Microservicios con Spring Cloud

© JMA 2016. All rights reserved

Contenidos

- Arquitectura de microservicios
- Spring con Spring Boot
- Estilos de comunicación
  - Servicios REST
  - Spring HATEOAS
  - Swagger
  - Spring Data Rest
  - RabbitMQ
- Registro/Descubrimiento
  - Eureka
  - Ribbon
  - RestTemplate
  - Feign
  - Zuul
- Gestión de datos
  - Spring Data
  - MongoDB
  - Redis
  - Patrón: Saga
  - Patrón: API de Composición
  - Patrón: CQRS
  - Patrón: Event sourcing
  - Patrón: Eventos de aplicación
- Monitorización y Resiliencia
  - Spring Boot 2.x Actuator
  - Spring Boot Admin
  - Hystrix
  - Hystrix Dashboard
  - Turbine
- Seguridad
  - Spring Cloud Config
  - CORS
  - Spring Security
  - JWT: JSON Web Tokens

© JMA 2016. All rights reserved

Enlaces

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

© JMA 2016. All rights reserved

ARQUITECTURA DE MICROSERVICIOS

© JMA 2016. All rights reserved

Introducción

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

© JMA 2016. All rights reserved

Antecedentes

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

© JMA 2016. All rights reserved

SOA

- El concepto de dividir una aplicación en partes discretas no es nuevo. La idea para microservicios se origina en el patrón SOA de diseño de arquitectura orientado a servicios más amplio, en el que los servicios independientes realizan funciones distintas y se comunican utilizando un protocolo designado.
- Sin embargo, a diferencia de la arquitectura orientada a servicios, una arquitectura de microservicios (como su nombre indica) debe contener servicios que son explícitamente pequeños y ligeros y que son desplegables de forma independiente. Los objetivos son:
  - Poder utilizar diferentes tecnologías por cada servicio (Java EE, Node, …)
  - Permitir que cada servicio tenga un ciclo de vida independiente, es decir, versión independiente del resto, inclusive equipos de desarrollo diferentes.
  - Al ser servicios sin dependencia entre sí (especialmente de sesión), poder ejecutar el mismo en varios puertos, colocando un balanceador delante.
  - Poder crear instancias en servidores de diferentes regiones, lo que permitirá crecer (tanto verticalmente como horizontal) sin necesidad de cambiar el código fuente.

© JMA 2016. All rights reserved

Componentización a través de Servicios

- 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 de función en memoria, mientras que los servicios 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 Microservicio usarán librerías, pero su manera primaria de componentización es dividirlo 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 2016. All rights reserved

Organización de equipos alrededor de Capacidades Empresariales

- Cuando se busca dividir una aplicación 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 lógicos del servidor y equipos de bases de datos.
- Cuando los equipos están separados de esta amanera, incluso los cambios simples pueden conducir a proyectos cruzados entre equipos que suponen tiempo y coste.
- El enfoque de microservicio 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 2016. 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.
  - Los proponentes de microservicios tienden 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.
  - Los microservicios permiten usar la herramienta adecuada para cada caso.

© JMA 2016. All rights reserved

Endpoints inteligentes y tubos mudos

- 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 comunidad entre microservicio favorece un enfoque alternativo: **puntos finales inteligentes y conexiones tontas**.
- Las aplicaciones construidas a partir de microservicios 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 2016. All rights reserved

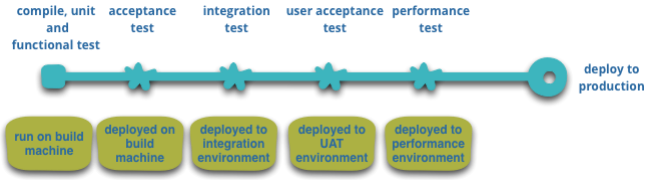
Gestión descentralizada de datos

- 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".
- Este proceso es útil tanto para arquitecturas monolíticas como para microservicios.
- Los microservicios 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 2016. 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 en particular ha reducido la complejidad operativa de la creación, implementación y operación de microservicios.
- Muchos de los productos o sistemas que se están construyendo con microservicios 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 2016. All rights reserved

Diseño tolerante a fallos

- Una consecuencia del uso de servicios 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.
- Las aplicaciones de microservicio ponen 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 que provoca que los equipos de desarrollo investiguen.

