByFirstnameEndingWith	... where x.firstname like ?1 (parámetro enlazado con % antepuesto)

© JMA 2020. All rights reserved

Repositorio

P. Clave	Muestra	Fragmento de JPQL
Containing	findByFirstnameContaining	... where x.firstname like ?1 (parámetro enlazado entre %)
OrderBy	findByAgeOrderByLastnameDesc	... where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	... where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	... where x.age not in ?1
True	findByActiveTrue()	... where x.active = true
False	findByActiveFalse()	... where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	... where UPPER(x.firstname) = UPPER(?1)

© JMA 2020. All rights reserved

Repository

- Valor de retorno de consultas síncronas:
 - find, read, query, get:
 - List<Entidad>
 - Stream<Entidad>
 - Optional<T>
 - count, delete:
 - long
- Valor de retorno de consultas asíncronas (deben ir anotadas con @Async):
 - Future<Entidad>
 - CompletableFuture<Entidad>
 - ListenableFuture<Entidad>

© JMA 2020. All rights reserved

Repository

- Mediante consultas JPQL:


```
@Query("from Profesor p where p.edad > 67")
List<Profesor> findJubilados();

@Modifying
@Query("delete from Profesor p where p.edad > 67")
List<Profesor> deleteJubilados();
```
- Mediante consultas SQL nativas:


```
@Query("select * from Profesores p where p.edad between ?1 and ?2",
nativeQuery=true)
List<Profesor> findActivos(int inicial, int final);
```

© JMA 2020. All rights reserved

Transacciones

- Por defecto, los métodos CRUD en las instancias del repositorio son transaccionales. Para las operaciones de lectura, el indicador `readOnly` de configuración de transacción se establece en `true` para optimizar el proceso. Todos los demás se configuran con un plano `@Transactional` para que se aplique la configuración de transacción predeterminada.
- Cuando se van a realizar varias llamadas al repositorio o a varios repositorios se puede anotar con `@Transactional` el método para que todas las operaciones se encuentren dentro de la misma transacción.

```
@Transactional
public void create(Pago pago) { ... }
```
- Para que los métodos de consulta sean transaccionales:

```
@Override
@Transactional(readOnly = false)
public List<User> findAll();
```

© JMA 2020. All rights reserved

Validaciones

- Desde la versión 3, Spring ha simplificado y potenciado en gran medida la validación de datos, gracias a la adopción de la especificación JSR 303. Este API permite validar los datos de manera declarativa, con el uso de anotaciones. Esto nos facilita la validación de los datos enviados antes de llegar al controlador REST. (**Dependencia: Starter I/O > Validation**)
- Las anotaciones se pueden establecer a nivel de clase, atributo y parámetro de método.
- Se puede exigir la validez mediante la anotación `@Valid` en el elemento a validar.

```
public ResponseEntity<Object> create(@Valid @RequestBody Persona item)
```
- Para realizar la validación manualmente:

```
@Autowired // Validation.buildDefaultValidatorFactory().getValidator()
private Validator validator;
Set<ConstraintViolation<Persona>> constraintViolations = validator.validate( persona );
Set<ConstraintViolation<Persona>> constraintViolations = validator.validateProperty( persona,
    "nombre" );
```

© JMA 2020. All rights reserved

Validaciones (JSR 303)

- @Null : Comprueba que el valor anotado es null
- @NotNull : Comprueba que el valor anotado no sea null
- @NotEmpty : Comprueba si el elemento anotado no es nulo ni está vacío
- @NotBlank : Comprueba que la secuencia de caracteres anotados no sea nula y que la longitud recortada sea mayor que 0. La diferencia @NotEmpty es que esta restricción solo se puede aplicar en secuencias de caracteres y que los espacios en blanco finales se ignoran.
- @AssertFalse : Comprueba que el elemento anotado es falso.
- @AssertTrue : Comprueba que el elemento anotado es verdadero

© JMA 2020. All rights reserved

Validaciones (JSR 303)

- @Max(value=) : Comprueba si el valor anotado es menor o igual que el máximo especificado
- @Min(value=) : Comprueba si el valor anotado es mayor o igual que el mínimo especificado
- @Negative : Comprueba si el elemento es estrictamente negativo. Los valores cero se consideran inválidos.
- @NegativeOrZero : Comprueba si el elemento es negativo o cero.
- @Positive : Comprueba si el elemento es estrictamente positivo. Los valores cero se consideran inválidos.
- @PositiveOrZero : Comprueba si el elemento es positivo o cero.
- @DecimalMax(value=, inclusive=) : Comprueba si el valor numérico anotado es menor que el máximo especificado, cuando inclusive= falso. De lo contrario, si el valor es menor o igual al máximo especificado.
- @DecimalMin(value=, inclusive=) : Comprueba si el valor anotado es mayor que el mínimo especificado, cuando inclusive= falso. De lo contrario, si el valor es mayor o igual al mínimo especificado.

© JMA 2020. All rights reserved

Validaciones (JSR 303)

- @Digits: El elemento anotado debe ser un número cuyo valor tenga el número de dígitos especificado.
- @Past : Comprueba si la fecha anotada está en el pasado
- @PastOrPresent : Comprueba si la fecha anotada está en el pasado o en el presente
- @Future : Comprueba si la fecha anotada está en el futuro.
- @FutureOrPresent : Comprueba si la fecha anotada está en el presente o en el futuro
- @Email : Comprueba si la secuencia de caracteres especificada es una dirección de correo electrónico válida.
- @Pattern(regex=, flags=) : Comprueba si la cadena anotada coincide con la expresión regular regex considerando la bandera dadamatch.
- @Size(min=, max=) : Comprueba si el tamaño del elemento anotado está entre min y max (inclusive)

© JMA 2020. All rights reserved

Validaciones (Hibernate)

- @CreditCardNumber(ignoreNonDigitCharacters=): Comprueba que la secuencia de caracteres pasa la prueba de suma de comprobación de Luhn.
- @Currency(value=): Comprueba que la unidad monetaria de un javax.money.MonetaryAmount es una de las unidades monetarias especificadas.
- @DurationMax(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea mayor que el construido a partir de los parámetros de la anotación.
- @DurationMin(days=, hours=, minutes=, seconds=, millis=, nanos=, inclusive=): Comprueba que el elemento java.time.Duration no sea menor que el construido a partir de los parámetros de anotación.
- @EAN: Comprueba que la secuencia de caracteres sea un código de barras EAN válido
- @ISBN: Comprueba que la secuencia de caracteres sea un ISBN válido.

© JMA 2020. All rights reserved

Validaciones (Hibernate)

- @Length(min=, max=): Valida que la secuencia de caracteres esté entre min e max incluidos
- @CodePointLength(min=, max=, normalizationStrategy=): Valida que la longitud del punto de código de la secuencia de caracteres esté entre min e max incluidos.
- @LuhnCheck(startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos de la secuencia de caracteres pasan el algoritmo de suma de comprobación de Luhn.
- @Mod10Check(multiplier=, weight=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo genérico de suma de comprobación mod 10.
- @Mod11Check(threshold=, startIndex=, endIndex=, checkDigitIndex=, ignoreNonDigitCharacters=, treatCheck10As=, treatCheck11As=): Comprueba que los dígitos dentro de la secuencia de caracteres pasan el algoritmo de suma de comprobación mod 11.

© JMA 2020. All rights reserved

Validaciones (Hibernate)

- @Normalized(form=): Valida que la secuencia de caracteres anotados se normalice de acuerdo con lo dado form.
- @Range(min=, max=): Comprueba si el valor anotado se encuentra entre (inclusive) el mínimo y el máximo especificados
- @SafeHtml(additionalTags=, additionalTagsWithAttributes=, baseURL=, whitelistType=): Comprueba si el valor anotado contiene fragmentos potencialmente maliciosos como <script>.
- @ScriptAssert(lang=, script=, alias=, reportOn=): Comprueba si la secuencia de comandos proporcionada se puede evaluar correctamente con el elemento anotado.
- @UniqueElements: Comprueba que la colección solo contiene elementos únicos.
- @URL(protocol=, host=, port=, regexp=, flags=): Comprueba si la secuencia de caracteres anotada es una URL válida según RFC2396.

© JMA 2020. All rights reserved

Validaciones personalizadas

- Anotación personalizada para la validación:

```
import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.validation.Constraint;
import javax.validation.Payload;

@Target({ ElementType.FIELD })
@Retention(RetentionPolicy.RUNTIME)
@Constraint(validatedBy = NifValidator.class)
@Documented
public @interface NIF {
    String message() default "{validation.NIF.message}";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

© JMA 2020. All rights reserved

Validaciones personalizadas

- Clase del validator

```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;
public class NifValidator implements ConstraintValidator<NIF, String> {
    @Override
    public boolean isValid(String value, ConstraintValidatorContext context) {
        if(value == null) return true;
        value = value.toUpperCase();
        if(!value.matches("^[\\d{1,8}[A-Z]$")) return false;
        return "TRWAGMYFPDXBNJZSQVHLCKE".charAt(Integer.parseInt( value.substring(0, value.length() - 1)
            % 23)
            == value.charAt(value.length() - 1);
    }
}
```

- En el fichero ValidationMessages.properties

validation.NIF.message=\${validatedValue} no es un NIF válido.

© JMA 2020. All rights reserved

DTO

- Un objeto de transferencia de datos (DTO) es un objeto que define cómo se enviarán los datos a través de la red.
- Su finalidad es:
 - Desacoplar del nivel de servicio de la capa de base de datos.
 - Quitar las referencias circulares.
 - Ocultar determinadas propiedades que los clientes no deberían ver.
 - Omitir algunas de las propiedades con el fin de reducir el tamaño de la carga.
 - Eliminar el formato de grafos de objetos que contienen objetos anidados, para que sean más convenientes para los clientes.
 - Evitar el "exceso" y las vulnerabilidades por publicación.

© JMA 2020. All rights reserved

Lombok

<https://projectlombok.org/>

- En las clases Java hay mucho código que se repite una y otra vez: constructores, equals, getters y setters. Métodos que quedan definidos una vez que dicha clase ha concretado sus propiedades, y que salvo ajustes menores, serán siempre sota, caballo y rey.
- Project Lombok es una biblioteca de Java que se conecta automáticamente al editor y crea herramientas que automatizan la escritura de Java.
- Mediante simples anotaciones ya nunca mas vuelves a escribir otro método get o equals.


```
@Data @AllArgsConstructor @NoArgsConstructor public class MyDTO {
    private long id;
    private String name;
}
```
- La anotación @Value (no confundir con la de Spring) crea la versión de solo lectura.
- Es necesario agregar las bibliotecas al proyecto y configurar el entorno.

© JMA 2020. All rights reserved

ModelMapper

<http://modelmapper.org/>

- Las aplicaciones a menudo contienen modelos de objetos similares pero diferentes, donde los datos en dos modelos pueden ser similares pero la estructura y las responsabilidades de los modelos son diferentes. El mapeo de objetos facilita la conversión de un modelo a otro, permitiendo que los modelos separados permanezcan segregados.
- ModelMapper facilita el mapeo de objetos, al determinar automáticamente cómo se mapea un modelo de objeto a otro, de acuerdo con las convenciones, de la misma forma que lo haría un ser humano, al tiempo que proporciona una API simple y segura de refactorización para manejar casos de uso específicos.

```
ModelMapper modelMapper = new ModelMapper();
OrderDTO orderDTO = modelMapper.map(order, OrderDTO.class);
```

© JMA 2020. All rights reserved

Proyecciones

- Los métodos de consulta de Spring Data generalmente devuelven una o varias instancias de la raíz agregada administrada por el repositorio. Sin embargo, a veces puede ser conveniente crear proyecciones basadas en ciertos atributos de esos tipos. Spring Data permite modelar tipos de retorno dedicados, para recuperar de forma más selectiva vistas parciales de los agregados administrados.
- La forma más sencilla de limitar el resultado de las consultas solo a los atributos deseados es declarar una interfaz o DTO que exponga los métodos de acceso para las propiedades a leer, que deben coincidir exactamente con las propiedades de la entidad:

```
public interface NamesOnly {
    String getNombre();
    String getApellidos();
}
```

- El motor de ejecución de consultas crea instancias de proxy de esa interfaz en tiempo de ejecución para cada elemento devuelto y reenvía las llamadas a los métodos expuestos al objeto de destino.

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    List<NamesOnly> findByNombreStartingWith(String nombre);
}
```

© JMA 2020. All rights reserved

Proyecciones

- Las proyecciones se pueden usar recursivamente.

```
interface PersonSummary {
    String getNombre();
    String getApellidos();
    DireccionSummary getDireccion();
}
interface DireccionSummary {
    String getCiudad();
}
```

- En las proyecciones abiertas, los métodos de acceso en las interfaces de proyección también se pueden usar para calcular nuevos valores:

```
public interface NamesOnly {
    @Value("#{args[0] + ' ' + target.nombre + ' ' + target.apellidos}")
    String getNombreCompleto(String tratamiento);
    default String getFullName() {
        return getNombre.concat(" ").concat(getApellidos());
    }
}
```

© JMA 2020. All rights reserved

Proyecciones

- Se puede implementar una lógica personalizada mas compleja en un bean de Spring y luego invocarla desde la expresión SpEL:

```
@Component
class MyBean {
    String getFullName(Person person) { ... }
}
interface NamesOnly {
    @Value("#{@myBean.getFullName(target)}")
    String getFullName();
    ...
}
```

- Las proyecciones dinámicas permiten utilizar genéricos en la definición del repositorio para resolver el tipo devuelto en el momento de la invocación:

```
public interface ProfesorRepository extends JpaRepository<Profesor, Long> {
    <T> List<T> findByNombreStartingWith(String prefijo, Class<T> type);
}
dao.findByNombreStartingWith("J", ProfesorShortDTO.class)
```

© JMA 2020. All rights reserved

Serialización Jackson

- Jackson es una librería de utilidad de Java que nos simplifica el trabajo de serializar (convertir un objeto Java en una cadena de texto con su representación JSON), y des serializar (convertir una cadena de texto con una representación de JSON de un objeto en un objeto real de Java) objetos-JSON.
- Jackson es bastante “inteligente” y sin decirle nada es capaz de serializar y des serializar bastante bien los objetos. Para ello usa básicamente la reflexión de manera que si en el objeto JSON tenemos un atributo “name”, para la serialización buscará un método “getName()” y para la des serialización buscará un método “setName(String s)”.


```
ObjectMapper objectMapper = new ObjectMapper();
String jsonText = objectMapper.writeValueAsString(person);
Person person = new ObjectMapper().readValue(jsonText, Person.class);
```
- El proceso de serialización y des serialización se puede controlar declarativamente mediante anotaciones:

<https://github.com/FasterXML/jackson-annotations>

© JMA 2020. All rights reserved

Serialización Jackson

- **@JsonProperty:** indica el nombre alternativo de la propiedad en JSON.


```
@JsonProperty("name") public String getTheName() { ... }
@JsonProperty("name") public void setTheName(String name) { ... }
```
- **@JsonFormat:** especifica un formato para serializar los valores de fecha/hora.


```
@JsonFormat(shape = JsonFormat.Shape.STRING, pattern = "dd-MM-yyyy hh:mm:ss")
public Date eventDate;
```
- **@JsonIgnore:** marca que se ignore una propiedad (nivel miembro).


```
@JsonIgnore public int id;
```

© JMA 2020. All rights reserved

Serialización Jackson

- **@JsonIgnoreProperties**: marca que se ignore una o varias propiedades (nivel clase).


```
@JsonIgnoreProperties({ "id", "ownerName" })
@JsonIgnoreProperties(ignoreUnknown=true)
public class Item {
```
- **@JsonInclude**: se usa para incluir propiedades con valores vacíos/nulos/predeterminados.


```
@JsonInclude(Include.NON_NULL)
public class Item {
```
- **@JsonAutoDetect**: se usa para anular la semántica predeterminada qué propiedades son visibles y cuáles no.


```
@JsonAutoDetect(fieldVisibility = Visibility.ANY)
public class Item {
```

© JMA 2020. All rights reserved

Serialización Jackson

- **@JsonView**: permite indicar la Vista en la que se incluirá la propiedad para la serialización / deserialización.

```
public class Views {
    public static class Partial {}
    public static class Complete extends Partial {}
}

public class Item {
    @JsonView(Views.Partial.class)
    public int id;
    @JsonView(Views.Partial.class)
    public String itemName;
    @JsonView(Views.Complete.class)
    public String ownerName;
}

String result = new ObjectMapper().writerWithView(Views.Partial.class)
    .writeValueAsString(item);
```

© JMA 2020. All rights reserved

Serialización Jackson

- **@JsonFilter:** indica el filtro que se utilizará durante la serialización (es obligatorio suministrarlo).

```
@JsonFilter("ItemFilter")
public class Item {
    public int id;
    public String itemName;
    public String ownerName;
}

FilterProvider filters = new SimpleFilterProvider().addFilter("ItemFilter",
SimpleBeanPropertyFilter.filterOutAllExcept("id", "itemName"));
MappingJacksonValue mapping = new MappingJacksonValue(dao.findAll());
mapping.setFilters(filters);
return mapping;
```

© JMA 2020. All rights reserved

Serialización Jackson

- **@JsonManagedReference** y **@JsonBackReference**: se utilizan para manejar las relaciones maestro/detalle marcando la colección en el maestro y la propiedad inversa en el detalle (multiples relaciones requieren asignar nombres unicos).

```
@JsonManagedReference
public User owner;
@JsonBackReference
public List<Item> userItems;
```

- **@JsonIdentityInfo:** indica la identidad del objeto para evitar problemas de recursión infinita.

```
@JsonIdentityInfo(
    generator = ObjectIdGenerators.PropertyGenerator.class,
    property = "id")
public class Item {
    public int id;
```

© JMA 2020. All rights reserved

Serialización XML (JAXB)

- JAXB (Java XML API Binding) proporciona una manera rápida, conveniente de crear enlaces bidireccionales entre los documentos XML y los objetos Java. Dado un esquema, que especifica la estructura de los datos XML, el compilador JAXB genera un conjunto de clases de Java que contienen todo el código para analizar los documentos XML basados en el esquema. Una aplicación que utilice las clases generadas puede construir un árbol de objetos Java que representa un documento XML, manipular el contenido del árbol, y regenerar los documentos del árbol, todo ello en XML sin requerir que el desarrollador escriba código de análisis y de proceso complejo.
- Los principales beneficios de usar JAXB son:
 - Usa tecnología Java y XML
 - Garantiza datos válidos
 - Es rápida y fácil de usar
 - Puede restringir datos
 - Es personalizable
 - Es extensible

© JMA 2020. All rights reserved

Anotaciones principales (JAXB)

- Para indicar a los formateadores JAXB como transformar un objeto Java a XML y viceversa se puede anotar (`javax.xml.bind.annotation`) la clases JavaBean para que JAXP infiera el esquema de unión.
- Las principales anotaciones son:
 - `@XmlRootElement(namespace = "namespace")`: Define la raíz del XML.
 - `@XmlElement(name = "newName")`: Define el elemento de XML que se va usar.
 - `@XmlAttribute(required=true)`: Serializa la propiedad como un atributo del elemento.
 - `@XmlAttribute`: Mapea un propiedad JavaBean como un XML ID.
 - `@XmlType(propOrder = { "field2", "field1",... })`: Permite definir en qué orden se van escribir los elementos dentro del XML.
 - `@XmlElementWrapper`: Envuelve en un elemento los elementos de una colección.
 - `@XmlTransient`: La propiedad no se serializa.