© JMA 2016. All rights reserved

Diseño Evolutivo

- La implementación de componentes en los servicios 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 los microservicios, sin embargo, sólo es necesario volver a implementar el servicio que 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.
- Con microservicios, 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 tiene 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 el mundo de los microservicios 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 2016. All rights reserved

Monolítico: Beneficios

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

© JMA 2016. All rights reserved

Monolítico: Inconvenientes

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

© JMA 2016. All rights reserved

Monolítico: Inconvenientes

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

© JMA 2016. All rights reserved

Monolítico: Inconvenientes

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

© JMA 2016. All rights reserved

Microservicios: Beneficios

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

© JMA 2016. All rights reserved

Microservicios: Inconvenientes

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

© JMA 2016. All rights reserved

Cuándo usar la arquitectura de microservicios

- Un desafío con el uso de este enfoque es decidir cuándo tiene sentido usarlo.
  - Productos maduros
  - Sistemas (web) muy grandes
  - Implementación a nivel corporativo
  - Cuanto mayor sea el producto/proyecto mayor ventaja
- Al desarrollar la primera versión de una aplicación, a menudo no se tienen los problemas que este enfoque resuelve. Además, el uso de una arquitectura elaborada y distribuida ralentizará el desarrollo.
- Esto puede ser un problema importante para las startups cuyo mayor desafío es a menudo cómo evolucionar rápidamente el modelo de negocio y la aplicación que lo acompaña.
- Más adelante, sin embargo, cuando el desafío es cómo escalar y se necesita utilizar descomposición funcional, las interdependencias podrían dificultar la descomposición de una aplicación monolítica en un conjunto de servicios. Unido a esto está la madurez de la empresa y de los equipos que la componen.

© JMA 2016. All rights reserved

Que debe cumplir un microservicio

- Cubrir una y solo una funcionalidad muy concreta (en torno a capacidades empresariales).
- Que se pueda ejecutar en un proceso y ser desplegado de forma independiente a los demás.
- Tener su propia base de datos
- Poder estar implementado en el lenguaje de programación y plataforma mas adecuada.
- Ser altamente mantenible, sustituible, descartable y comprobable
- Débilmente acoplado, comunicación a través de HTTP, mensajería, service bus, …
- Debería ser entendible por una sola persona

© JMA 2016. All rights reserved

Cómo descomponer la aplicación en servicios

- Se puede descomponer por capacidad empresarial y definir servicios que correspondan a las capacidades empresariales.
- Se puede descomponer por contextos delimitados (Bounded Context) del DDD.
- Se puede descomponer por verbo o caso de uso y definir servicios que son responsables de acciones particulares. p.ej. un servicio de envío que es responsable de enviar pedidos completos. (Orientación a funcionalidad)
- Se puede descomponer por sustantivos o recursos definiendo un servicio que es responsable de todas las operaciones en entidades / recursos de un tipo determinado. p.ej. un Servicio de Cuentas que es responsable de administrar cuentas de usuario. (Orientación al recurso)
- Idealmente, cada servicio debe tener sólo un pequeño conjunto de responsabilidades (S.O.L.I.D.).
- Modelo de dos capas:
  - Capa de acceso a recurso (DaaS)
  - Capa de lógica de negocio

© JMA 2016. All rights reserved

Cómo mantener la consistencia de los datos

- Para garantizar el acoplamiento débil, cada servicio debe tener su propia base de datos.
- Mantener la coherencia de los datos entre los servicios es un reto. Una aplicación debe usar el patrón Saga:
  - Un servicio publica un evento cuando sus datos cambian.
  - Otros servicios consumen ese evento y actualizan sus datos.
- Existen varias maneras de actualizar de manera fiable los datos y los eventos de publicación, incluyendo el Sourcing de eventos y el registro de "registros de transacciones".
- Otro desafío es implementar consultas que necesitan recuperar datos pertenecientes a múltiples servicios.
- Los siguientes patrones pueden ser útiles:
  - Api de Composición: http://microservices.io/patterns/data/api-composition.html)
  - Segregación de Responsabilidad de Consultas de Comando (CQRS): http://microservices.io/patterns/data/cqrs.html

© JMA 2016. All rights reserved

Buenas practicas

- Construir un almacenamiento de datos separado por cada microservicio.
- Construir y desplegar cada microservicio por separado.
- Centralizar la configuración
- Desplegar con contenedores
- Automatizar el despliegue
- Seguir la cultura Cloud: compra lo que necesites, cuando lo necesites y paga por lo que uses
- Aplicar metodologías agiles y TDD
- Establecer una cultura anti-burocracia: libertad con responsabilidad

© JMA 2016. All rights reserved

Preocupaciones transversales

- La tendencia natural en cuanto a microservicios es crecer, tanto por nueva funcionalidad 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 …

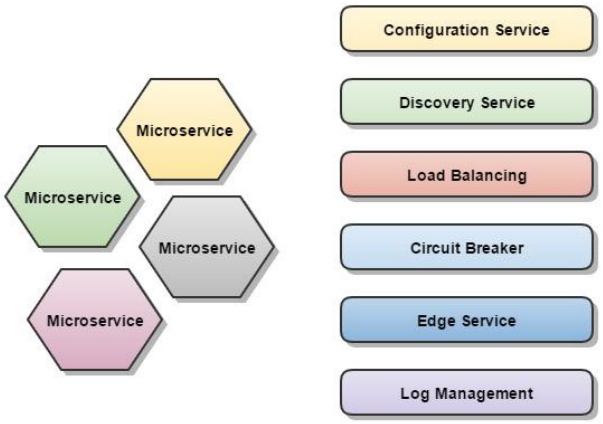
© JMA 2016. All rights reserved

Implantación

- Para la implantación de una arquitectura de microservicios hemos tener en cuenta 3 aspectos principalmente:
  - Un modelo de referencia en el que definir las necesidades de una arquitectura de microservicios.
  - 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 2016. All rights reserved

Modelo de referencia



© JMA 2016. 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.
- Mensajeria:
  - Las invocaciones siempre serán síncronas (REST, SOAP, …) o también llamadas asíncronas (AMQP).

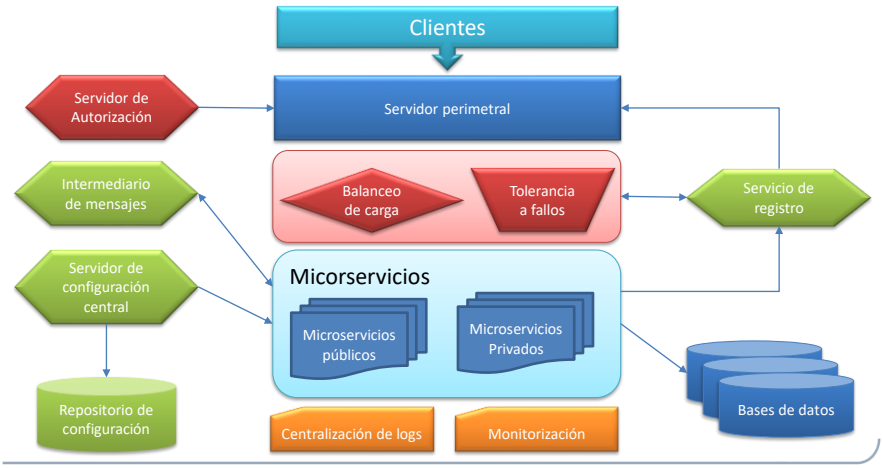
© JMA 2016. All rights reserved

Modelo de referencia

- Servidor de configuración central
  - Este componente se encargará de centralizar y proveer remotamente la configuración a cada microservicio. 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...

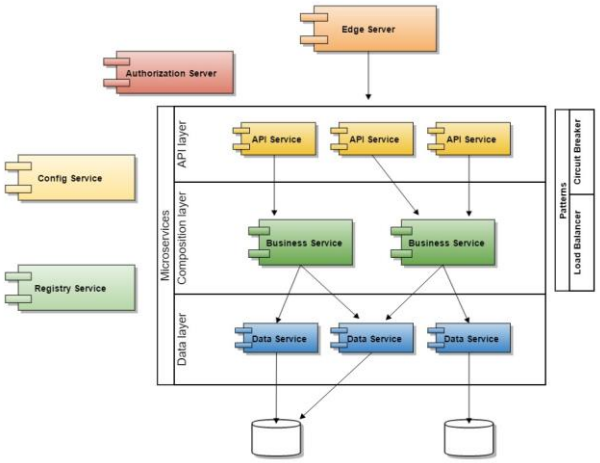
© JMA 2016. All rights reserved

Modelo de referencia



© JMA 2016. All rights reserved

Modelo de referencia



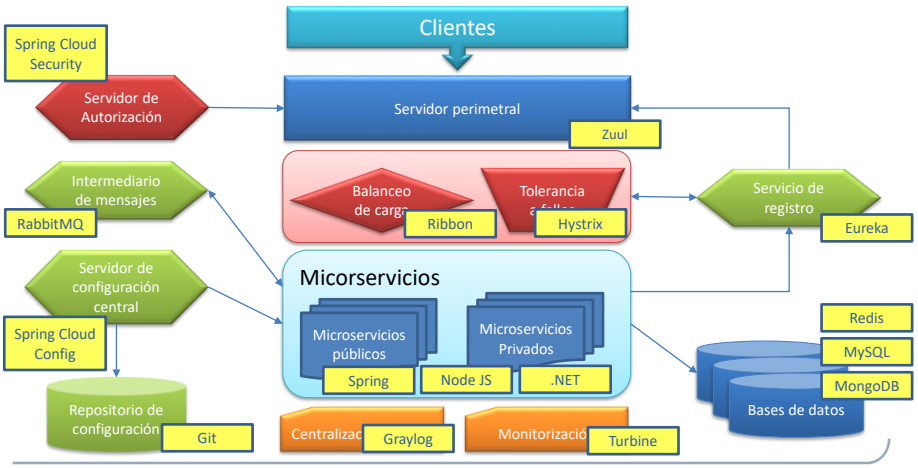
© JMA 2016. All rights reserved

Modelo de implementación

- 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 nuestro 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 2016. All rights reserved

Modelo de implementación



© JMA 2016. 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.