© JMA 2020. All rights reserved

Jackson DataFormat XML

- Extensión de formato de datos para Jackson (<http://jackson.codehaus.org>) para ofrecer soporte alternativo para serializar POJOs como XML y deserializar XML como pojos. Soporte implementado sobre la API de Stax (`javax.xml.stream`), mediante la implementación de tipos de API de Streaming de Jackson como `JsonGenerator`, `JsonParser` y `JsonFactory`. Algunos tipos de enlace de datos también se anularon (`ObjectMapper` se clasificó como `XmlMapper`).
- Dependencia:

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

© JMA 2020. All rights reserved

MongoDB

- MongoDB (<https://www.mongodb.com/>) es una base de datos orientada a documentos, open source, que toma su nombre del inglés **humongous**, "enorme" y se enmarca en la familia de bases de datos NoSQL.
- En lugar de guardar los datos en tablas/filas/columnas, guarda los datos en documentos (estructura jerárquica).
- Los documentos se expresan en JSON y son almacenados en formato BSON (representación binaria de JSON).
- Los documentos se almacenan en colecciones, concepto similar a una tabla de una base de datos relacional.
- Una de las diferencias más importantes con respecto a las bases de datos relacionales, es que no es necesario seguir un esquema.
- Los documentos de una misma colección pueden tener esquemas diferentes, es decir, en una colección se pueden almacenar diferentes tipos de documentos.
- Todos los documentos tienen un identificador único denominado `_id`, se autogenera en caso de ser necesario

© JMA 2020. All rights reserved

MongoDB

Cuando usar MongoDB

- Prototipos y aplicaciones simples
- Hacer la transición de front a back
- Aplicaciones con mucha carga de escritura
- Agregado de datos a un nivel medio/alto
- Aplicaciones con datos muy heterogéneos
- Enormes colecciones de datos (sharding)
- Almacenar ficheros (sharding)

Cuando no usar MongoDB

- Mongo no puede hacer JOINs
- El lenguaje de consulta menos potente que SQL
- No tiene transacciones
- La velocidad baja al subir la seguridad (escritura)
- Aunque es muy fácil empezar con MongoDB, si la aplicación crece mucho, los modelos complejos van a requerir JOINs

© JMA 2020. All rights reserved

MongoDB: Instalación

- Se puede utilizar un instalador o instalarla manualmente.
- Descargar la última versión estable del Community Server desde:
 - <https://www.mongodb.com/download-center?jmp=nav#community>
- Instalación manual:
 - Crear una carpeta donde realizar la instalación
 - Descomprimir el fichero en el directorio
 - Crear una carpeta \data que contendrá los ficheros de datos
 - En una consola de comandos, en el directorio \bin, hay que levantar el servidor:
 - mongod --dbpath <path\data> --nojournal
 - Es conveniente agregar al PATH la carpeta \bin
- Desde otra consola, sobre el directorio \bin, se puede acceder al interfaz de comandos
 - mongo host:puerto/database -u usuario -p password

© JMA 2020. All rights reserved

MongoDB: Comandos

- **show dbs**: muestra los nombres de las bases de datos
- **show collections**: muestra las colecciones en la base de datos actual
- **show users**: muestra los usuarios en la base de datos actual
- **show profile**: muestra las entradas más recientes de system.profile con el tiempo > = 1ms
- **show logs**: muestra los nombres de los logs accesibles
- **show log [name]**: imprime el último segmento de registro en la memoria, 'global' es el predeterminado
- **use <db_name>**: establece la base de datos actual, creándola si es necesario
- **db**: muestra el nombre de la base de datos actual
- **db.dropDatabase()**: se usa para eliminar una base de datos existente.
- **exit**: sale de la shell de mongo

© JMA 2020. All rights reserved

MongoDB: Comandos

- **colección.count**: número de documentos
- **colección.findOne**: recuperar un documento
- **colección.find**: recuperar varios documentos
- **colección.insert**: inserta un documento, crea la colección si es necesario
- **colección.update**: modifica parcialmente documentos
- **colección.save**: guardar/actualizar un documento
- **colección.remove**: borrar uno o varios documentos
- **colección.rename**: cambia de nombre la colección
- **colección.drop**: elimina la colección

© JMA 2020. All rights reserved

MongoDB: Spring Boot

- Añadir al proyecto:
 - NoSQL > Spring Data MongoDB

- Configurar:

```
#spring.data.mongodb.uri=mongodb://localhost:27017/db
# Por defecto usa la base de datos "test"
spring.data.mongodb.database=db
#spring.data.mongodb.host=localhost
#spring.data.mongodb.port=27017
#spring.data.mongodb.username=
#spring.data.mongodb.password=
#spring.data.mongodb.repositories.enabled=true
```

© JMA 2020. All rights reserved

Anotaciones NoSQL

Anotación	Descripción
@Document	de clase, para indicar que esta clase es un documento para la asignación a la base de datos (puede especificarse el nombre de la colección donde se almacenará la base de datos).
@CompoundIndex	de clase, para declarar índices compuestos.
@Id	de campo, para marcar el campo utilizado para la identidad.
@Field	de campo, para indicar el nombre en la base de datos
@DBRef	de campo, para indicar que se debe almacenar utilizando un com.mongodb.DBRef.
@Indexed	de campo, para describir cómo se indexa.
@GeoSpatialIndexed	de campo, para describir cómo se geoindexa.
@Transient	excluye el campo del almacenamiento en la base de datos
@PersistenceConstructor	marca el constructor (incluso en un paquete protegido), para usar cuando se crea una instancia del objeto desde la base de datos. Los argumentos del constructor se asignan por nombre a los valores clave en el DBObject recuperado.
@Value	se puede aplicar a los argumentos del constructor para usar una declaración de Spring Expression Language en la transformación del valor de una clave recuperada en la base de datos antes de que se use para construir un objeto de dominio.

© JMA 2020. All rights reserved

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
After	findByBirthdateAfter(Date date)	{"birthdate" : {"\$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"\$gt" : age}}
GreaterThanOrEqualTo	findByAgeGreaterThanOrEqualTo(int age)	{"age" : {"\$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"\$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"\$lt" : age}}
LessThanOrEqualTo	findByAgeLessThanOrEqualTo(int age)	{"age" : {"\$lte" : age}}
Between	findByAgeBetween(int from, int to)	{"age" : {"\$gt" : from, "\$lt" : to}}
In	findByAgeIn(Collection ages)	{"age" : {"\$in" : [ages...]}}
NotIn	findByAgeNotIn(Collection ages)	{"age" : {"\$nin" : [ages...]}}
IsNotNull, NotNull	findByFirstnameNotNull()	{"firstname" : {"\$ne" : null}}
IsNull, Null	findByFirstnameNull()	{"firstname" : null}

© JMA 2020. All rights reserved

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
Like, StartingWith, EndingWith	findByFirstnameLike(String name)	{"firstname" : name} (name as regex)
NotLike, IsNotLike	findByFirstnameNotLike(String name)	{"firstname" : { "\$not" : name }} (name as regex)
Containing on String	findByFirstnameContaining(String name)	{"firstname" : name} (name as regex)
NotContaining on String	findByFirstnameNotContaining(String name)	{"firstname" : { "\$not" : name}} (name as regex)
Containing on Collection	findByAddressesContaining(Address address)	{"addresses" : { "\$in" : address}}
NotContaining on Collection	findByAddressesNotContaining(Address address)	{"addresses" : { "\$not" : { "\$in" : address}}}
Regex	findByFirstnameRegex(String firstname)	{"firstname" : {"\$regex" : firstname }}
(No keyword)	findByFirstname(String name)	{"firstname" : name}
Not	findByFirstnameNot(String name)	{"firstname" : {"\$ne" : name}}

© JMA 2020. All rights reserved

Repositorio (MongoDB)

P. Clave	Muestra	Resultado lógico
Near	findByLocationNear(Point point)	{"location" : {"\$near" : [x,y]}}
Near	findByLocationNear(Point point, Distance max)	{"location" : {"\$near" : [x,y], "\$maxDistance" : max}}
Near	findByLocationNear(Point point, Distance min, Distance max)	{"location" : {"\$near" : [x,y], "\$minDistance" : min, "\$maxDistance" : max}}
Within	findByLocationWithin(Circle circle)	{"location" : {"\$geoWithin" : {"\$center" : [[x, y], distance]}}}
Within	findByLocationWithin(Box box)	{"location" : {"\$geoWithin" : {"\$box" : [[x1, y1], x2, y2]}}}
IsActive, True	findByActiveIsActive()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"\$exists" : exists }}

© JMA 2020. All rights reserved

Repositorio

- Consultas basadas en JSON:

```
@Query("{ 'age' : {'$gt' : 67}}")
List<Profesor> findJubilados();

@Query(value="{ 'nombre' : ?0 }", fields="{ 'nombre' : 1, 'apellidos' : 1}")
List<ProfesorShort> findByNombre(String nombre);
```

- Consultas basadas en JSON con expresiones SpEL

```
@Query("{ 'lastname': ?#[[0]] }")
List<Person> findByQueryWithExpression(String param0);

@Query("{ 'id': ?#[ [0] ? {$exists :true} : [1] ] }")
List<Person> findByQueryWithExpressionAndNestedObject(boolean param0, String param1);
```

© JMA 2020. All rights reserved

Redis

- Redis (<https://redis.io>) es un almacén estructurado de datos en memoria de código abierto, que se utiliza como base de datos, caché y agente de mensajes.
- Basado en el almacenamiento en tablas de hashes (clave/valor) es compatible con estructuras de datos como cadenas, hashes, listas, conjuntos, conjuntos ordenados con consultas de rango, mapas de bits, hiperloglogs, índices geoespaciales con consultas de radio y flujos.
- Redis tiene replicación incorporada, scripts Lua, desalojo de LRU, transacciones y diferentes niveles de persistencia en disco, y proporciona alta disponibilidad a través de Redis Sentinel y partición automática con Redis Cluster.
- La popularidad de Redis se debe en gran medida a su espectacular velocidad, ya que mantiene la información en memoria; pero también a su sencillez de uso, productividad y flexibilidad.
- Sin embargo lo que hace que Redis sea tan popular, es que además de permitir asociar valores de tipo string a una clave, permite utilizar tipos de datos avanzados, que junto a las operaciones asociadas a estos tipos de datos, logra resolver muchos casos de uso de negocio, que a priori no pensariamos que fuéramos capaces con clave-valor. No en vano, Redis se define usualmente como un servidor de estructuras de datos, aunque en sus inicios, haciendo honor a su nombre, fuese un simple diccionario remoto (REmote DIctionary Server).

© JMA 2020. All rights reserved

Redis: Casos de uso

- Caché de páginas web
 - Podemos usar Redis como caché de páginas HTML, o fragmentos de estas, lo que acelerará el acceso a las mismas, a la vez que evitamos llegar a los servidores web o de aplicaciones, reduciendo la carga en estos y en los sistemas de bases de datos a los que los mismos accedan. Además de un incremento en la velocidad, esto puede suponer un importante ahorro económico en términos de hardware y licencias de software.
- Almacenamiento de sesiones de usuario
 - Podemos usar Redis como un almacén de sesiones de muy rápido acceso, en el que mantengamos el identificador de sesión junto con toda la información asociada a la misma. Además de reducir los tiempos de latencia de nuestra solución, igual que en el caso anterior evitaremos una vez más accesos a otras bases de datos. Además Redis permite asociar un tiempo de expiración a cada clave, con lo que las sesiones finalizarán automáticamente sin tener que gestionarlo en el código de la aplicación.

© JMA 2020. All rights reserved

Redis: Casos de uso

- Almacenamiento de carritos de la compra
 - De forma muy similar al almacén de sesiones de usuario, podemos almacenar en Redis los artículos contenidos en la cesta de la compra de un usuario, y la información asociada a los mismos, permitiéndonos acceder en cualquier momento a ellos con una latencia mínima.
- Caché de base de datos
 - Otra forma de descargar a las bases de datos operacionales es almacenar en Redis el resultado de determinadas consultas que se ejecuten con mucha frecuencia, y cuya información no cambia a menudo, o no es crítico mantener actualizada al instante.
- Contadores y estadísticas
 - Para muchos casos de uso es necesario manejar contadores y estadísticas en tiempo real, y Redis tiene soporte para la gestión concurrente y atómica de los mismos. Algunos ejemplos posibles serían el contador de visualización de un producto, votos de usuarios, o contadores de acceso a un recurso para poder limitar su uso.

© JMA 2020. All rights reserved

Redis: Casos de uso

- Listas de elementos recientes
 - Es muy habitual mostrar listas en las que aparecen las últimas actualizaciones de algún elemento hechas por los usuarios. Por ejemplo, los últimos comentarios sobre un post, las últimas imágenes subidas, los artículos de un catálogo vistos recientemente, etc. Este tipo de operaciones suele ser muy costoso para las bases de datos relacionales, sobre todo cuando el volumen de información se va haciendo mayor, pero Redis es capaz de resolver esta operación con independencia del volumen.
- Base de datos principal
 - Para determinados casos, Redis se puede usar como almacenamiento principal gracias a la potencia de modelado que permiten sus avanzados tipos de datos. Destaca su uso en casos como los microservicios, en los que podemos aprovechar la velocidad de Redis para construir soluciones especializadas, simples de implementar y mantener, que a la vez ofrecen un alto

© JMA 2020. All rights reserved

Redis: Comandos

- Valores:
 - APPEND
 - GET
 - GETBIT
 - GETRANGE
 - GETSET
 - BITCOUNT
 - BITFIELD
 - BITOP
 - BITPOS
 - MGET
 - MSET
 - MSETNX
 - PSETEX
 - SET
 - SETBIT
 - SETEX
 - SETNX
 - SETRANGE
- Contadores:
 - STRLEN
 - INCR
 - INCRBY
 - INCRBYFLOAT
 - DECR
 - DECRBY
- Listas
 - BLPOP
 - BRPOP
 - BRPOPLPUSH
 - LINDEX
 - LINsert
 - LLEN
 - LPOP
 - LPUSH
 - LPUSHX
 - LRANGE
 - LREM
- Conjuntos:
 - LSET
 - LTRIM
 - RPOP
 - RPOPLPUSH
 - RPUSH
 - RPUSHX
 - SADD
 - SCARD
 - SDIFF
 - SDIFFSTORE
 - SINTER
 - SINTERSTORE
 - SISMEMBER
 - SMEMBERS
 - SMOVE
 - SPOP
 - SRANDMEMBER
 - SREM
- SET Ordenados
- Hash
 - HDEL
 - HEXISTS
 - HGET
 - HGETALL
 - HINCRBY
 - HINCRBYFLOAT
 - HKEYS
 - HLEN
 - HMGET
 - HMSET
 - HSCAN
 - HSET
 - HSETNX
 - HSTRLEN
- Mensajería:
 - PSUBSCRIBE
 - PUBLISH
 - PUBSUB
 - PUNSUBSCRIBE
 - SUBSCRIBE
 - UNSUBSCRIBE
- Administración
 - [Y muchos más](#)

© JMA 2020. All rights reserved

Redis: Instalación

- Se puede utilizar un instalador o instalarla manualmente.
- Descargar la última versión estable para Windows desde <https://github.com/MicrosoftArchive/redis/releases>
- Puerto: 6379
- Instalación manual:
 - Crear una carpeta donde realizar la instalación
 - Descomprimir el fichero en el directorio
 - Instalar el servicio:
 - redis-server --service-install redis.windows-service.conf --loglevel verbose
- Para arrancar el servicio:
 - redis-server --service-start
- Para parar el servicio:
 - redis-server --service-stop
- Desde otra consola se puede acceder al interfaz de comandos con:
 - redis-cli