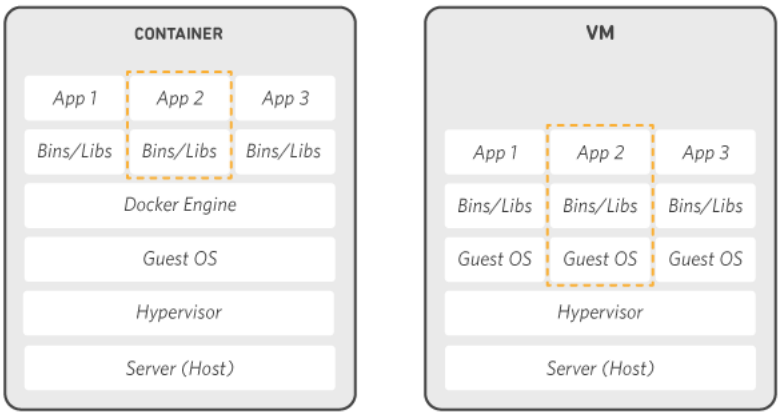
© JMA 2016. All rights reserved

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 2016. All rights reserved

Contenedores



© JMA 2016. 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 2016. All rights reserved

http://spring.io

SPRING CON SPRING BOOT

© JMA 2016. 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
  - MVC
  - Negocio (donde empezó originalmente)
  - Acceso a datos

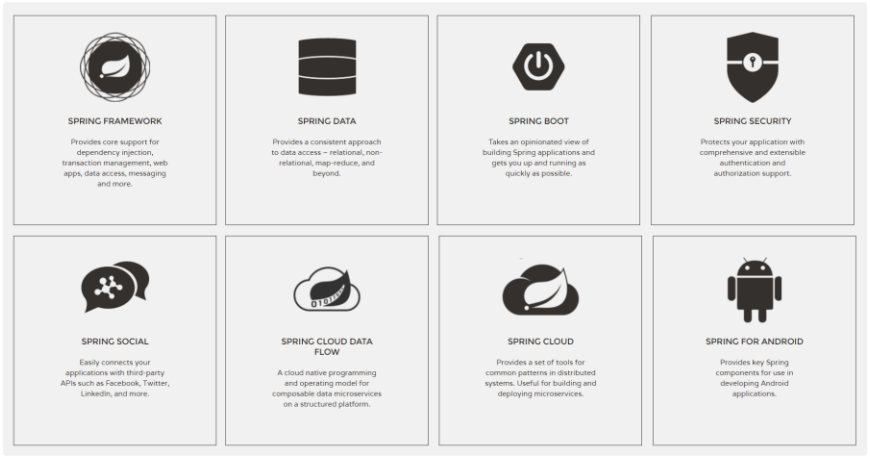
© JMA 2016. All rights reserved

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 2016. All rights reserved

Proyectos



© JMA 2016. 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 2016. 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 absolutamente 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 2016. All rights reserved

Dependencias: starters

- Los starters son un conjunto de descriptores de dependencias convenientes (versiones compatibles, ya probadas) que se pueden incluir en su 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 2016. All rights reserved

Spring Tool

- https://spring.io/tools
- Spring Tool Suite
  - IDE gratuito, personalización del Eclipse
- Plug-in para Eclipse (VSCode, Atom)
  - Help → Eclipse Marketplace …
    - Spring Tools 4 for Spring Boot

© JMA 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Configuración

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

```
# Oracle settings
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=hr
spring.datasource.password=hr
spring.datasource.driver-class=oracle.jdbc.driver.OracleDriver

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

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

server.port=${PORT:8080}
```

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

© JMA 2016. 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 -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 --name mysql-sakila 1maa/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
- 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 2016. All rights reserved

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

IOC CON SPRING CORE

© JMA 2016. All rights reserved

Inversión de Control

- Inversión de control (Inversion of Control en inglés, IoC) es un concepto junto a 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 2016. 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 2016. All rights reserved

Inyección

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

© JMA 2016. All rights reserved

Introducción

- Spring proporciona un contenedor encargado de la inyección de dependencias (Spring Core Container).
- Este contenedor nos posibilita inyectar 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 singletons sino se especifica lo contrario.

© JMA 2016. 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 2016. All rights reserved

Beans

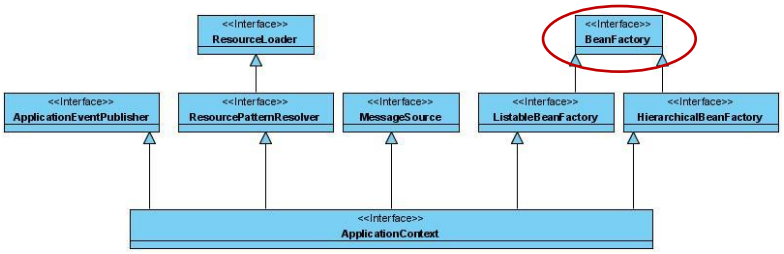
- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories, 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 2016. All rights reserved

Bean factory

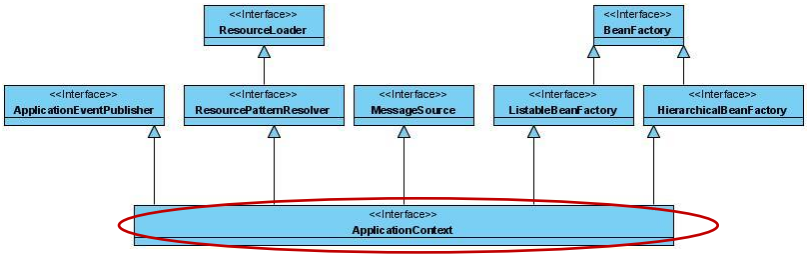
- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory)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



© JMA 2016. 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 2016. 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");
    BeanFactory factory = context;
    Client client = (Client )factory.getBean("ID_Cliente");
    client.go();
}
```

- Muestra:
  - Este es un servicio...

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Inyección

- 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")
public class MyNewInterfaceImpl implements MyInterface { ... }

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

© JMA 2016. 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 2016. 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 2016. 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 ira 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 2016. All rights reserved

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 2016. 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 2016. 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.

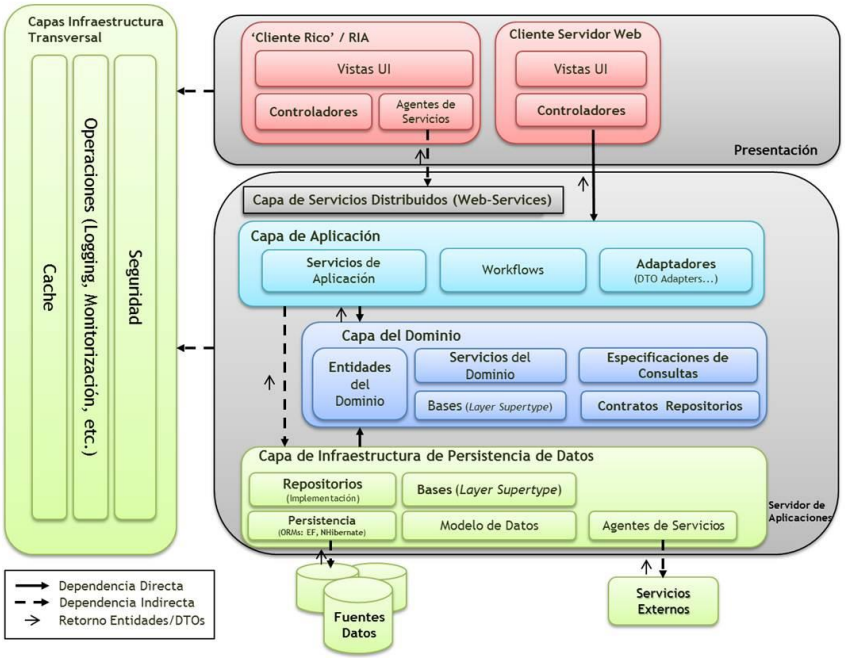
Anotaciones Spring	Anotaciones Estándar (javax.inject.*) JSR-330
@Autowired	@Inject
@Component	@Named / @ManagedBean
@Scope("singleton")	@Singleton
@Qualifier	@Qualifier / @Named
@Value	-
@Required	-
@Lazy	-

© JMA 2016. All rights reserved

GESTIÓN DE DATOS

© JMA 2016. 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 (Infraestructure)
  - 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 2016. 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 publica (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 2016. 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 un 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 2016. 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 2016. 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 CQRS 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 2016. 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 2016. All rights reserved

Anotaciones JPA

Anotación	Descripción
@Entity	- Se aplica a la clase. - Indica que esta clase Java es una entidad a persistir.
@Table(name="Tabla")	- 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	- Se aplica a una propiedad Java e indica que este atributo es la clave primaria.
@Column(name="Id")	- 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(…)	- name: nombre - length: longitud - precision: número total de dígitos - scale: número de digitos decimales - unique: restriccion valor unico - nullable: restriccion valor obligatorio - insertable: es insertable - updatable: es modificable
@Transient	- Se aplica a una propiedad Java e indica que este atributo no es persistente

© JMA 2016. 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 2016. 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 2016. 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 2016. 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
- Acepta múltiples valores:
  - @OneToMany(mappedBy="profesor", cascade={CascadeType.PERSIST, CascadeType.MERGE})

© JMA 2016. 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 2016. 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 2016. All rights reserved

Repositorio

- Con el soporte de Spring Data, la tarea repetitiva de crear las implementaciones concretas de DAO para las entidades se simplifica porque solo vamos se necesita 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 2016. All rights reserved

Repositorio

- 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 especifica para crear dichos nombre:

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

int deleteByEdadGreaterThan(int valor);
```

© JMA 2016. All rights reserved

Repositorio

- Prefijo consulta derivada:
  - find (read, query, get), count, delete
- Opcionalmente, limitar los resultados de la consulta:
  - Distinct, Top*NumFilas* y First*NumFilas*
- Expresión de propiedad: By*Propiedad*
  - Operador (Between, LessThan, GreaterThan, Like, …) por defecto equal.
  - Se pueden concatenar varias con And y Or
  - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, OrderBy*Propiedad*Asc 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 2016. All rights reserved

Repositorio

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
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1

© JMA 2016. 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 2016. 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.firstame) = UPPER(?1)

© JMA 2016. All rights reserved

Repositorio

- 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 2016. All rights reserved

Repositorio

- 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 2016. 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 2016. 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 conveniente para los clientes.
  - Evitar el "exceso" y las vulnerabilidades por publicación.