© JMA 2020. All rights reserved

Redis: Spring Boot

- Añadir al proyecto:
 - NoSQL > Spring Data Redis (Access+Driver)
- Configurar:


```
spring.redis.port=6379
spring.redis.host=localhost
#spring.redis.password=
```
- Uso de repositorios:
 - Anotaciones de entidad:
 - @RedisHash: de clase, para indicar que esta clase es parte de un conjunto para la asignación a la base de datos (debe especificarse el nombre del conjunto donde se almacenará en la base de datos).
 - @Id: de campo, para marcar el campo utilizado para la identidad.
 - Creación del repositorio


```
public interface PersonasRepository extends PagingAndSortingRepository<Persona, String> {}
```

© JMA 2020. All rights reserved

Redis: Repositorios

P.Clave	Muestra	Fragmento Redis
And	findByLastnameAndFirstname	SINTER ...:firstname:rand ...:lastname:al'thor
Or	findByLastnameOrFirstname	SUNION ...:firstname:rand ...:lastname:al'thor
Is, Equals	findByFirstname, findByFirstnamels, findByFirstnameEquals	SINTER ...:firstname:rand
IsTrue	FindByAliveIsTrue	SINTER ...:alive:1
IsFalse	findByAliveIsFalse	SINTER ...:alive:0
Top,First	findFirst10ByFirstname, findTop5ByFirstname	

© JMA 2020. All rights reserved

Redis: RedisTemplate

```

@RestController
@RequestMapping(path = "/me-gusta")
public class ContadorResource {
    public final String ME_GUSTA_CONT="megusta";
    @Autowired
    private StringRedisTemplate template;
    private ValueOperations<String, String> redisValue;
    @PostConstruct
    private void inicializa() {
        redisValue = template.opsForValue();
        if(redisValue.get(ME_GUSTA_CONT) == null)
            redisValue.set(ME_GUSTA_CONT, "0");
    }
    @GetMapping
    private String get() {
        return "Llevas " + redisValue.get(ME_GUSTA_CONT);
    }
    @PostMapping
    private String add() {
        return "Llevas " + redisValue.increment(ME_GUSTA_CONT);
    }
    @PutMapping("/{id}")
    private String add(@PathVariable int id) {
        long r = 0;
        Date ini = new Date();
        for(int i= 0; i++ < id*1000; r = redisValue.increment(ME_GUSTA_CONT));
        return "Llevas " + r + ". Tardo " + ((new Date()).getTime() - ini.getTime()) + " ms.";
    }
}

```

© JMA 2020. All rights reserved

Patrón: Base de datos compartida

- Motivación:
 - Ha aplicado el patrón de arquitectura de Microservicios. La mayoría de los servicios necesitan conservar los datos en algún tipo de base de datos..
- Intención:
 - ¿Cuál es la arquitectura de la base de datos en una aplicación de microservicios?
- Requisitos:
 - Los servicios se deben acoplar libremente para que se puedan desarrollar, implementar y escalar de forma independiente
 - Algunas transacciones de negocio deben imponer invariantes que abarcan múltiples servicios, otras deben actualizar los datos propiedad de múltiples servicios.
 - Algunas transacciones de negocio necesitan consultar datos de múltiples servicios.
 - Algunas consultas deben unir datos que son propiedad de múltiples servicios.
 - Las bases de datos a veces se deben replicar y fragmentar para escalar.
 - Los diferentes servicios tienen diferentes requisitos de almacenamiento de datos. Para algunos servicios, una base de datos relacional es la mejor opción, otros pueden necesitar una base de datos NoSQL como MongoDB o Redis.
- Solución:
 - Utilizar una base de datos (única) que sea compartida por múltiples servicios. Cada servicio accede libremente a los datos que son propiedad de otros servicios que utilizan transacciones ACID locales.

© JMA 2020. All rights reserved

Patrón: Base de datos compartida

- Implementación:
 - Crear servicios con Spring Boot y Spring Data
- Consecuencias:
 - Los beneficios de este patrón son:
 - Un desarrollador utiliza transacciones ACID familiares y directas para imponer la consistencia de los datos
 - Una sola base de datos es más simple de operar
 - Los inconvenientes de este patrón son:
 - Acoplamiento en tiempo de desarrollo: un desarrollador que trabaja un servicio necesitará coordinar los cambios de esquema con los desarrolladores de otros servicios que acceden a las mismas tablas. Dichos cambios implican un despliegue coordinado de todos los servicios afectados. Este acoplamiento y la coordinación adicional ralentizarán el desarrollo.
 - Acoplamiento en tiempo de ejecución: dado que todos los servicios acceden a la misma base de datos, pueden interferir entre sí y provocar fallos en cascada.
 - La base de datos única puede no satisfacer los requisitos de almacenamiento y acceso a datos de todos los servicios.
- Patrones relacionados:
 - El patrón de Base de datos por servicio es un patrón alternativo.

© JMA 2020. All rights reserved

Patrón: Base de datos por servicio

- Motivación:
 - Ha aplicado el patrón de arquitectura de Microservicios. La mayoría de los servicios necesitan conservar los datos en algún tipo de base de datos..
- Intención:
 - ¿Cuál es la arquitectura de la base de datos en una aplicación de microservicios?
- Requisitos:
 - Los servicios se deben acoplar libremente para que se puedan desarrollar, implementar y escalar de forma independiente
 - Algunas transacciones de negocio deben imponer invariantes que abarcan múltiples servicios, otras deben actualizar los datos propiedad de múltiples servicios.
 - Algunas transacciones de negocio necesitan consultar datos que son propiedad de múltiples servicios.
 - Algunas consultas deben unir datos que son propiedad de múltiples servicios.
 - Las bases de datos a veces se deben replicar y fragmentar para escalar.
 - Los diferentes servicios tienen diferentes requisitos de almacenamiento de datos. Para algunos servicios, una base de datos relacional es la mejor opción. Otros servicios pueden necesitar una base de datos NoSQL como MongoDB, que es buena para almacenar datos complejos y no estructurados, o Redis, que está diseñada para almacenar y consultar datos de memoria.

© JMA 2020. All rights reserved

Patrón: Base de datos por servicio

- Solución:
 - Mantener los datos persistentes de cada microservicio en privados para el servicio y accesibles solo a través de su API. Las transacciones de un servicio solo involucran su base de datos.
 - La base de datos del servicio es parte de la implementación de ese servicio y no se puede acceder directamente por otros servicios.
 - Hay algunas formas diferentes de mantener la privacidad de los datos persistentes de un servicio. No es necesario aprovisionar un servidor de base de datos para cada servicio. Por ejemplo, si está utilizando una base de datos relacional, las opciones son:
 - Tablas privadas por servicio: cada servicio posee un conjunto de tablas a las que solo debe acceder ese servicio
 - Esquema por servicio: cada servicio tiene un esquema de base de datos que es privado para ese servicio
 - Servidor de base de datos por servicio: cada servicio tiene su propio servidor de base de datos.
 - Las tablas privadas por servicio y el esquema por servicio tienen la sobrecarga más baja. Usar un esquema por servicio es atractivo ya que hace que la propiedad sea más clara. Algunos servicios de alto rendimiento pueden necesitar su propio servidor de base de datos.
 - Es una buena idea crear barreras que impongan esta modularidad. Se debería asignar un usuario de base de datos diferente a cada servicio y utilizar control de acceso y permisos de la base de datos, dado que los desarrolladores de otros servicios siempre estarán tentados a pasar por alto la API de un servicio y acceder a sus datos directamente.
- Implementación:
 - Crear servicios con Spring Boot y Spring Data

© JMA 2020. All rights reserved

Patrón: Base de datos por servicio

- Consecuencias:
 - El uso de una base de datos por servicio tiene los siguientes beneficios:
 - Ayuda a asegurar que los servicios estén acoplados libremente. Los cambios en la base de datos de un servicio no afectan a ningún otro servicio.
 - Cada servicio puede utilizar el tipo de base de datos que mejor se adapte a sus necesidades
 - El uso de una base de datos por servicio tiene los siguientes inconvenientes:
 - Implementar transacciones distribuidas que abarcan múltiples servicios no es sencillo. Es mejor evitar las transacciones distribuidas debido al teorema de CAP: es imposible en un sistema distribuido garantizar simultáneamente la consistencia (Consistency), la disponibilidad (Availability) y tolerancia al particionado (Partition Tolerance). Además, muchas bases de datos modernas (NoSQL) no las admiten. La mejor solución es usar el patrón Saga: Los servicios publican eventos cuando actualizan datos y otros servicios se suscriben a eventos y actualizan sus datos en respuesta.
 - Implementar consultas que unen datos que ahora están en múltiples bases de datos es un desafío.
 - Complejidad de gestionar múltiples bases de datos SQL y NoSQL
- Patrones relacionados:
 - El patrón de arquitectura de microservicio crea la necesidad de este patrón
 - El patrón Saga es una forma útil de implementar transacciones eventualmente consistentes
 - El patrón de API de composición de la API y el de Segregación de responsabilidad de consultas y comandos (CQRS) son formas útiles de implementar consultas
 - El patrón de Base de datos compartida es un patrón alternativo.

© JMA 2020. All rights reserved

Patrón: Eventos de dominio

- Motivación:
 - Un servicio a menudo necesita publicar eventos cuando actualiza sus datos. Estos eventos pueden ser necesarios, por ejemplo, para actualizar una vista CQRS. Alternativamente, el servicio podría participar en una Saga basada en coreografía, que utiliza eventos para la coordinación.
- Intención:
 - ¿Cómo publica un servicio un evento cuando actualiza sus datos?
- Solución:
 - Organizar la lógica de negocios de un servicio como una colección de agregados DDD que emiten eventos de dominio cuando se crean o actualizan. El servicio publica estos eventos de dominio para que puedan ser consumidos por otros servicios.
- Patrones relacionados:
 - Los patrones Saga y CORS crean la necesidad de este patrón.
 - El patrón Agregado se utiliza para estructurar la lógica empresarial.
 - El patrón de Bandeja de salida transaccional se utiliza para publicar eventos como parte de una transacción de base de datos
 - El aprovisionamiento de eventos se utiliza a veces para publicar eventos de dominio.

© JMA 2020. All rights reserved

Patrón: Saga

- Motivación:
 - Se ha aplicado el patrón base de datos por servicio. Cada servicio tiene su propia base de datos. Sin embargo, algunas transacciones abarcan varios servicios, por lo que necesita un mecanismo para garantizar la consistencia de los datos en todos los servicios. Dado que datos se encuentran en diferentes bases de datos, la aplicación no puede simplemente usar una transacción local ACID.
- Intención:
 - ¿Cómo mantener la consistencia de los datos en todos los servicios?
- Requisitos:
 - Las transacciones de Confirmación en dos fases (2PC) no son una opción
- Solución:
 - Implementar cada transacción que abarque múltiples servicios como una saga (de películas o libros). Una saga es una secuencia de transacciones locales. Cada transacción local actualiza su base de datos local y publica un mensaje o evento para desencadenar la siguiente transacción local en la saga. Si una transacción local falla porque viola una regla de negocio, entonces la saga ejecuta una serie de transacciones compensatorias (reversión) que deshacen los cambios realizados por las transacciones locales anteriores.
 - Hay dos formas de coordinar las sagas:
 - Coreografía: cada transacción local publica eventos de dominio que activan transacciones locales en otros servicios
 - Orquestación: un orquestador (objeto) le dice a los participantes qué transacciones locales deben ejecutar

© JMA 2020. All rights reserved

Patrón: Saga

- Implementación:
 - Crear servicios con Spring Boot y Spring Data, Spring for RabbitMQ, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Permite que una aplicación mantenga la consistencia de los datos en múltiples servicios sin usar transacciones distribuidas
 - Esta solución tiene los siguientes inconvenientes:
 - El modelo de programación es más complejo, se debe diseñar transacciones de compensación que deshagan explícitamente los cambios realizados anteriormente en una saga.
 - No permite utilizar bloqueos que eviten efectos secundarios a consecuencia de las transacciones de compensación.
 - También hay que abordar los siguientes temas:
 - Para ser confiable, un servicio debe actualizar su base de datos y publicar un mensaje/evento. No puede usar el mecanismo tradicional de una transacción distribuida que abarca la base de datos y el intermediario de mensajes. En su lugar, debe utilizar patrones adicionales.
- Patrones relacionados:
 - El patrón de base de datos por servicio crea la necesidad de este patrón.
 - Los patrones Abastecimiento de eventos y Eventos de aplicación son formas de actualizar de forma atómica el estado y publicar mensajes/eventos:
 - Una saga basada en coreografía puede publicar eventos usando agregados y eventos de dominio.

© JMA 2020. All rights reserved

Patrón: Event sourcing

- Motivación:
 - Un servicio normalmente necesita actualizar de forma atómica la base de datos y publicar mensajes / eventos. Por ejemplo cuando utiliza el patrón Saga. Para ser confiable, cada paso de una saga debe actualizar atómicamente la base de datos y publicar mensajes / eventos. Alternativamente, podría usar el patrón de evento de dominio, tal vez para implementar CQRS. En cualquier caso, no es viable utilizar una transacción distribuida que abarque la base de datos y el intermediario de mensajes para actualizar de forma atómica la base de datos y publicar mensajes / eventos.
- Intención:
 - ¿Cómo actualizar de forma fiable/atómica la base de datos y publicar mensajes/eventos?
- Requisitos:
 - Las transacciones de Confirmación en dos fases (2PC) no son una opción
- Solución:
 - Una buena solución es registrar el historial de la secuencia de eventos de cambio de una entidad de dominio desde el momento de su creación hasta el momento actual y hacer persistente dicha secuencia. Cuando el estado de una entidad de dominio cambia, se agrega un nuevo evento a la lista de eventos. Dado que guardar un evento es una sola operación, es inherentemente atómico. La aplicación reconstruye el estado actual de una entidad mediante la reproducción de los eventos.
 - Las aplicaciones persisten eventos en un almacén de eventos, que es una base de datos de eventos. El almacén de eventos tiene una API para agregar y recuperar eventos de una entidad. El almacén de eventos también se comporta como un intermediario de mensajes. Proporciona una API que permite a los servicios suscribirse a eventos. Cuando un servicio guarda un evento en el almacén de eventos, se entrega a todos los suscriptores interesados.
 - Algunas entidades pueden tener una gran cantidad de eventos. Para optimizar la carga, una aplicación puede guardar periódicamente una instantánea del estado actual de una entidad. Para reconstruir el estado actual, la aplicación encuentra la instantánea más reciente y los eventos que han ocurrido desde esa instantánea. Como resultado, hay menos eventos para reproducir.

© JMA 2020. All rights reserved

Patrón: Event sourcing

- Implementación:
 - Crear servicios con Spring Boot y Spring Data, Spring for RabbitMQ, ...
- Consecuencias:
 - El patrón tiene varios beneficios:
 - Resuelve uno de los problemas clave en la implementación de una arquitectura dirigida por eventos y permite publicar eventos de manera confiable cada vez que cambia el estado.
 - Debido a que persisten los eventos en lugar de los objetos de dominio, principalmente evita el problema de desajuste de impedancia relacional del objeto.
 - Proporciona un registro de auditoría 100% confiable de los cambios realizados en una entidad de dominio
 - Permite implementar consultas temporales que determinan el estado de una entidad en cualquier momento.
 - La lógica de negocios basada en la obtención de eventos consiste en entidades dominio débilmente acopladas que intercambian eventos. Esto hace que sea mucho más fácil migrar desde una aplicación monolítica a una arquitectura de microservicio.
 - El patrón también tiene varios inconvenientes:
 - Es un estilo de programación diferente y desconocido, por lo que hay una curva de aprendizaje.
 - El almacén de eventos es difícil de consultar ya que requiere consultas típicas para reconstruir el estado de las entidades dominio. Es probable que sea complejo e inefficiente. Como resultado, la aplicación debe utilizar Segregación de responsabilidad de consultas y comandos (CQRS) para implementar consultas. Esto, a su vez, significa que las aplicaciones deben manejar datos finalmente consistentes.
- Patrones relacionados:
 - Los patrones de Saga y Eventos Dominio crean la necesidad de este patrón.
 - El CQRS se debe utilizar a menudo con event sourcing.
 - El event sourcing implementa el patrón de registro de auditoría .

© JMA 2020. All rights reserved

Patrón: Segregación de responsabilidad de consultas y comandos (CQRS)

- Motivación:
 - Se han aplicado los patrones de arquitectura de microservicios y de base de datos por servicio. Como resultado, ya no es sencillo implementar consultas que unen datos de múltiples servicios. Si además se ha aplicado el patrón Event sourcing, los datos ya no se consultan fácilmente.
- Intención:
 - ¿Cómo implementar una consulta que recupera datos de múltiples servicios en una arquitectura de microservicio?
- Requisitos:
- Solución:
 - La idea básica es que puede dividir las operaciones de un sistema en dos categorías claramente separadas:
 - Consultas: Devuelven un resultado sin cambiar el estado del sistema y no tienen efectos secundarios.
 - Comandos: Cambian el estado de un sistema.
 - La solución pasa por dividir en dos subsistemas diferenciados, uno responsable de los comandos y otro responsable de las consultas. El sistema tradicional quedaría como el subsistema de comandos para el mantenimiento de los datos y se derivaría todas las consultas al subsistema de consultas. El subsistema de consultas contaría con su propio modelo de información y mecanismo de persistencia desnortinalizado y optimizado para el tipo de consultas que tenga que atender, obtendría sus datos mediante mecanismos de sincronización.

© JMA 2020. All rights reserved

Patrón: Segregación de responsabilidad de consultas y comandos (CQRS)

- Implementación:
 - Crear servicios con Spring Boot y Spring Data, Spring for RabbitMQ, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Soporta múltiples vistas desnormalizadas que son escalables y de alto rendimiento.
 - Separación de preocupaciones mejorada = modelos de comando y consulta más simples
 - Ambos subsistemas pueden evolucionar por separado y escalar; el mantenimiento y despliegue puede estar diferenciado.
 - Este patrón tiene los siguientes inconvenientes:
 - Mayor complejidad
 - Duplicación de datos con posibles inconsistencias
 - Replicación con retrasos / eventos inconsistentes
- Patrones relacionados:
 - El patrón de base de datos por servicio crea la necesidad de este patrón.
 - El patrón de API de composición es una solución alternativa.
 - El patrón de eventos del dominio generará los eventos.
 - CQRS se utiliza a menudo en combinación con el patrón Event sourcing

© JMA 2020. All rights reserved

Patrón: API de composición

- Motivación:
 - Se han aplicado los patrones de arquitectura de microservicios y de base de datos por servicio. Como resultado, ya no es sencillo implementar consultas que unen datos de múltiples servicios.
- Intención:
 - ¿Cómo implementar una consulta que recupera datos de múltiples servicios en una arquitectura de microservicio?
- Requisitos:
- Solución:
 - Crear un servicio REST que implemente las consultas como la composición de la invocación de los servicios que poseen los datos y la unión en memoria de los resultados.
- Implementación:
 - Crear servicios con Spring Boot y Spring Data, Spring for RabbitMQ, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Es una forma sencilla de consultar datos en una arquitectura de microservicio.
 - Este patrón tiene los siguientes inconvenientes:
 - Algunas consultas darían resultado ineficaces, combinaciones en memoria de grandes conjuntos de datos.
- Patrones relacionados:
 - El patrón de base de datos por servicio crea la necesidad de este patrón.
 - El patrón CQRS de composición es una solución alternativa.

© JMA 2020. All rights reserved

Estilos de comunicación

SERVICIOS REST

© JMA 2020. All rights reserved

Patrón: Invocación a procedimiento remoto (RPI)

- Motivación:
 - Ha aplicado el patrón de Microservicios. Los servicios deben manejar las solicitudes de los clientes de la aplicación. Además, los servicios a veces deben colaborar para manejar esas solicitudes. Deben utilizar un protocolo de comunicación entre procesos.
- Intención:
 - ¿Cómo se van a comunicar los servicios?
- Requisitos:
 - Los servicios están escritos en diferentes tecnologías
 - Los clientes, a su vez, utilizan diferentes tecnologías. Hay aplicaciones web del lado del cliente (AJAX/WebSokets)
- Solución:
 - Utilizar RPI para la comunicación entre servicios. El cliente utiliza un protocolo basado en solicitud/respuesta para realizar solicitudes a un servicio. Los servicios públicos utilizarán REST.

© JMA 2020. All rights reserved

Patrón: Invocación a procedimiento remoto (RPI)

- Implementación:
 - Crear servicios REST con Spring Boot y Spring MVC con Spring HATEOAS, Spring Data Rest, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Sencillo y familiar
 - El patrón solicitud/respuesta síncrona es fácil
 - Sistema más sencillo ya que no requiere componentes intermediarios.
 - Este patrón tiene los siguientes inconvenientes:
 - Por lo general, solo admite la solicitud/respuesta y no otros patrones de interacción, como notificaciones, solicitud/respuesta asíncrona, publicación/suscripción, publicación/respuesta asíncrona
 - Disponibilidad reducida ya que el cliente y el servicio deben estar disponibles durante la interacción
 - El cliente necesita descubrir ubicaciones de instancias de servicio.
- Patrones relacionados:
 - La mensajería es un patrón alternativo.
 - La configuración externalizada proporciona la ubicación de red (lógica).
 - Un cliente debe utilizar el descubrimiento del lado del cliente y el descubrimiento del lado del servidor para localizar una instancia de servicio
 - Normalmente, un cliente usará el patrón cortocircuito (interruptor automático) para mejorar la confiabilidad

© JMA 2020. All rights reserved

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 hipertexto. 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).

© JMA 2020. All rights reserved

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)

```

POST /server/payment HTTP/1.1 ①
Host: www.myserver.com
Content-Type: application/x-www-form-urlencoded ②
Accept: application/json
Accept-Encoding:gzip,deflate,sdch
Accept-Language:en-US,en;q=0.8
Cache-Control:max-age=0
Connection:keep-alive

orderId=34fry423&payment-method=visa&card-number=2345123423487648&sn=345 ③
  
```

© JMA 2020. All rights reserved

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.

```

HTTP/1.1 201 Created ①
Content-Type: application/json;charset=utf-8
Location: https://www.myserver.com/services/payment/3432
Cache-Control:max-age=21600
Connection:close
Date:Mon, 23 Jul 2012 14:20:19 GMT
ETag:"2e08-3e3073913b100"
Expires:Mon, 23 Jul 2012 20:20:19 GMT

[ ②
  {
    "id": "https://www.myserver.com/services/payment/3432",
    "status": "pending"
  }
]
  
```

© JMA 2020. All rights reserved

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)

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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 responsables de los típicos errores de <i>Página no encontrada</i>
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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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. La única 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.

© JMA 2020. All rights reserved

Estilo de arquitectura

- Métodos GET
 - Normalmente, un método GET correcto devuelve el código de estado HTTP 200 (Correcto). Si no se encuentra el recurso, el método debe devolver HTTP 404 (No encontrado).
- Métodos POST
 - 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. El cuerpo de respuesta contiene una representación del recurso.
 - Si el método realiza algún procesamiento pero 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. O bien, si no hay ningún resultado para devolver, el método puede devolver el código de estado HTTP 204 (Sin contenido) 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). El cuerpo de respuesta puede contener información adicional sobre el error o un vínculo a un URI que proporciona más detalles.
- Métodos DELETE
 - El servidor web debe responder con un 204 (Sin contenido), que indica que la operación de eliminación es correcta, pero que el cuerpo de respuesta no contiene información adicional. Si el recurso no existe, el servidor web puede devolver un 404 (No encontrado).

© JMA 2020. All rights reserved

Estilo de arquitectura

- Métodos PUT
 - Si un método PUT crea un nuevo recurso, devuelve el código de estado HTTP 201 (Creado), al igual que con un método POST. 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, el servidor debe devolver un 400 (Solicitud incorrecta), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) 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 PATCH
 - 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.
 - 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 se admite el formato de documento de revisión devolverá un 415 (Tipo de medio no compatible), si no es posible actualizar el recurso existente devolverá un 409 (Conflicto) o el recurso no existe, puede devolver un 404 (No encontrado).

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 hacen que la solicitud sea condicional.
- Para los métodos GET y HEAD con If-None-Match: si el ETag no coincide con los datos, 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 datos, 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 servicio debe devolver una respuesta HTTP con el código de estado 404 (no encontrado).

© JMA 2020. All rights reserved

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 qué formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

© JMA 2020. All rights reserved

Peticiones

- Request: GET /users
 - accept:application/json
- Response: 200
 - content-type:application/json
 - BODY

- Request: GET /users/11
 - accept:application/json
- Response: 200
 - content-type:application/json
 - BODY

- Request: POST /users
 - accept:application/json
 - content-type:application/json
 - BODY

- Response: 201
 - content-type:application/json
 - BODY

- Request: PUT /users/11
 - accept:application/json
 - content-type:application/json
 - BODY

- Response: 200
 - content-type:application/json
 - BODY

- Request: DELETE /users/11
 - content-type:application/json
 - BODY
- Response: 204 no content

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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
      }
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Políticas de versionado

- El sistema SEMVER (Semantic Versioning), ampliamente adoptado, es un conjunto de reglas para proporcionar un significado claro y definido a las versiones del software.
- La versión SEMVER se compone de 3 números, siguiendo la estructura X.Y.Z, donde:
 - X se denomina Major: indica cambios rupturistas
 - Y se denomina Minor: indica cambios compatibles con la versión anterior
 - Z se denomina Patch: indica resoluciones de bugs (compatibles)
- Básicamente, cuando se arregla un bug se incrementa el patch, cuando se introduce una mejora se incrementa el minor y cuando se introducen cambios que no son compatibles con la versión anterior, se incrementa el major.
- De este modo, cualquier desarrollador sabe qué esperar ante una actualización de su librería favorita. Si sale una actualización donde el major se ha incrementado, sabe que tendrá que ensuciarse las manos con el código para pasar su aplicación existente a la nueva versión.
- Las nuevas versiones del código de las APIs no tienen por qué implicar nuevas versiones de las APIs, solo los cambios rupturistas (major) deberían ser reflejados.

© JMA 2020. All rights reserved

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`

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 Candidates (RC) o revisiones para cada versión mayor y menor. Estas versiones provisionales desaparecerán con el lanzamiento de la versión definitiva.

© JMA 2020. All rights reserved

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, Móvil, ...) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net ...)

© JMA 2020. All rights reserved

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 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);
}
```

© JMA 2020. All rights reserved

RequestMapping

- La anotación `@RequestMapping` permite asignar solicitudes a los métodos 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`

© JMA 2020. All rights reserved

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) {
```

```
    // ...
```

```
}
```

© JMA 2020. All rights reserved

Restricciones

- consumes: 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) {
```

© JMA 2020. All rights reserved

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")
```

© JMA 2020. All rights reserved

Inyección de Parámetros

Anotación	Descripción
@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.

© JMA 2020. All rights reserved

Inyección de Parámetros

Anotación	Descripción
@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.

© JMA 2020. All rights reserved

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();
}
```

© JMA 2020. All rights reserved

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) { ... }
```

© JMA 2020. All rights reserved

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);
```

© JMA 2020. All rights reserved

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 <code>property,property(,ASC DESC)</code> . Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (<code>?sort=firstname&sort=lastname,asc</code>)

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

© JMA 2020. All rights reserved

Mapeo de respuestas genéricas a excepciones.

- Un requisito común para los servicios REST es incluir detalles de error en el cuerpo de la respuesta.
- Spring Framework no lo hace automáticamente porque la representación de los detalles de error en el cuerpo de la respuesta es específica de la aplicación.
- 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 devuelven un `ResponseEntity` que permite establecer el estado y el cuerpo de la respuesta.
- Esto mismo se puede hacer globalmente en clases anotadas con `@ControllerAdvice` que solo tienen los correspondientes métodos `@ExceptionHandler`.

© JMA 2020. All rights reserved

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);  
    }  
}
```

© JMA 2020. All rights reserved

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; }
}
```

© JMA 2020. All rights reserved

@ControllerAdvice

```
@ControllerAdvice
public class ApiExceptionHandler {
    @ResponseStatus(HttpStatus.NOT_FOUND)
    @ExceptionHandler({NotFoundException.class})
    @ResponseBody
    public ErrorMessage notFoundRequest(HttpServletRequest request, Exception exception) {
        return new ErrorMessage(exception.getMessage(), request.getRequestURI());
    }

    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler({ BadRequestException.class, MalformedHeaderException.class, FieldInvalidException.class })
    @ResponseBody
    public ErrorMessage badRequest(Exception exception) {
        return new ErrorMessage(exception.getMessage(), "");
    }
}
```

© JMA 2020. All rights reserved

Servicio Web RESTful

```
import javax.validation.ConstraintViolation;
import javax.validation.Valid;
import javax.validation.Validator;
import javax.websocket.serverPathParam;

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;
```

© JMA 2020. All rights reserved

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 {
        // ...
    }
}
```

© JMA 2020. All rights reserved

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.ACCEPTED)
    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) {
        // ..
    }
}

```

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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, nex, 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("direcciones").withRel("direcciones"));
```

© JMA 2020. All rights reserved

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());
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Spring Data Rest

- Spring Data REST es expone los métodos 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 (\$propertynname,)+[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.

© JMA 2020. All rights reserved

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 delete(Long id);
```

© JMA 2020. All rights reserved

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> {
```

© JMA 2020. All rights reserved

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>

© JMA 2020. All rights reserved

DOCUMENTACIÓN

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

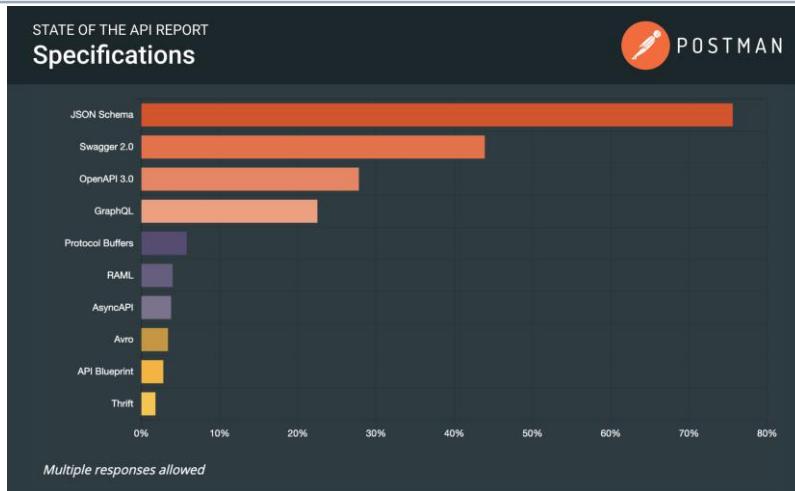
© JMA 2020. All rights reserved

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 máquinas.
- 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.

© JMA 2020. All rights reserved

Especificaciones mas utilizadas



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

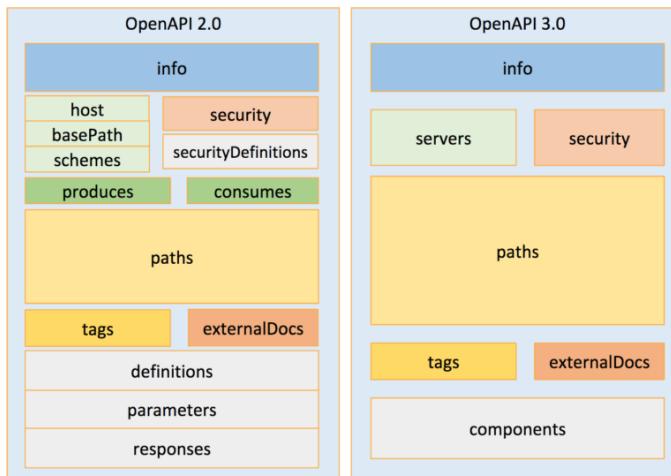
OpenAPI

<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 firma. 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.

© JMA 2020. All rights reserved

Cambio de versión



© JMA 2020. All rights reserved

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 cerrar 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 las 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.

© JMA 2020. All rights reserved

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 (').

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

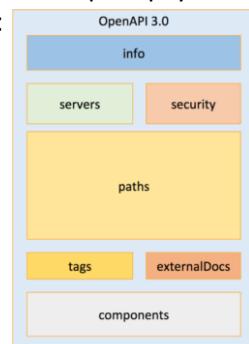
CommonMark de Markdown

- 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).

© JMA 2020. All rights reserved

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



© JMA 2020. All rights reserved

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

```

© JMA 2020. All rights reserved

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: []

```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

Respuestas

- Es obligatoria 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.
- Las posibles respuestas se identifican con el código de respuesta HTTP. Con default se puede definir la 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).

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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 mostrarán ordenadas al azar o según la lógica de las herramientas.

© JMA 2020. All rights reserved

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 dispone 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.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Modelos de entrada y salida

```

components:
schemas:
  Roles:
    type: array
    items:
      $ref: '#/components/schemas/Rol'
  Rol:
    type: object
    description: Roles de usuario
    properties:
      roleId:
        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:
  - roleId
  - name

```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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: []

```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Ecosistema Swagger

- **Swagger Open Source Tools**
 - 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 UI: Generar automáticamente la documentación desde la definición de OpenAPI para la interacción visual y un consumo más fácil.
- **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/>

© JMA 2020. All rights reserved

Swagger: Instalación para Spring Boot

- Se debe añadir la dependencia Maven de springfox-swagger.


```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.9.2</version>
  </dependency>
```
- Se puede añadir swagger-ui para poder ver la parte visual de Swagger:


```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
  </dependency>
```
- Para activar la documentación se anota la aplicación :


```
@EnableSwagger2
@SpringBootApplication
```
- Para acceder a la documentación:
 - <http://localhost:8080/swagger-ui.html> (versión HTML)
 - <http://localhost:8080/v2/api-docs> (versión JSON)

© JMA 2020. All rights reserved

Soporte para JSR-303

- En la versión 2.3.2, se agregó soporte para las anotaciones de validación de bean, específicamente para @NotNull, @Min, @Max y @Size.
- Es necesario incluir la dependencia:

```
<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-bean-validators</artifactId>
<version>2.9.2</version>
</dependency>
```
- Anotar la clase principal con:

```
@EnableSwagger2
@Import(BeanValidatorPluginsConfiguration.class)
```

© JMA 2020. All rights reserved

Soporte para Spring Data Rest

- En la versión 2.6.0, se agregó soporte para Spring Data Rest.
- Es necesario incluir la dependencia:

```
<dependency>
<groupId>io.springfox</groupId>
<artifactId>springfox-data-rest</artifactId>
<version>2.9.2</version>
</dependency>
```
- Anotar la clase principal con:

```
@EnableSwagger2
@Import(SpringDataRestConfiguration.class)
```

© JMA 2020. All rights reserved

Spring Boot: Instalación (v3.0)

- Se debe añadir la dependencia Maven del starter de springfox-swagger.

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

- Establecer el contexto en el fichero application.properties

```
springfox.documentation.swagger-ui.base-url=/apidoc
```

- Para activar la documentación se anota la aplicación :

```
@EnableOpenApi
@SpringBootApplication
```

- Para acceder a la documentación:

- <http://localhost:8080/swagger-ui/index.html> (versión HTML)
- <http://localhost:8080/v3/api-docs> (versión JSON)
- <https://springfox.github.io/springfox/docs/current/#introduction> (framework)

© JMA 2020. All rights reserved

Spring Boot: Soporte adicional (v3.0)

- En la versión 2.3.2, se agregó soporte para las anotaciones de validación de bean JSR-303, específicamente para @NotNull, @Min, @Max y @Size. Es necesario incluir la dependencia:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-beanValidators</artifactId>
    <version>3.0.0</version>
</dependency>
```

- En la versión 2.6.0, se agregó soporte para Spring Data Rest. Es necesario incluir la dependencia:

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-data-rest</artifactId>
    <version>3.0.0</version>
</dependency>
```

© JMA 2020. All rights reserved

Spring Boot: Anotar el modelo

- **@ApiModel:** documenta la entidad, con una descripción corta (value) y una descripción más larga (description).