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 de 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 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Serialización XML (JAXB)

- JAXB (Java XML API Binding) proporciona a 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 2016. 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.
  - @XmlID: Mapea un propiedad JavaBean como un XML ID.
  - @XmlType(propOrder = { "field2", "field1",.. }): Permite definir en que 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 2016. 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 2016. 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 2016. 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 @NotEmptyes 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 2016. 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 númerico 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 2016. 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 2016. 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 forma parte 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 2016. 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 2016. 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=, baseURI=, 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 2016. 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 2016. All rights reserved

Validaciones personalizadas

- Clase del validador

```
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 2016. 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 hu***mongo***us, "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 jerarquica).
- 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 2016. 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 2016. All rights reserved

MongoDB: Instalación

- Se puede utilizar un instalador o instalarla manualmente.
- Descargar la última versión estable del Communitu 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 2016. 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 2016. All rights reserved

MongoDB: Comandos

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

© JMA 2016. 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 2016. 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 2016. All rights reserved

Repositorio (MongoDB)

P.Clave	Muestra	Resultado lógico
After	findByBirthdateAfter(Date date)	{"birthdate" : {"$gt" : date}}
GreaterThan	findByAgeGreaterThan(int age)	{"age" : {"$gt" : age}}
GreaterThanEqual	findByAgeGreaterThanEqual(int age)	{"age" : {"$gte" : age}}
Before	findByBirthdateBefore(Date date)	{"birthdate" : {"$lt" : date}}
LessThan	findByAgeLessThan(int age)	{"age" : {"$lt" : age}}
LessThanEqual	findByAgeLessThanEqual(int age)	{"age" : {"$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 2016. 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 2016. 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]}}}
IsTrue, True	findByActiveIsTrue()	{"active" : true}
IsFalse, False	findByActiveIsFalse()	{"active" : false}
Exists	findByLocationExists(boolean exists)	{"location" : {"$exists" : exists }}

© JMA 2016. 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 2016. 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 pensaríamos 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Redis: Comandos

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

© JMA 2016. 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 2016. 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 2016. 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, findByFirstnameIs, findByFirstnameEquals	SINTER …:firstname:rand
IsTrue	FindByAliveIsTrue	SINTER …:alive:1
IsFalse	findByAliveIsFalse	SINTER …:alive:0
Top,First	findFirst10ByFirstname, findTop5ByFirstname	

© JMA 2016. 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 2016. All rights reserved

Estilos de comunicación

SERVICIOS REST

© JMA 2016. All rights reserved

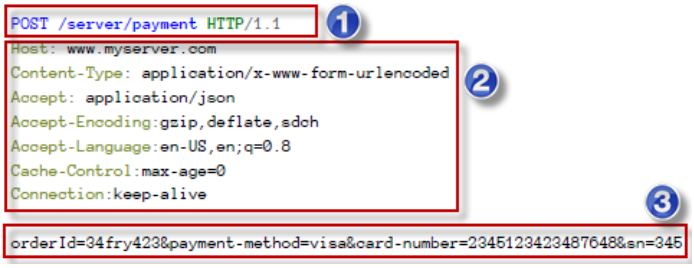
REST (REpresentational State Transfer)

- Un **estilo de arquitectura** para desarrollar aplicaciones web distribuidas que se basa en el uso del protocolo HTTP e Hypermedia.
- 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.
- El 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
- El HTTP permite en el encabezado transmitir la información de comportamiento:
  - Accept, Content-type, Response (códigos de estado), Authorization, Cache-control, …

© JMA 2016. All rights reserved

Petición HTTP

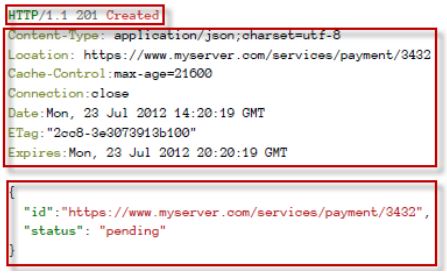
- Cuando realizamos una petición HTTP, el mensaje consta de:
  - Primera línea de texto indicando la versión del protocolo utilizado, el verbo y el URI
    - El verbo indica la acción a realizar sobre el recurso web localizado en la URI
  - Posteriormente vendrían las cabeceras (opcionales)
  - Después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



© JMA 2016. 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.



© JMA 2016. All rights reserved

Recursos

- Un recurso es cualquier elemento que dispone de un URI correcto y único.
- Es cualquier cosa que sea direccionable a través de internet.
- Estos recursos pueden ser manipulados por clientes y servidores.
  - Una noticia.
  - La temperatura en Madrid a las 22:00h.
  - Un estudiante de alguna clase en alguna escuela
  - Un ejemplar de un periódico, etc
- En REST todos los recursos comparten una interfaz única y constante. (http://...)
- Todos los recursos tienen las mismas operaciones (CRUD)
  - CREATE, READ, UPDATE, DELETE
- Las URI es el único medio por el que los clientes y servidores pueden realizar el intercambio de representaciones.
- Normalmente estos recursos son accesibles en una red o sistema.
- Para que un URI sea correcto, debe de 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 2016. 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 el 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Códigos de error del navegador

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

© JMA 2016. All rights reserved

Códigos de error del navegador

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 2016. 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 2016. 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
- Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- HTTP Status Code: Código de estado con el que el servidor informa del resultado de la petición.

© JMA 2016. All rights reserved

Peticiones

- Request: GET /users
- Response: 200
  - content-type:application/json
  - BODY
- Request: GET /users/11
- Response: 200
  - content-type:application/json
  - BODY
- Request: POST /users
  - BODY
- Response: 201
  - content-type:application/json
  - BODY
- Request: PUT /users
  - BODY
- Response: 200
  - content-type:application/json
  - BODY
- Request: DELETE /users/11
- Response: 204 no content

© JMA 2016. All rights reserved

Richardson Maturity Model

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

- # Nivel 1 (Pobre): Se usan URIs para identificar recursos:
  - Se debe identificar un recurso
    /invoices/?page=2 → /invoices/page/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): Se usa el protocolo HTTP adecuadamente
- # Nivel 3 (Óptimo): Se implementa hypermedia.

© JMA 2016. All rights reserved

Hypermedia

- 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 2016. 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 2016. All rights reserved

Diseño de un Servicio Web REST

- Para el desarrollo de los Servicios Web's REST es necesario conocer 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
  - Conocer los métodos soportados por el servicio
    - GET, POST, PUT, DELETE
  - 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 2016. 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, Movil, …) y de tecnologías (Java, Ruby, Python, Perl, Flex, SmallTalk, .Net …)

© JMA 2016. 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 2016. 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 2016. All rights reserved

Patrones de URI

- Establece que URLs serán derivadas al controlador.
- Puede asignar solicitudes utilizando los siguientes patrones globales y comodines:
  - ? cualquier carácter
  - * cero o más caracteres dentro de un segmento de ruta
  - ** cero o más segmentos de ruta
- 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 2016. All rights reserved

Restricciones

- cosumes: formatos MIME permitidos del encabezado Content-type

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

- produces : formatos MIME permitidos del encabezado Accept

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

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

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

© JMA 2016. 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 2016. All rights reserved

Inyección de Parámetros

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

© JMA 2016. All rights reserved

Inyección de Parámetros

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

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 property,property(,ASC|DESC). Por defecto: ascendente. Hay que utilizar varios sort para diferente direcciones (?sort=firstname&sort=lastname,asc)

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

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Servicio Web RESTful

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

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

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

© JMA 2016. 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 2016. 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 2016. 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 2016. 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, tetc.
- Spring Hateoas ahora proporciona una ControllerLinkBuilder que permite crear enlaces apuntando a clases de controladores:

```
import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;
```

- Para añadir una referencia a si mismo:

```
personaDTO.add(linkTo(PersonaResource.class).withSelfRel());
personaDTO.add(linkTo(PersonaResource.class).slash(personaDTO.getId()).withSelfRel());
```

- Para añadir una referencia a si mismo como método:

```
personaDTO.add(linkTo(PersonaResource.class.getMethod("get", Long.class),
        personaDTO.getId()).withSelfRel());
```

- Para crear una referencia a un elemento interno:

```
personaDTO.add(linkTo(PersonaResource.class).
        slash(personaDTO.getId()).slash("direciones").withRel("direciones"));
```

© JMA 2016. 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 2016. 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 2016. All rights reserved

Spring Data Rest

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

© JMA 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Documentar el servicio

- Web Application Description Language (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
  - Documentación a través de los test (casos de uso), evita enterrar el código entre anotaciones.
- Swagger
  - Conceptualmente similar al JavaDoc pero con anotación. Es el más ampliamente difundido.
  - Cuenta con un ecosistema propio.

© JMA 2016. 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 2016. All rights reserved

Swagger: Instalación

- 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 2016. 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 2016. 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 2016. All rights reserved

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
  - server.servlet.contextPath=/
- 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 2016. All rights reserved

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-bean-validators</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 2016. All rights reserved

Swagger: Anotar el modelo

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

```
@ApiModel(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 2016. All rights reserved

Swagger: Anotar el servicio

- **@Api**: documenta el servicio REST en si. 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.

```
@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)
      throws NotFoundException {
```

- **@ApiParam**: documenta los parámetros de cada método del servicio.
- **@ApiResponses**, @ApiResponse: documenta las posibles respuestas del método, con un mensaje explicativo.