```
@ApiModelProperty(value = "Entidad Personas", description = "Información completa de la personas")
public class Persona {
```
- **@ApiModelProperty:** documenta las propiedades, con una descripción (value) y si es obligatoria (required). También contiene otros parámetros de posibles valores (allowableValues), no mostrarlo con Swagger (hidden) y otros.


```
@ApiModelProperty(value = "Identificador de la persona", required = true)
private Long id;
```

© JMA 2020. All rights reserved

Spring Boot: Anotar el servicio

- **@Api:** documenta el servicio REST en sí. Va a ser la descripción que salga en el listado, entre otras cosas.


```
@RestController
@Api(value = "Microservice Personas", description = "API que permite el mantenimiento de personas")
public class PersonasResource {
```
- **@ApiOperation:** documenta cada método del servicio.
- **@ApiParam:** documenta los parámetros de cada método del servicio.


```
@GetMapping(path = "/{id}")
ApiOperation(value = "Buscar una persona", notes = "Devuelve una persona por su identificador" )
public Persona getOne(@ApiParam(value = "id" ,required=true) @PathVariable int id) {
```
- **@ApiResponse, @ApiResponse:** documenta las posibles respuestas del método, con un mensaje explicativo.


```
@ApiResponse({
    @ApiResponse(code = 200, message = "Persona encontrada"),
    @ApiResponse(code = 404, message = "Persona no encontrada")
})
```

© JMA 2020. All rights reserved

Spring Boot: Configuración

```
@Configuration
public class SwaggerConfiguration {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
            .select()
            .apis(RequestHandlerSelectors.basePackage("curso.controllers"))
            .paths(PathSelectors.ant("/**"))
            .build()
            .apiInfo(new ApiInfoBuilder()
                .title("Mis microservicios")
                .version("1.0")
                .license("Apache License Version 2.0")
                .contact(new Contact("Yo Mismo", "http://www.example.com", "myeaddress@example.com"))
                .build());
    }
}
```

© JMA 2020. All rights reserved

Spring Boot: Archivo de propiedades

- Hay que crear en resources un archivo de propiedades, por ejemplo, swagger.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:
 @ApiModelProperty(value = "\${person.id.value}", required = true)
- Hay que registrar el archivo de propiedades de la configuración a nivel de clase:
 @PropertySource ("classpath: swagger.properties")

© JMA 2020. All rights reserved

Swagger 2.X Annotations (OpenAPI 3.0)

<https://github.com/swagger-api/swagger-core/wiki/Swagger-2.X---Annotations>

- Para generar la documentación de OpenAPI, swagger-core ofrece un nuevo conjunto de anotaciones para declarar y manipular la salida. La salida principal cumple con la especificación OpenAPI.
- OpenAPI
 - @OpenAPIDefinition: Metadatos generales para una definición de OpenAPI
 - @Info: Metadatos de información para una definición de OpenAPI
 - @Contact: Propiedades para describir la persona de contacto para una definición de OpenAPI
 - @License: Propiedades para describir la licencia para una definición de OpenAPI
 - @Server: Representa servidores para una operación o para la definición de OpenAPI.
 - @Tag: Representa etiquetas para una operación o para la definición de OpenAPI.
- Modelos
 - @Schema: Permite la definición de datos de entrada y salida.
 - @ArraySchema: Permite la definición de datos de entrada y salida para tipos matriz.
 - @Content: Proporciona esquemas y ejemplos para un tipo de medio en particular.

© JMA 2020. All rights reserved

Swagger 2.X Annotations (OpenAPI 3.0)

- Operaciones
 - @Operation: Describe una operación o, por lo general, un método HTTP en una ruta específica.
 - @Parameter: Representa un solo parámetro en una operación de OpenAPI.
 - @RequestBody: Representa el cuerpo de la solicitud en una operación.
 - @ApiResponse: Representa la respuesta en una Operación.
 - @Callback: Describe un conjunto de solicitudes.
 - @Link: Representa un posible vínculo en tiempo de diseño para una respuesta.
- Seguridad
 - @SecurityRequirement: Enumera los esquemas de seguridad necesarios para ejecutar esta operación.
 - @SecurityScheme: Define un esquema de seguridad que puede ser utilizado por las operaciones.
- Extensiones y Otras
 - @Extension: Agrega una extensión con propiedades contenidas
 - @ExtensionPropiedad: Agrega propiedades personalizadas a una extensión
 - @Hidden: Oculta un recurso, una operación o una propiedad
 - @ExternalDocumentation: Proporciona documentación externa a un elemento de definición.

© JMA 2020. All rights reserved

Swagger 2.X Annotations (OpenAPI 3.0)

```

@OpenAPIDefinition(
    info = @Info(
        title = "the title",
        version = "0.0",
        description = "My API",
        license = @License(name = "Apache 2.0", url = "http://foo.bar"),
        contact = @Contact(url = "http://gigantic-server.com", name = "Fred", email = "Fred@gigantic-server.com")
    ),
    tags = {
        @Tag(name = "Tag 1", description = "desc 1", externalDocs = @ExternalDocumentation(description = "docs desc")),
        @Tag(name = "Tag 2", description = "desc 2", externalDocs = @ExternalDocumentation(description = "docs desc 2")),
        @Tag(name = "Tag 3")
    },
    externalDocs = @ExternalDocumentation(description = "definition docs desc"),
    security = {
        @SecurityRequirement(name = "req 1", scopes = {"a", "b"}),
        @SecurityRequirement(name = "req 2", scopes = {"b", "c"})
    },
    servers = {
        @Server(
            description = "server 1",
            url = "http://foo",
            variables = {
                @ServerVariable(name = "var1", description = "var 1", defaultValue = "1", allowableValues = {"1", "2"}),
                @ServerVariable(name = "var2", description = "var 2", defaultValue = "1", allowableValues = {"1", "2"})
            }
        )
    }
)

```

© JMA 2020. All rights reserved

Swagger 2.X Annotations (OpenAPI 3.0)

```

@GET
@Path("/{username}")
@Operation(summary = "Get user by user name",
    responses = {
        @ApiResponse(description = "The user",
            content = @Content(mediaType = "application/json",
                schema = @Schema(implementation = User.class))),
        @ApiResponse(responseCode = "400", description = "User not found")})
public Response getUserByName(
    @Parameter(description = "The name that needs to be fetched. Use user1 for testing. ", required = true) @PathParam("username")
String username)
    throws ApiException {
    User user = userData.findUserByName(username);
    if (null != user) {
        return Response.ok().entity(user).build();
    } else {
        throw new NotFoundException(404, "User not found");
    }
}

```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

Estilos de comunicación

MENSAJERÍA

© JMA 2020. All rights reserved

Patrón: Mensajería

- Motivación:
 - Ha aplicado el patrón de Microservicios. Los servicios deben manejar las solicitudes de los clientes de la aplicación. Además, los servicios a veces deben colaborar para manejar esas solicitudes. Deben utilizar un protocolo de comunicación entre procesos.
- Intención:
 - ¿Cómo se van a comunicar los servicios?
- Requisitos:
 - Los servicios están escritos en diferentes tecnologías
 - Los clientes, a su vez, utilizan diferentes tecnologías
- Solución:
 - Utilizar la mensajería asíncrona para la comunicación entre servicios. Servicios que se comunican mediante el intercambio de mensajes a través de canales de mensajería. El estándar AMQP (Advanced Message Queuing Protocol) es un protocolo estándar abierto en la capa de aplicaciones de un sistema de comunicación. Esta ampliamente difundido y soportado. Sus características son la orientación a mensajes, encolamiento ("queuing"), enrutamiento (tanto punto-a-punto como publicación-suscripción), exactitud y seguridad.

© JMA 2020. All rights reserved

Patrón: Mensajería

- Implementación:
 - Requiere un intermediario de mensajes: RabbitMQ, Apache Kafka
 - Crear servicios con Spring Boot con Spring for RabbitMQ, ...
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Bajo acoplamiento en tiempo de ejecución ya que desacopla el remitente del mensaje del consumidor
 - Disponibilidad mejorada desde que el intermediario de mensajes almacena los mensajes en búfer hasta que el consumidor puede procesarlos
 - Admite una variedad de patrones de comunicación que incluyen solicitud/respuesta síncrona y asíncrona, notificaciones, publicación/suscripción, publicación/respuesta asíncrona, etc.
 - Este patrón tiene los siguientes inconvenientes:
 - Mayor complejidad de los procesos asíncronos frente a los secuenciales síncronos.
 - Complejidad adicional, el intermediario de mensajes debe ser otro componente del sistema que debe instalarse y configurarse. También deberá ser replicado para disponibilidad y capacidad.
- Patrones relacionados:
 - La Invocación a procedimiento remoto (RPI) mensajería es un patrón alternativo.
 - La configuración externalizada proporciona los nombres de los canales de mensajes (lógicos) y la ubicación del intermediario de mensajes.
 - El patrón Saga y el patrón CQRS usan mensajes
 - El patrón de Bandeja de salida transaccional permite que los mensajes se envíen como parte de una transacción de base de datos

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 ZeroMQ.
- 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.

© JMA 2020. All rights reserved

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 suscribirse 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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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.



© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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”.

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 receive(MessageDTO inMsg) {
    // Procesar el mensaje recibido: inMsg
}
```

© JMA 2020. All rights reserved

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;
}
```

© JMA 2020. All rights reserved

CONSULTAS ENTRE SERVICIOS

© JMA 2020. All rights reserved

Patrón: Registro de servicios

- Motivación:
 - Los clientes de un servicio utilizan el descubrimiento del lado del cliente o el descubrimiento del lado del servidor para determinar la ubicación de una instancia de servicio a la que enviar las solicitudes.
- Intención:
 - ¿Cómo saben los clientes de un servicio (en el caso del descubrimiento del lado del cliente) y / o enrutadores (en el caso del descubrimiento del lado del servidor) acerca de las instancias disponibles de un servicio?
- Requisitos:
 - Cada instancia de un servicio expone una API remota como HTTP/REST o Thrift, etc. en una ubicación particular (host y puerto)
 - El número de instancias de servicios y sus ubicaciones cambia dinámicamente. A las máquinas virtuales y los contenedores se les suele asignar una dirección IP dinámica. Un grupo de autoescalamiento de EC2, por ejemplo, ajusta el número de instancias en función de la carga.
- Solución:
 - Implementar un registro de servicios, que es una base de datos de servicios, sus instancias y sus ubicaciones. Las instancias de servicio se registran con el registro de servicios en el inicio y se cancelan en el cierre. El cliente del servicio y / o los enrutadores consultan el registro de servicios para encontrar las instancias disponibles de un servicio. Un registro de servicios puede invocar la API de comprobación de estado de una instancia de servicio para verificar que es capaz de manejar solicitudes

© JMA 2020. All rights reserved

Patrón: Registro de servicios

- **Implementación:**
 - Crear un servidor Eureka con Spring Boot y Spring Cloud
- **Consecuencias:**
 - Los beneficios del patrón de registro de servicios incluyen:
 - El cliente del servicio y / o los enruteadores pueden descubrir la ubicación de las instancias de servicio.
 - También hay algunos inconvenientes:
 - A menos que el registro de servicios esté integrado en la infraestructura, es otro componente de la infraestructura que debe que debe instalarse, configurarse y mantenerse.
 - Además, el registro de servicios es un componente crítico del sistema, debe estar altamente disponible.
 - Si bien los clientes deben almacenar en caché los datos proporcionados por el registro de servicios, si el registro de servicios falla, dichos datos eventualmente quedarán desactualizados.

© JMA 2020. All rights reserved

Patrón: Registro de servicios

- **Consecuencias:**
 - Se debe decidir cómo se registran las instancias de servicio con el Registro de servicios. Hay dos opciones:
 - Patrón de auto registro: las instancias de servicio se registran a sí mismas.
 - Patrón de registro de terceros : un tercero registra las instancias de servicio con el Registro de servicios.
 - Los clientes del registro de servicios necesitan conocer las ubicaciones de las instancias del registro de servicios. Las instancias de registro de servicios deben implementarse en direcciones IP fijas y conocidas. Los clientes se configuran con esas direcciones IP.
 - Por ejemplo, las instancias de servicio de Netflix Eureka se implementan normalmente utilizando direcciones IP elásticas. El grupo disponible de direcciones IP elásticas se configura mediante un archivo de propiedades o mediante DNS. Cuando se inicia una instancia de Eureka, consulta la configuración para determinar qué dirección IP elástica disponible usar. Un cliente Eureka también está configurado con el conjunto de direcciones IP elásticas.
- **Patrones relacionados:**
 - Descubrimiento del lado del cliente y el descubrimiento del lado del servidor crean la necesidad de un registro de servicios
 - El registro automático y el registro de terceros son dos formas diferentes en que las instancias de servicio pueden registrarse con el registro de servicio
 - Health Check API : el registro de servicios invoca la API de control de salud de una instancia de servicio para verificar que puede manejar solicitudes

© JMA 2020. All rights reserved

Patrón: Auto registro

- Motivación:
 - Ha aplicado el patrón de descubrimiento de servicios del lado del cliente o el patrón de descubrimiento de servicios del lado del servidor. Las instancias de servicio se deben registrar en el registro de servicios al inicio para que puedan descubrirse y anularse el registro en el cierre.
- Intención:
 - ¿Cómo se registran y anulan del registro de servicios las instancias de servicio?
- Requisitos:
 - Las instancias de servicio se deben registrar con el registro de servicios en el inicio y anular el registro en el cierre
 - Las instancias de servicio que se cuelgan se deben anular del registro de servicios
 - Las instancias de servicio que se ejecutan pero que no pueden manejar las solicitudes deben ser anuladas del registro de servicios
- Solución:
 - Una instancia de servicio es responsable de registrarse en el registro de servicios. Al iniciarse, la instancia de servicio se registra a sí misma (host y dirección IP) con el registro de servicios y queda disponible para su descubrimiento. El cliente normalmente debe renovar periódicamente su registro para que el servidor de registro sepa que aún está vivo. Al cerrarse, la instancia de servicio se anula el registro del servicio.

© JMA 2020. All rights reserved

Patrón: Auto registro

- Implementación:
 - Incluir y configurar el cliente de Eureka en los microservicios
- Consecuencias:
 - Los beneficios del patrón de auto registro incluyen los siguientes:
 - Una instancia de servicio conoce su propio estado, por lo que puede implementar un modelo de estado más complejo que ARRIBA / ABAJO, por ejemplo, INICIAR, DISPONIBLE, ...
 - También hay algunos inconvenientes:
 - Acopla el servicio al registro de servicios.
 - Debe implementar la lógica de registro de servicios en cada tecnología que se utilice para escribir sus servicios, por ejemplo, NodeJS / JavaScript, Java, .NET, etc.
 - Una instancia de servicio que se está ejecutando pero que no puede manejar las solicitudes a menudo carecerá de la autoconciencia para darse de baja del registro de servicios.
- Patrones relacionados:
 - Registro de servicios: una parte esencial del descubrimiento de servicios
 - Descubrimiento del lado del cliente: una forma en que se descubre una instancia de servicio
 - Descubrimiento del lado del servidor: otra forma en que se descubre una instancia de servicio
 - Chasis de microservicio: el auto registro es responsabilidad del marco del chasis de microservicio
 - El registro de terceros es una solución alternativa

© JMA 2020. All rights reserved

Patrón: Registro de terceros

- Motivación:
 - Ha aplicado el patrón de descubrimiento de servicios del lado del cliente o el patrón de descubrimiento de servicios del lado del servidor. Las instancias de servicio se deben registrar en el registro de servicios al inicio para que puedan descubrirse y anularse el registro en el cierre.
- Intención:
 - ¿Cómo se registran y anulan del registro de servicios las instancias de servicio?
- Requisitos:
 - Las instancias de servicio se deben registrar con el registro de servicios en el inicio y anular el registro en el cierre
 - Las instancias de servicio que se cuelgan se deben anular del registro de servicios
 - Las instancias de servicio que se ejecutan pero que no pueden manejar las solicitudes deben ser anuladas del registro de servicios
- Solución:
 - Un registrador externo es responsable de registrar y anular el registro de una instancia de servicio en el registro de servicios. Cuando se inicia la instancia de servicio, el registrador registra la instancia de servicio con el registro de servicios. Cuando la instancia de servicio se apaga, el registrador anula el registro de la instancia de servicio del registro de servicios.

© JMA 2020. All rights reserved

Patrón: Registro de terceros

- Implementación:
 - Netflix Prana: una aplicación de "side car" que se ejecuta junto con una aplicación que no es JVM y registra la aplicación con Eureka.
 - AWS AutoScaling Groups: (des)registran automáticamente las instancias EC2 con Elastic Load Balancer
 - Joyent's Container o Registrar: (des)registran el registro de contenedores Docker
 - Los marcos de clústeres como Kubernetes y Marathon (des)registran instancias de servicio con el registro integrado / implícito
- Consecuencias:
 - Los beneficios del patrón de registro de terceros incluyen:
 - No requiere código en el servicio como cuando se usa el patrón de registro automático, ya que no es responsable de registrarse.
 - El registrador puede realizar comprobaciones de estado en una instancia de servicio y registrar / anular el registro de la instancia basándose en la comprobación de estado
 - También hay algunos inconvenientes:
 - El registrador externo solo puede tener un conocimiento superficial del estado de la instancia de servicio, por ejemplo, EN EJECUCIÓN O NO EN EJECUCIÓN y, por lo tanto, puede no saber si puede manejar solicitudes. Sin embargo, algunos registradores como Netflix Prana realizan una verificación de estado para determinar la disponibilidad de la instancia de servicio.
 - A menos que el registrador sea parte de la infraestructura, es otro componente que debe instalarse, configurarse y mantenerse. Además, como es un componente crítico del sistema, debe estar altamente disponible.
- Patrones relacionados:
 - Registro de servicios
 - Descubrimiento del lado del cliente
 - Descubrimiento del lado del servidor
 - Auto registro es una solución alternativa

© JMA 2020. All rights reserved

Eureka

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

© JMA 2020. All rights reserved

Eureka Server

- Añadir al proyecto:
 - Spring Boot + Cloud Discovery: Eureka Server + Core: Cloud Bootstrap
- Anotar aplicación:


```
@EnableEurekaServer
@SpringBootApplication
public class MsEurekaServiceDiscoveryApplication {
```
- Configurar:


```
#Servidor Eureka Discovery Server
eureka.instance.hostname: localhost
eureka.client.registerWithEureka: false
eureka.client.fetchRegistry: false
server.port: ${PORT:8761}
```
- Arrancar servidor
- Acceder al dashboard de Eureka: <http://localhost:8761/>

© JMA 2020. All rights reserved

Auto registro de servicios

- Añadir al proyecto:
 - Eureka Discovery, Cloud Bootstrap
- Anotar aplicación:


```
@EnableEurekaClient
@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/
server.port: ${PORT:8001}
```
- Arrancar microservicio
- Refrescar dashboard de Eureka:
 - <http://localhost:8761/>

© JMA 2020. All rights reserved

Patrón: Descubrimiento del lado del cliente

- Motivación:
 - Los servicios normalmente necesitan llamarse unos a otros. En una aplicación monolítica, los servicios se invocan unos a otros a través de un método de nivel de lenguaje o llamadas a procedimientos. En una implementación de sistema distribuido tradicional, los servicios se ejecutan en ubicaciones fijas y bien conocidas (hosts y puertos) y, por lo tanto, pueden llamarse fácilmente utilizando HTTP / REST o algún mecanismo RPC. Sin embargo, una aplicación moderna basada en microservicios generalmente se ejecuta en entornos virtualizados o en contenedores, donde la cantidad de instancias de un servicio y sus ubicaciones cambian dinámicamente.
- Intención:
 - ¿Cómo descubre el cliente de un servicio, la puerta de enlaces API u otro servicio, la ubicación de una instancia de servicio?
- Requisitos:
 - Cada instancia de un servicio expone una API remota como HTTP / REST o Thrift, etc. en una ubicación particular (host y puerto)
 - El número de instancias de servicios y sus ubicaciones cambia dinámicamente.
 - Las máquinas virtuales y los contenedores suelen tener asignadas direcciones IP dinámicas.
 - El número de instancias de servicios puede variar dinámicamente. Por ejemplo, un grupo de autoescalado de EC2 ajusta el número de instancias en función de la carga.

© JMA 2020. All rights reserved

Patrón: Descubrimiento del lado del cliente

- Solución:
 - Al realizar una solicitud a un servicio, el cliente obtiene la ubicación de una instancia de servicio consultando un registro de servicios, que conoce las ubicaciones de todas las instancias de servicio.
- Implementación:
 - Crear un cliente con Spring Boot y Spring Cloud: Cliente Eureka, balanceo de carga (Ribbon), cliente Rest (RestTemplate o Feign)
- Consecuencias:
 - El descubrimiento del lado del cliente tiene los siguientes beneficios:
 - Menos partes móviles y saltos de red en comparación con el descubrimiento del lado del servidor
 - El descubrimiento del lado del cliente también tiene los siguientes inconvenientes:
 - Este patrón une al cliente con el registro de servicios.
 - Debe implementar la lógica de descubrimiento en cada tecnología que se utilice para escribir los servicios, por ejemplo, NodeJS / JavaScript, Java, .NET, etc. Por ejemplo, Netflix Prana proporciona un enfoque basado en proxy HTTP para el descubrimiento de servicios para clientes que no son JVM.
- Patrones relacionados:
 - Registro de servicios: una parte esencial del descubrimiento de servicios
 - Chasis de microservicio: el descubrimiento del servicio del lado del cliente es responsabilidad del marco del chasis de microservicio
 - Descubrimiento del lado del servidor es una solución alternativa

© JMA 2020. All rights reserved

Patrón: Descubrimiento del lado del servidor

- Motivación:
 - Los servicios normalmente necesitan llamarse unos a otros. En una aplicación monolítica, los servicios se invocan unos a otros a través de un método de nivel de idioma o llamadas a procedimientos. En una implementación de sistema distribuido tradicional, los servicios se ejecutan en ubicaciones fijas y bien conocidas (hosts y puertos) y, por lo tanto, pueden llamarse fácilmente utilizando HTTP / REST o algún mecanismo RPC. Sin embargo, una aplicación moderna basada en microservicios generalmente se ejecuta en entornos virtualizados o en contenedores, donde la cantidad de instancias de un servicio y sus ubicaciones cambian dinámicamente.
- Intención:
 - ¿Cómo descubre el cliente de un servicio, la puerta de enlaces API u otro servicio, la ubicación de una instancia de servicio?
- Requisitos:
 - Cada instancia de un servicio expone una API remota como HTTP / REST o Thrift, etc. en una ubicación particular (host y puerto)
 - El número de instancias de servicios y sus ubicaciones cambia dinámicamente.
 - Las máquinas virtuales y los contenedores suelen tener asignadas direcciones IP dinámicas.
 - El número de instancias de servicios puede variar dinámicamente. Por ejemplo, un grupo de autoescalado de EC2 ajusta el número de instancias en función de la carga.

© JMA 2020. All rights reserved

Patrón: Descubrimiento del lado del servidor

- Solución:
 - Al realizar una solicitud a un servicio, el cliente realiza una solicitud a través de un enrutador (también conocido como equilibrador de carga) que se ejecuta en una ubicación bien conocida. El enrutador consulta un registro de servicios , que podría estar integrado en el enrutador, y reenvia la solicitud a una instancia de servicio disponible.
- Implementación:
 - AWS Elastic Load Balancer (ELB) o algunas solución de agrupación en clústeres, como Kubernetes, Marathon o Azure Service Fabric, que ejecutan un proxy en cada host que funciona como un enrutador de descubrimiento del lado del servidor.
 - Crear un cliente con Spring Boot y Spring Cloud: cliente Rest (RestTemplate o Feign)
- Consecuencias:
 - El descubrimiento del lado del servidor tiene los siguientes beneficios:
 - En comparación con el descubrimiento del lado del cliente, el código del cliente es más simple, ya que no tiene que lidiar con el descubrimiento. En su lugar, un cliente simplemente hace una solicitud al enrutador
 - Algunos entornos de nube proporcionan esta funcionalidad, por ejemplo, AWS Elastic Load Balancer.
 - El descubrimiento del lado del servidor también tiene los siguientes inconvenientes:
 - A menos que sea parte del entorno de la nube, el enrutador debe ser otro componente del sistema que debe instalarse y configurarse. También deberá ser replicado para disponibilidad y capacidad.
 - El enrutador debe admitir los protocolos de comunicación necesarios (por ejemplo, HTTP, gRPC, Thrift, etc.) a menos que sea un enrutador basado en TCP
 - Se requieren más saltos de red que al usar el descubrimiento del lado del cliente y se tiene menos control.
- Patrones relacionados:
 - Registro de servicios: una parte esencial del descubrimiento de servicios
 - Descubrimiento del lado del cliente es una solución alternativa

© JMA 2020. All rights reserved

RestTemplate

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


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


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

© JMA 2020. All rights reserved

RestTemplate

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

© JMA 2020. All rights reserved

RestTemplate

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

© JMA 2020. All rights reserved

RestTemplate

- Para recuperar uno:

```
PersonaDTO rsIt = srvRest.getForObject(
    "http://localhost:8080/api/personas/{id}", PersonaDTO.class, 1);
```

- Para recuperar todos (si no se dispone de una implementación de List<PersonaDTO>):

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

© JMA 2020. All rights reserved

RestTemplate

- Para crear o modificar un recurso:

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

- Para crear o modificar un recurso con identificador:

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

- Para borrar un recurso con identificador:

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

© JMA 2020. All rights reserved

RestTemplate

- De forma predeterminada, RestTemplate lanzará una de estas excepciones en caso de un error de HTTP:
 - HttpClientErrorException: en estados HTTP 4xx
 - HttpServerErrorException: en estados HTTP 5xx
 - 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;
```

© JMA 2020. All rights reserved

LinkDiscoverers

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


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


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

© JMA 2020. All rights reserved

Feign

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

© JMA 2020. All rights reserved

Feign

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

© JMA 2020. All rights reserved

Spring Cloud LoadBalancer

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

© JMA 2020. All rights reserved

Spring Cloud LoadBalancer

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

© JMA 2020. All rights reserved

Cientes Load Balancer

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

```
@LoadBalanced
@Bean RestTemplate restTemplate() { return new RestTemplate(); }
@LoadBalanced
@Bean public WebClient.Builder webClientBuilder() { return WebClient.builder(); }
```

- Las peticiones sustituyen en la URL el nombre del dominio por el nombre registrado en el servidor de descubrimiento para que utilicen el balanceo de carga:

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

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

© JMA 2020. All rights reserved

Patrón: API Gateway

- Motivación:
 - Ha aplicado el patrón de arquitectura de Microservicios, necesita desarrollar varias versiones de la interfaz de usuario:
 - UI basada en JavaScript / HTML5 para navegadores de escritorio y móviles (SPA): el navegador del usuario interactúa con el servidor a través de peticiones AJAX a las API REST
 - Clientes nativos de Android y iPhone: estos clientes interactúan con el servidor a través de las API REST
 - Dado que se utiliza el patrón de arquitectura de Microservicios, los datos a mostrar y mantener se distribuyen en múltiples servicios.
- Intención:
 - ¿Cómo acceden los clientes de una aplicación basada en microservicios a los servicios individuales?
- Requisitos:
 - La granularidad de las API proporcionadas por los microservicios (APIs detalladas) a menudo es diferente de lo que necesita un cliente, lo que implica que los clientes necesitan interactuar con múltiples servicios.
 - Diferentes clientes necesitan diferentes datos. Por ejemplo, la versión del navegador de escritorio de una página suele ser más elaborada que la versión móvil.
 - El rendimiento de la red es diferente para los diferentes tipos de clientes: una red móvil suele ser mucho más lenta y tiene una latencia mucho mayor que una red no móvil, cualquier WAN es mucho más lenta que una LAN. Esto significa que un cliente móvil nativo usa una red que tiene características de rendimiento muy diferentes a las de una LAN utilizada por una aplicación web del lado del servidor. La aplicación web del lado del servidor puede realizar múltiples solicitudes a los servicios de backend sin afectar la experiencia del usuario, donde un cliente móvil solo puede realizar algunas.
 - La cantidad de instancias de servicio y sus ubicaciones (host + puerto) cambian dinámicamente
 - La partición en servicios puede cambiar con el tiempo y debe ocultarse a los clientes
 - Los servicios pueden usar un conjunto diverso de protocolos, algunos de los cuales pueden no ser compatibles con la web

© JMA 2020. All rights reserved

Patrón: API Gateway

- Solución:
 - Implementar una puerta de enlaces API que sea el único punto de entrada para todos los clientes.
 - La puerta de enlaces API maneja las solicitudes de una de dos maneras.
 - Algunas solicitudes simplemente se envían por proxy / enrutan al servicio apropiado.
 - Otras solicitudes se manejan mediante la distribución a múltiples servicios.
 - La puerta de enlaces API puede exponer una API diferente para cada tipo de cliente, en lugar de proporcionar una API de estilo único para todos: un adaptador que proporciona a cada tipo de cliente la API que mejor se adapta a sus requisitos.
 - Backends para frontends es una variación de este patrón que define un API Gateway separado para cada tipo de cliente (web, móviles, terceros, ...).
 - La puerta de enlaces API también puede implementar balanceo de carga y seguridad, por ejemplo, para verificar que el cliente esté autorizado para realizar la solicitud
- Implementación:
 - Crear un servidor Zuul o Spring Cloud Gateway con Spring Boot y Spring Cloud

© JMA 2020. All rights reserved

Patrón: API Gateway

- Consecuencias:
 - La puerta de enlaces API tiene los siguientes beneficios:
 - Aísla a los clientes de cómo se partitiona la aplicación en microservicios
 - Aísla a los clientes del problema de determinar las ubicaciones de las instancias de servicio
 - Proporciona la API óptima para cada cliente
 - Simplifica el cliente moviendo la lógica para llamar múltiples servicios desde el cliente a la puerta de enlaces API, lo que reduce el número de peticiones: un solo viaje de ida y vuelta para recuperar datos de múltiples servicios. Menos solicitudes implica menos gastos generales y mejora la experiencia del usuario. Una puerta de enlaces API es esencial para las aplicaciones móviles.
 - Puede traducir de un protocolo API público estándar y fácil de usar a cualquier protocolo que se use internamente.
 - La puerta de enlaces API también tiene los siguientes inconvenientes:
 - Mayor complejidad: la puerta de enlaces API es otro componente que debe desarrollarse, implementarse y administrarse
 - Aumento del tiempo de respuesta debido al salto de red adicional a través de la puerta de enlaces API. Sin embargo, para la mayoría de las aplicaciones, el costo de un viaje de ida y vuelta adicional es insignificante.
- Patrones relacionados:
 - La puerta de enlaces API debe usar uno de descubrimiento del servicio
 - Circuit Breaker para invocar servicios
 - Access token, si implementa seguridad.
 - API Composition, si unifica las llamadas a múltiples servicios en una.

© JMA 2020. All rights reserved

Spring Cloud Gateway

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

© JMA 2020. All rights reserved

Spring Cloud Gateway

- Añadir proyecto:
 - Spring Cloud Routing Gateway, Eureka Discovery Client
- Anotar la aplicación:

```
@EnableDiscoveryClient  
@EnableEurekaClient
```
- Configurar:

```
eureka:  
  client:  
    fetchRegistry: true  
    registerWithEureka: false  
    serviceUrl:  
      defaultZone: ${DISCOVERY_URL:http://localhost:8761}/eureka/  
  instance:  
    appname: apigateway-server  
  server:  
    port: ${PORT:8080}
```

© JMA 2020. All rights reserved

Spring Cloud Gateway

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

© JMA 2020. All rights reserved

Spring Cloud Gateway

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

© JMA 2020. All rights reserved

Spring Reactive Web

SPRING WEBFLUX

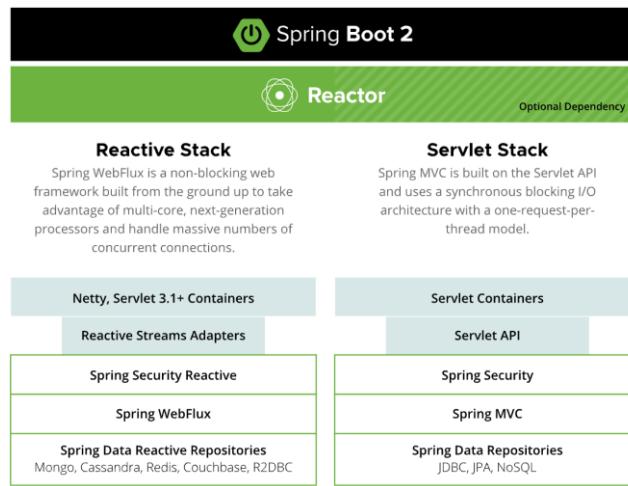
© JMA 2020. All rights reserved

Introducción

- El marco web original incluido en Spring Framework, Spring Web MVC, fue diseñado específicamente para la API de Servlet y los contenedores de Servlet. El marco web de pila reactiva, Spring WebFlux, se agregó más adelante en la versión 5.0 con soporte reactivo no bloqueante con contrapresión.
- Una de las principales razones por las que los desarrolladores pasan del código bloqueante al código no bloqueante es la eficiencia. El código reactivo hace más trabajo con menos recursos. Project Reactor y Spring WebFlux permiten a los desarrolladores aprovechar los procesadores multinúcleo de próxima generación, que manejan cantidades potencialmente masivas de conexiones simultáneas. Con el procesamiento reactivo, puede satisfacer a más usuarios simultáneos con menos instancias de microservicio.
- Es importante acceder a los datos y procesarlos de forma reactiva. MongoDB, Redis y Cassandra tienen soporte reactivo nativo en Spring Data. Muchas bases de datos relacionales (Postgres, Microsoft SQL Server, MySQL, H2 y Google Spanner) tienen soporte reactivo a través del proyecto Reactive Relational Database Connectivity (R2DBC). En el mundo de la mensajería, Spring Cloud Stream también admite el acceso reactivo a plataformas como RabbitMQ y Kafka.
- Así mismo, deben ejecutarse en servidores sin bloqueo, como los contenedores Netty, Tomcat, Jetty o Undertow, o que tengan soporte para Servlet 3.1+. Esto y disponer del acceso reactivo a base de datos limita los escenarios de uso.

© JMA 2020. All rights reserved

Stack



© JMA 2020. All rights reserved

Reactor

- Project Reactor es una base totalmente no bloqueante con soporte de contrapresión incluido. Es la base de la pila reactiva en el ecosistema Spring y se presenta en proyectos como Spring WebFlux, Spring Data y Spring Cloud Gateway.
- Proporciona los tipos de API Mono y Flux para trabajar en secuencias de datos de 0..1 (Mono) y 0..N (Flux) a través de un rico conjunto de operadores alineados con el vocabulario de operadores de ReactiveX.
- Reactor es una biblioteca de Flujos reactivos y, por lo tanto, todos sus operadores admiten contrapresión sin bloqueo. Reactor tiene un fuerte enfoque en Java del lado del servidor. Se desarrolla en estrecha colaboración con Spring.

© JMA 2020. All rights reserved

Reactivo no bloqueante con contrapresión

- El término "reactivo" se refiere a modelos de programación que se construyen en torno a la reacción al cambio: componentes de red que reaccionan a eventos de E / S, controladores de UI que reaccionan a eventos de ratón o teclado y otros. En ese sentido, el no bloqueo es reactivo, porque, en lugar de estar bloqueado para esperar, ahora estamos en el modo de reaccionar a las notificaciones cuando las operaciones se completan o los datos están disponibles.
- También hay otro mecanismo importante que asociamos con "reactivo" y es la contrapresión sin bloqueo, contra la presión de demasiadas invocaciones que produzcan una saturación. En el código imperativo sincrónico, el bloqueo de llamadas sirve como una forma natural de contrapresión que obliga a la persona que llama a esperar. En el código sin bloqueo, se vuelve importante controlar la tasa de eventos para que un productor rápido no abrume su destino.

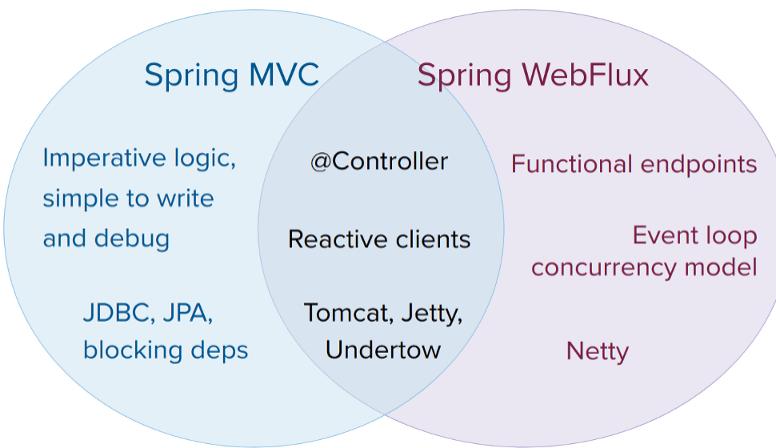
© JMA 2020. All rights reserved

Modelos de programación

- Spring WebFlux ofrece dos modelos de programación:
 - Controladores anotados: coherentes con Spring MVC y basados en las mismas anotaciones del módulo Spring MVC. Tanto los controladores Spring MVC como WebFlux admiten tipos de retorno reactivos (Reactor y RxJava). WebFlux también admite @RequestBody y argumentos reactivos.
 - Puntos finales funcionales: modelo de programación funcional, ligero y basado en Lambda. Puede pensar en esto como una pequeña biblioteca o un conjunto de utilidades que una aplicación puede usar para enrutar y manejar solicitudes. La gran diferencia con los controladores anotados es que la aplicación está a cargo del manejo de solicitudes de principio a fin en lugar de declarar la intención a través de anotaciones y ser devuelto a la llamada.

© JMA 2020. All rights reserved

Spring MVC vs Spring WebFlux



© JMA 2020. All rights reserved

Controladores anotados

- Spring WebFlux proporciona un modelo de programación basado en anotación, donde los componentes utilizan anotaciones `@Controller` y `@RestController` para expresar asignaciones de petición, de entrada solicitud, excepciones y demás. Los controladores anotados tienen firmas de métodos flexibles y no tienen que extender clases base ni implementar interfaces específicas.
- Permite definir el mapeo con las anotaciones `@RequestMapping`, `@GetMapping`, `@PostMapping`, `@PutMapping`, `@DeleteMapping` y `@PatchMapping` siguiendo las convenciones de patrones de Spring MVC.
- Así mismo, soporta la inyección de parámetros con las mismas anotaciones de Spring MVC.
- Es en el `@ResponseBody` donde aparecen la diferencias al admitir los tipos reactivos:
 - `Flux<T>`, `Mono<T>`, `ResponseEntity<Mono<T>>` o `Mono<ResponseEntity<T>>`, `Observable<ServerSentEvent>`, `Flux<ServerSentEvent>` y otros tipos reactivos.

© JMA 2020. All rights reserved

Respuesta asincronas

- Mono<T>: Respuesta de tipo reactivo de valor único.
- Flux<T>: Respuesta de tipo reactivo de valor multiple.
- ResponseEntity<Mono<T>>o ResponseEntity<Flux<T>>: hace que el estado de la respuesta y los encabezados se conozcan de inmediato, mientras que el cuerpo se proporciona de forma asincrónica en un momento posterior.
- Mono<ResponseEntity<T>>: proporciona las respuesta: estado de respuesta, encabezados y cuerpo, de forma asincrónica en un momento posterior. Esto permite que el estado de respuesta y los encabezados varíen según el resultado del manejo de solicitudes asincrónicas.
- Mono<ResponseEntity<Mono<T>>>o Mono<ResponseEntity<Flux<T>>>: son otra alternativa posible, aunque menos común: proporcionan el estado de respuesta y los encabezados de forma asincrónica primero y luego el cuerpo de la respuesta, también de forma asincrónica, en segundo lugar.

© JMA 2020. All rights reserved

Flux

- A Flux<T> es un Publisher<T> estándar que representa una secuencia asíncrona de 0 a N elementos emitidos, opcionalmente terminado por una señal de finalización o un error. Al igual que en la especificación Reactive Streams, estos tres tipos de señales se traducen en llamadas a los métodos onNext, onComplete y onError de un suscriptor del flujo.
- Con esta amplia gama de posibles señales, Flux es el tipo reactivo de uso general. Tenga en cuenta que todos los eventos, incluso los que terminan, son opcionales: ningún evento onNext, pero un evento onComplete representa una secuencia finita vacía, pero si se elimina el onComplete se tendrá una secuencia vacía infinita (no es particularmente útil, excepto para las pruebas de cancelación). De manera similar, las secuencias infinitas no están habitualmente vacías. Por ejemplo, Flux.interval(Duration) produce un Flux<Long> infinito y emite ticks regulares desde un reloj.

© JMA 2020. All rights reserved

Mono

- A Mono<T> es una especialización de Publisher<T> que emite como máximo un elemento a través de la señal onNext y luego termina con una señal onComplete (éxito, con o sin valor), o solo emite una sola señal onError (fallido).
- En la mayoría de las implementaciones Mono se espera que se llame inmediatamente a onComplete en su Subscriber después de haber llamado onNext. Mono.never() es un valor atípico: no emite ninguna señal (infinito), lo que técnicamente no está prohibido aunque no es muy útil fuera de las pruebas. Por otro lado, una combinación de onNext y onError está explícitamente prohibida.
- Mono ofrece solo un subconjunto de los operadores que están disponibles para a Flux, y algunos operadores (en particular los que combinan el Mono con otro Publisher) lo cambian a un Flux. Por ejemplo, Mono#concatWith(Publisher) devuelve un Flux mientras Mono#then(Mono) devuelve otro Mono.
- Hay que tener en cuenta que puede utilizar un Mono para representar procesos asincrónicos sin valor, que solo tienen el concepto de finalización (similar a un Runnable). Para crear uno, se puede utilizar un Mono<Void>.

© JMA 2020. All rights reserved

Creación de tipos reactivos

- La forma más sencilla de empezar con Flux y Mono es utilizar uno de los numerosos métodos de fábrica que se encuentran en sus respectivas clases: just, empty, range,


```
Mono<String> noData = Mono.empty(), data = Mono.just("foo");
Flux<Integer> numbers = Flux.range(5, 3);
Flux<String> seq = Flux.fromIterable(list);
```
- La forma más simple de creación programática de un Flux (síncrono y uno por uno) es a través del método generate, que toma una función de generador.


```
Flux<String> flux = Flux.generate(
    () -> 0, // Suministramos el valor de estado inicial de 0.
    (estado, suscriptor) -> {
        suscriptor.next("3 x " + estado + " = " + 3* estado); // suministramos el siguiente valor
        if (estado == 10) suscriptor.complete(); // indicamos que se ha completado la secuencia.
        return estado + 1;
    });

```
- Los métodos push (asíncrono y monoproceso) y créate (asíncrono y multiproceso) son formas más avanzada de creación programática de un Flux.

© JMA 2020. All rights reserved

Suscriptores

- Cuando se trata de suscribirse a Flux y Mono se recomienda hacer uso de las lambdas de Java 8. El método .subscribe() tiene una amplia variedad de sobrecargas para diferentes combinaciones de devoluciones de llamada:

```
.subscribe();
.subscribe(Consumer<? super T> consumer);
.subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer);
.subscribe(Consumer<? super T> consumer, Consumer<? super Throwable> errorConsumer,
Runnable completeConsumer);
```

```
Flux<Integer> pares = Flux.just(2, 4, 6, 8, 10);
ints.subscribe(i -> System.out.println(i),
    error -> System.err.println("Error " + error),
    () -> System.out.println("Done"));
```

- Todas estas variantes basadas en lambda devuelven Disposable. La interfaz Disposable representa el hecho de que la suscripción se puede cancelar, llamando a su método dispose().

```
ints.dispose()
```

© JMA 2020. All rights reserved

Suscriptores

- Como alternativa a las lambdas se pueden utilizar instancias de herederos de BaseSubscriber, que implementen como métodos las diferentes acciones. Las instancias de BaseSubscriber(o subclases de él) son de un solo uso, lo que significa que si un BaseSubscriber cancela su suscripción al primer Publisher también lo hace a un segundo Publisher si está suscrito.

```
public class SampleSubscriber<T> extends BaseSubscriber<T> {
    public void hookOnSubscribe(Subscription subscription) {
        System.out.println("Subscribed");
        request(1);
    }
    public void hookOnNext(T value) {
        System.out.println(value);
        request(1);
    }
}
SampleSubscriber<Integer> mySubscriber = new SampleSubscriber<Integer>();
Flux<Integer> ints = Flux.range(1, 4);
ints.subscribe(mySubscriber);
```

© JMA 2020. All rights reserved

Operadores

- Flux y Mono suministran métodos adicionales para permitir la manipulación sofisticada de los flujos:
 - Transformación: transform, cast, collect, collectList, collectMap, concatMap, map, scan
 - Combinación: combineLatest, concat, concatMap, concatWith, merge, join,
 - Utilidad: delay, delayElements, replay, retry,
 - Creación: create, generate, push, just, from, fromArray, fromIterable, fromStream, never, range, empty
 - Filtrado: distinct, debounceTime, distinctUntilChanged, reduce, skip, take, takeUntil
- Los métodos block(), blockFirst() y blockLast() permiten transformar un proceso asíncrono e síncrono, bloqueando el proceso hasta que finalice y entregue todos los valores.

© JMA 2020. All rights reserved

WebClient

- Spring WebFlux incluye un cliente para realizar solicitudes HTTP. WebClient tiene una API fluida y funcional basada en Reactor, que permite la composición declarativa de lógica asíncrona sin la necesidad de lidiar con subprocessos o simultaneidad. Sin bloqueo, admite transmisión y se basa en los mismos códecs que también se utilizan para codificar y decodificar el contenido de solicitudes y respuestas en el lado del servidor.
- El método retrieve() se puede utilizar para declarar cómo extraer la respuesta.

```
WebClient client = WebClient.create("https://example.org");
Mono<ResponseEntity<Person>> result = client.get()
    .uri("/persons/{id}", id).accept(MediaType.APPLICATION_JSON)
    .retrieve()
    .toEntity(Person.class);
```

© JMA 2020. All rights reserved

WebClient

- Los métodos `exchangeToMono()` y `exchangeToFlux()` son útiles para casos más avanzados que requieren más control, como para decodificar la respuesta de manera diferente según el estado de la respuesta:

```
Mono<Object> entityMono = client.get()
    .uri("/persons/1")
    .accept(MediaType.APPLICATION_JSON)
    .exchangeToMono(response -> {
        if (response.statusCode().equals(HttpStatus.OK)) {
            return response.bodyToMono(Person.class);
        } else if (response.statusCode().is4xxClientError()) {
            return response.bodyToMono(ErrorContainer.class); // Suppress error status code
        } else {
            return response.createException().flatMap(Mono::error); // Turn to error
        }
    });
}
```

© JMA 2020. All rights reserved

WebClient

- El cuerpo de la solicitud se puede codificar desde cualquier tipo asíncrono manejado por `ReactiveAdapterRegistry`:

```
Mono<Person> personMono = ... ;
Mono<Void> result = client.put()
    .uri("/persons/{id}", id)
    .contentType(MediaType.APPLICATION_JSON)
    .body(personMono, Person.class)
    .retrieve()
    .bodyToMono(Void.class);
```

© JMA 2020. All rights reserved

SEGURIDAD

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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
```

© JMA 2020. All rights reserved

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
```
- 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) debe anotarse con **@RefreshScope**.
- Para refrescar la configuración en caliente después de realizar un commit al repositorio hay que hacer un POST a:
 - <http://localhost:8001/actuator/refresh>

© JMA 2020. All rights reserved

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;
```

© JMA 2020. All rights reserved

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>

© JMA 2020. All rights reserved

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 qué 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+

© JMA 2020. All rights reserved

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);
    }
}
```

© JMA 2020. All rights reserved

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.

© JMA 2020. All rights reserved

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 @EnableGlobalMethodSecurity en la configuración que deseé.
- 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.

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

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, que significa 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.
- UserService es la interfaz utilizada para crear el objeto UserDetails.

© JMA 2020. All rights reserved

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).

© JMA 2020. All rights reserved

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 concuerden con las esperadas, añadiéndole al objeto Authentication las autorizaciones asociadas a esa identidad o generando 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.

© JMA 2020. All rights reserved

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 mostrará un mensaje de error.

© JMA 2020. All rights reserved

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_”.

© JMA 2020. All rights reserved

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());
```

© JMA 2020. All rights reserved

## Configuración de Autenticación

- Para realizar la configuración crear una clase, anotada con @Configuration y @EnableWebSecurity, que extienda a WebSecurityConfigurerAdapter.
  - La sobreescritura del método configure(AuthenticationManagerBuilder) permite fijar el UserDetailsService y el PasswordEncoder.
- ```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(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());
    }
}
```

© JMA 2020. All rights reserved

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);
    }
}

```

© JMA 2020. All rights reserved

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");
}

```

© JMA 2020. All rights reserved

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

© JMA 2020. All rights reserved

Configuración

- La sobreescritura del método configure(HttpSecurity) permite configurar el http.authorizeRequests():
 - .antMatchers("/static/**").permitAll() acceso a los recursos
 - .anyRequest().authenticated() se requiere estar autenticado para todas las peticiones.
 - .antMatchers("/**").permitAll() equivale a anyRequest()
 - .antMatchers("/privado/**", "/config/**").authenticated() equivale a @PreAuthorize("authenticated")
 - .antMatchers("/admin/**").hasRole("ADMIN") equivale a @PreAuthorize("hasRole('ROLE_ADMIN')")
- El método .and() permite concatenar varias definiciones.

© JMA 2020. All rights reserved

Seguridad: Configuración

```

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

```

© JMA 2020. All rights reserved

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.
- Se puede habilitar la seguridad basada en anotaciones utilizando la anotación `@EnableGlobalMethodSecurity` en cualquier instancia `@Configuration`.
- `@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){
```
- `@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')")
@PreAuthorize("hasPermission(#contact, 'admin')")
```
- `@PostAuthorize`: Anotación para especificar una expresión de control de acceso al método que se evaluará después de que se haya invocado un método.

© JMA 2020. All rights reserved

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 agregaría. 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

© JMA 2020. All rights reserved

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')

© JMA 2020. All rights reserved

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 autenticar al propietario del recurso y obtener autorización.

© JMA 2020. All rights reserved

OAuth 2

Flujo de protocolo abstracto



© JMA 2020. All rights reserved

JWT: JSON Web Tokens

<https://jwt.io>

- 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 de 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 esté correctamente autenticado y saber quién 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 datos 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.

© JMA 2020. All rights reserved

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 son:
 - 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.

© JMA 2020. All rights reserved

Servidor de Autenticación/Autorización

- Dependencias: Spring Web y Spring Security

- En pom.xml

```
<dependency>
    <groupId>com.auth0</groupId>
    <artifactId>java-jwt</artifactId>
    <version>3.15.0</version>
</dependency>
```

- Configurar:

```
server.port=8081
spring.application.name=autentication-service
autenticacion.clave.secreta=Una clave secreta al 99% segura
autenticacion.expiracion.ms=3600000
```

© JMA 2020. All rights reserved

API de autenticación y obtención del token

```
@RestController
public class UserResource {
    @Value("${autenticacion.clave.secreta}")
    private String SECRET;
    @Value("${autenticacion.expiracion.ms}")
    private static final int EXPIRES_IN_MILLISECOND = 3600000;

    @PostMapping("/login")
    public UserToken login(@RequestBody UserCredential usr){
        if(usr.getUser() != null && usr.getPassword() != null && !usr.getUser().equals(usr.getPassword()))
            return new UserToken(usr.getUser(), createToken(usr.getUser(), "USER", "ADMIN"));
        return new UserToken("(Anonimo)", null);
    }
    public String createToken(String user, String... roles) {
        return "Bearer " + JWT.create()
            .withIssuer("MicroserviciosJWT")
            .withIssuedAt(new Date()).withNotBefore(new Date())
            .withExpiresAt(new Date(System.currentTimeMillis() + EXPIRES_IN_MILLISECOND))
            .withClaim("user", user)
            .withArrayClaim("roles", roles)
            .sign(Algorithm.HMAC256(SECRET));
    }
}
```

© JMA 2020. All rights reserved

Filtro de decodificación del token

```
public class JWTAuthorizationFilter extends OncePerRequestFilter {
    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 IOException, ServletException {
        String authenticationHeader = request.getHeader("Authorization");
        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("user").toString(), null, authorities);
            SecurityContextHolder.getContext().setAuthentication(auth);
        }
        chain.doFilter(request, response);
    }
}
```

© JMA 2020. All rights reserved

Configurar con el filtro

```
@EnableWebSecurity
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Value("${autenticacion.clave.secreta}")
    private String SECRET;

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .addFilterAfter(new JWTAuthorizationFilter(SECRET), UsernamePasswordAuthenticationFilter.class)
            .authorizeRequests()
            .antMatchers(HttpMethod.GET, "/publico").permitAll()
            .anyRequest().authenticated();
    }
}
```

© JMA 2020. All rights reserved

MONITORIZACIÓN Y RESILIENCIA

© JMA 2020. All rights reserved

Patrón: Log Aggregation

- Motivación:
 - Ha aplicado el patrón de arquitectura de microservicio.
- Intención:
 - ¿Cómo comprender el comportamiento de la aplicación y solucionar problemas?
- Requisitos:
 - Cualquier solución debe tener una sobrecarga mínima de tiempo de ejecución
- Solución:
 - Instrumentar los servicios para recopilar estadísticas sobre operaciones individuales. Métricas agregadas en el servicio de métricas centralizadas, que proporciona informes y alertas. Hay dos modelos para agregar métricas:
 - push: el servicio envía métricas al servicio de métricas
 - pull: los servicios de métricas extraen métricas del servicio.
- Implementación:
 - Crear servicios con Spring Boot y Spring Boot Actuator o Prometheus
- Consecuencias:
 - Este patrón tiene los siguientes beneficios: Proporciona una visión profunda del comportamiento de la aplicación.
 - Este patrón tiene los siguientes inconvenientes: El código de métricas está entrelazado con la lógica empresarial, lo que lo hace más complicado.
 - Este patrón tiene los siguientes problemas: La agregación de métricas puede requerir una infraestructura significativa

© JMA 2020. All rights reserved

Estado y diagnóstico

- Un microservicio debe notificar su estado y diagnóstico.
- En caso contrario, hay poca información desde una perspectiva operativa.
- Correlacionar eventos de diagnóstico en un conjunto de servicios independientes y tratar los desajustes en el reloj de la máquina para dar sentido al orden de los eventos suponen un reto.
- De la misma manera que interactúa con un microservicio según protocolos y formatos de datos acordados, hay una necesidad de estandarizar cómo registrar los eventos de estado y diagnóstico que, en última instancia, terminan en un almacén de eventos para que se consulten y se vean.
- En un enfoque de microservicios, es fundamental que distintos equipos se pongan de acuerdo en un formato de registro único.
- Debe haber un enfoque coherente para ver los eventos de diagnóstico en la aplicación.

© JMA 2020. All rights reserved

Comprobaciones de estado

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

© JMA 2020. All rights reserved

Monitorización

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

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

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

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

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

© JMA 2020. All rights reserved

Spring Boot 2.x Actuator

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

© JMA 2020. All rights reserved

Información de salud

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

© JMA 2020. All rights reserved

Métricas

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

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

© JMA 2020. All rights reserved

Spring Boot Admin

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

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

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

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

- Para exponer un actuator específico 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']
```

© JMA 2020. All rights reserved

Micrometer, Prometheus, Grafana

- Para consultar la recopilación: <http://localhost:9090>
- Para acceder y configurar Grafana:
 - <http://localhost:9090> 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://localhost: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 mediante su id o JSON (Ej: JVM (*Micrometer*) id: 4701)
- Las alertas permiten identificar problemas en el sistema momentos después de que ocurran. Al identificar rápidamente los cambios no deseados en el sistema, se puede minimizar las interrupciones en los servicios.

© JMA 2020. All rights reserved

Patrón: Health Check API

- Motivación:
 - Se ha aplicado el patrón de arquitectura Microservicios. A veces, una instancia de servicio puede ser incapaz de manejar solicitudes y aunque esté ejecutándose. Por ejemplo, podría haberse quedado sin conexiones de base de datos. Cuando esto ocurre, el sistema de monitoreo debe generar una alerta. Además, el equilibrador de carga o el registro de servicios no debe enrutar las solicitudes a la instancia de servicio fallida.
- Intención:
 - ¿Cómo detectar que una instancia de servicio en ejecución no puede manejar las solicitudes?
- Requisitos:
 - Se debe generar una alerta cuando falla una instancia de servicio
 - Las solicitudes deben dirigirse a las instancias de servicio en funcionamiento.
- Solución:
 - Un servicio tiene un extremo API de comprobación de estado (por ejemplo, HTTP /health) que devuelve el estado del servicio. El manejador de extremo API realiza varias comprobaciones, como
 - el estado de las conexiones a los servicios de infraestructura utilizados por la instancia de servicio
 - el estado del host, por ejemplo, espacio en disco
 - lógica específica de la aplicación
 - Los clientes de comprobación de estado, los servicios de supervisión, el registro de servicios o el balanceador de carga, invocan periódicamente el extremo para comprobar el estado de la instancia del servicio

© JMA 2020. All rights reserved

Resiliencia

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

© JMA 2020. All rights reserved

Patrón: Circuit Breaker

- Motivación:
 - Se ha 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 caída en cascada a otros servicios?
- Solución:
 - Las peticiones a un servicio remoto se deben invocar a través de un proxy que funciona de manera similar a un interruptor de circuito eléctrico (disyuntor). Cuando el número de fallos consecutivos cruza un umbral, el interruptor se dispara y, durante un período de tiempo de espera, todos los intentos de invocar el servicio remoto fallarán de inmediato. Una vez que el tiempo de espera expira, el interruptor permite que pase un número limitado de solicitudes de prueba. Si esas solicitudes son correctas, el interruptor reanuda el funcionamiento normal. De lo contrario, si persiste el fallo, el período de tiempo de espera comienza nuevamente.

© JMA 2020. All rights reserved

Patrón: Circuit Breaker

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

© JMA 2020. All rights reserved

Spring Cloud Circuit Breaker con Resilience4j

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

© JMA 2020. All rights reserved

Resilience4j

- Resilience4j es una biblioteca de tolerancia a fallas 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, reintentos 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.

© JMA 2020. All rights reserved

Resilience4j

- Instalación: Spring Cloud Circuit Breaker > Resilience4j
- Para proporcionar una configuración predeterminada para todos los disyuntores:


```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> defaultCustomizer() {
    return factory -> factory.configureDefault(id -> new Resilience4JConfigBuilder(id)
        .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(4)).build())
        .circuitBreakerConfig(CircuitBreakerConfig.ofDefaults())
        .build());
}
```
- De manera similar, se puede proporcionar una configuración personalizada:


```
@Bean
public Customizer<Resilience4JCircuitBreakerFactory> slowCustomizer() {
    return factory -> factory.configure(builder -> builder.circuitBreakerConfig(CircuitBreakerConfig.ofDefaults()
        .timeLimiterConfig(TimeLimiterConfig.custom().timeoutDuration(Duration.ofSeconds(2)).build()),
        "slow"));
}
```

© JMA 2020. All rights reserved

Resilience4j

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

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

© JMA 2020. All rights reserved

Resilience4j

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

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

© JMA 2020. All rights reserved

Resilience4j

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

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

© JMA 2020. All rights reserved

Patrón: Distributed Tracing

- Motivación:
 - Ha aplicado el patrón de arquitectura de microservicio. Las solicitudes suelen abarcar varios servicios. Cada servicio maneja una solicitud realizando una o más operaciones, por ejemplo, consultas a la base de datos, publicación de mensajes, etc..
- Intención:
 - ¿Cómo comprender el comportamiento de una aplicación y solucionar problemas?
- Requisitos:
 - La monitorización externa solo le dice el tiempo de respuesta general y el número de invocaciones, sin información sobre las operaciones individuales
 - Cualquier solución debe tener una sobrecarga mínima de tiempo de ejecución
 - Las entradas de registro para una solicitud se encuentran dispersas en numerosos registros
- Solución:
 - Instrumentalizar el código del servicios para que:
 - Asigne a cada solicitud externa una identificación de solicitud externa única
 - Pase el ID de solicitud externa a todos los servicios que participan en el manejo de la solicitud.
 - Incluya la identificación de solicitud externa en todos los mensajes de registro
 - Registre información (por ejemplo, hora de inicio, hora de finalización) sobre las solicitudes y operaciones realizadas al manejar una solicitud externa en un servicio centralizado
 - Esta instrumentación puede ser parte de la funcionalidad proporcionada por un marco de chasis de microservicio.

© JMA 2020. All rights reserved

Patrón: Distributed Tracing

- Implementación:
 - Crear servicios con Spring Boot y Spring Cloud Sleuth con un servidor Zipkin
- Consecuencias:
 - Este patrón tiene los siguientes beneficios:
 - Proporciona información útil sobre el comportamiento del sistema, incluidas las fuentes de latencia.
 - Permite a los desarrolladores ver cómo se maneja una solicitud individual al buscar en registros agregados su ID de solicitud externa
 - Este patrón tiene los siguientes problemas:
 - La agregación y el almacenamiento de seguimientos pueden requerir una infraestructura significativa.
- Patrones relacionados:
 - Log Aggregation: La identificación de la solicitud externa se incluye en cada mensaje de registro.

© JMA 2020. All rights reserved

Spring Cloud Sleuth 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.
- Spring Cloud Sleuth 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 Sleuth en la ejecución de los microservicios e información de los tiempos de respuesta de las invocaciones que han intervenido en una transacción. Ofrece las dos funcionalidades la recolección de datos y la obtención de los mismos. Tanto la recolección como el almacenamiento ofrecen diferentes herramientas para implementarlo: la recolección puede ser mediante peticiones HTTP, RabbitMQ o Kafka y el almacenamiento en memoria, MySQL, Cassandra o Elasticsearch.

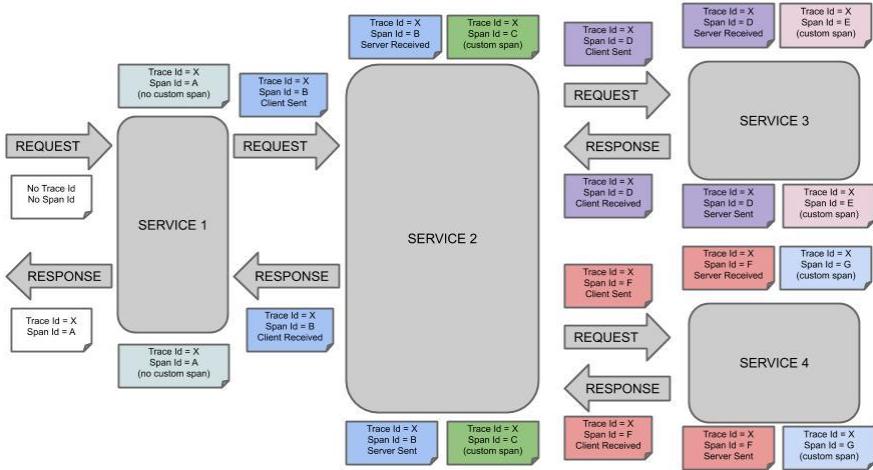
© JMA 2020. All rights reserved

Spring Cloud Sleuth 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 realizar 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). Restarla 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. Restarla de la marca temporal del cs indica todo el tiempo que necesita el cliente para recibir la respuesta del servidor.

© JMA 2020. All rights reserved

Spring Cloud Sleuth y Zipkin



© JMA 2020. All rights reserved

Spring Cloud Sleuth 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 > Sleuth, Observability > Zipkin Client
- Configurar los clientes Zipkin en application.properties:
 - spring.zipkin.baseUrl=http://localhost:9411/
 - spring.zipkin.enabled=true
 - spring.zipkin.sender.type=web
- Para consultar las trazas:
 - <http://localhost:9411/>

© JMA 2020. All rights reserved

Spring Cloud Sleuth y Zipkin

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

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

© JMA 2020. All rights reserved

Patrón: Log Aggregation

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

© JMA 2020. All rights reserved

Registros

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

© JMA 2020. All rights reserved

Registro

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



© JMA 2020. All rights reserved

Niveles de registro

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

© JMA 2020. All rights reserved

Controladores y Formateadores

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

© JMA 2020. All rights reserved

Configuración

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

© JMA 2020. All rights reserved

Realizar notificaciones

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

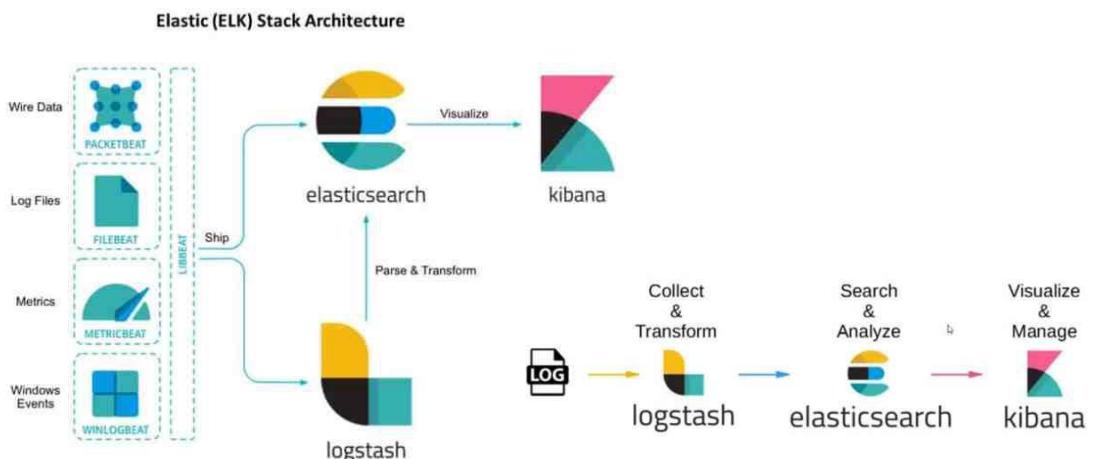
© JMA 2020. All rights reserved

ELK: Elasticsearch , Logstash y Kibana

- “ELK” son las siglas para tres proyectos open source: Elasticsearch, Logstash y Kibana. Elasticsearch es un motor de búsqueda textual y analítica. Logstash es un pipeline (ETL) de procesamiento de datos del lado del servidor que obtiene datos de una multitud de fuentes simultáneamente, los transforma y luego los envía a un almacenamiento como Elasticsearch. Kibana permite a los usuarios visualizar los datos de Elasticsearch en cuadros y gráficas.
- Elasticsearch es un motor de búsqueda y analítica RESTful distribuido capaz de abordar multitud de casos de uso. Como núcleo del Elastic Stack, almacena e indexa de forma centralizada los datos para una búsqueda (estructuradas, no estructuradas, geográficas, métricas, ...) a gran velocidad, con relevancia refinada y analíticas que escalan con facilidad.
- Kibana es una interfaz de usuario gratuita y abierta que permite visualizar los datos de Elasticsearch y navegar en el Elastic Stack, desde rastrear la carga de búsqueda hasta comprender la forma en que las solicitudes fluyen por las apps. Como una imagen vale más que mil líneas de log, Kibana envía los datos en forma de histogramas, grafos de líneas, gráficos circulares, proyecciones solares y más, para análisis de logs, monitoreo de infraestructura, APM (Application Performance Monitoring), operaciones de seguridad, analítica de negocios, ...
- Logstash es un ETL 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 poderoso. Los resultados se pueden enviar a Elasticsearch para búsquedas y análisis o a una variedad de salidas.

© JMA 2020. All rights reserved

ELK: Elasticsearch , Logstash y Kibana



© JMA 2020. All rights reserved

DESPLIEGUE

© JMA 2020. All rights reserved

Modelo de despliegue

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

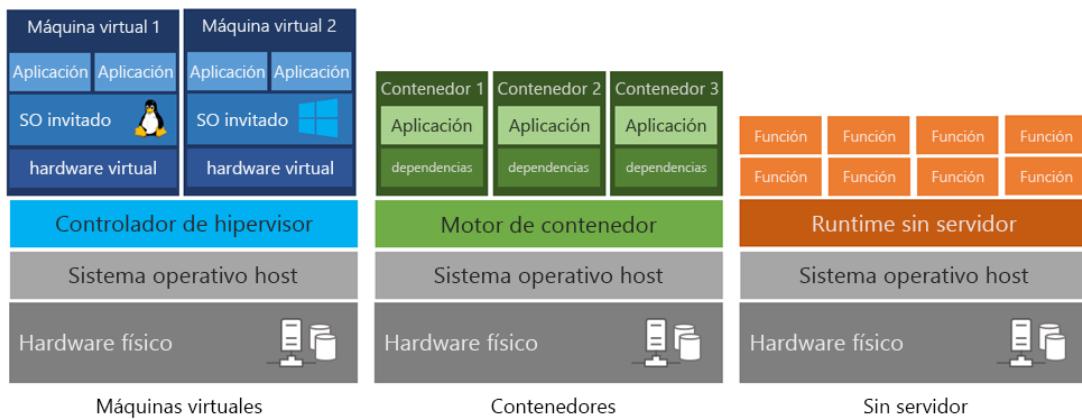
© JMA 2020. All rights reserved

Modelo de despliegue

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

© JMA 2020. All rights reserved

Contenedores



© JMA 2020. All rights reserved

Modelo de despliegue

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

© JMA 2020. All rights reserved

FaaS (Functions-as-a-Service)

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

© JMA 2020. All rights reserved

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