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

© JMA 2016. All rights reserved

Swagger: 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 2016. All rights reserved

Archivo de propiedades

- 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 2016. All rights reserved

Estilos de comunicación

MENSAJERÍA

© JMA 2016. 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 2016. 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 2016. 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 subscribirse a los eventos y hacer su trabajo cuando sea necesario.
- La desventaja de la coreografía es que la vista explícita del proceso de negocio del modelado de proceso ahora sólo se refleja de manera implícita en el sistema.
- Esto implica que se necesita de trabajo adicional para asegurarse de que alguien pueda controlar y realizar un seguimiento de que hayan sucedido las cosas correctas.
- Para solucionar este problema, normalmente es necesario crear un sistema de monitoreo que refleje explícitamente la vista del proceso de negocio del modelado de procesos, pero que a su vez, haga un seguimiento de lo que cada uno de los servicios realiza como entidad independiente que le permite ver excepciones mapeadas al flujo de proceso más explícito.

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 recive(MessageDTO inMsg) {
        // Procesar el mensaje recibido: inMsg
}
```

© JMA 2016. 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 2016. All rights reserved

CONSULTAS ENTRE SERVICIOS

© JMA 2016. All rights reserved

Eureka

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

© JMA 2016. 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 2016. 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}/e
        ureka/
server.port: ${PORT:8001}
```

- Arrancar microservicio
- Refrecar dashboard de Eureka:
  - http://localhost:8761/

© JMA 2016. All rights reserved

RestTemplate

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

```
<dependency>
   <groupId>org.springframework</groupId>
   <artifactId>spring-web</artifactId>
</dependency>
<dependency>
   <groupId>com.fasterxml.jackson.core</groupId>
   <artifactId>jackson-databind</artifactId>
</dependency>
```

- Para poder inyectar la dependencia:

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

© JMA 2016. 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 2016. 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 2016. All rights reserved

RestTemplate

- Para recuperar uno:

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

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

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

© JMA 2016. All rights reserved

RestTemplate

- Para crear o modificar un recurso:

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

- Para crear o modificar un recurso con identificador:

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

- Para borrar un recurso con identificador:

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

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Ribbon

- Ribbon es un equilibrador de carga del lado del cliente que le da mucho control sobre el comportamiento de los clientes HTTP y TCP, que se integra perfectamente con Eureka, RestTemplate y Feign.
- Las características de Ribbon:
  - Balanceo de carga, permite utilizar y configurar diferentes algoritmos
  - Tolerancia a fallos: determina dinámicamente qué servicios están corriendo y activos, al igual que cuales están caídos
  - Soporte de protocolo múltiple (HTTP, TCP, UDP) en un modelo asincrónico y reactivo
  - Almacenamiento en caché y procesamiento por lotes
  - Integración con los servicios de autodescubrimiento, como por ejemplo Eureka o Consul
- Las arquitecturas basadas en microservicios suelen tener implementación multiregional y multizona ya que así mejora la disponibilidad y la resiliencia de los servicios.

© JMA 2016. All rights reserved

Ribbon

- Ribbon utiliza básicamente dos elementos a la hora de decidir cuál será la instancia a la que finalmente derivará la petición, estos dos elementos son los balanceadores (con sus filtros asociados) y las reglas.
- Así, en primer lugar Ribbon utilizará un balanceador y filtros para descartar una serie de instancias del microservicio a invocar, en base a diversos criterios: que las instancias estén caídas, que estén en una zona con una alta carga de peticiones…
- Una vez pasada esta primera etapa quedarán una serie de instancias que son las que cumplen las condiciones implementadas en los filtros, y ahí entrarán en juego las reglas de balanceo para determinar a cual de esas instancias enviar la petición.
- Ribbon dispone de diferentes filtros y reglas así como la posibilidad de implementar los que deseemos.

© JMA 2016. All rights reserved

Ribbon

- A la hora de seleccionar y configurar los algoritmos existen las siguientes posibilidades:
  - RoundRobinRule: algoritmo de uso muy extendido, donde los servidores van rotando uno tras otro
  - AvailabilityFilteringRule: el balanceador de carga puede evitar una zona al elegir el servidor
  - WeightedResponseTimeRule: esta regla usa los tiempos medios de respuesta para asignar pesos a los servidores, es como usar un «Weighted Round Robin»
  - DynamicServerListLoadBalancer: tiene la capacidad de obtener la lista de servidores candidatos utilizando una fuente dinámica
  - Podemos definir nuestra propia regla custom
- Además del algoritmo de balanceo de carga, para la resiliencia, ofrece opciones de configuración, tales como:
  - Tiempos de espera de conexión
  - Reintentos
  - Algoritmos de reintento (exponencial, límite acotado)

© JMA 2016. All rights reserved

Ribbon

- Dependencia: Spring Cloud Routing > Ribbon
- Una vez incluidas las dependencias en nuestro proyecto, solo deberemos inyectar la instancia de RestTemplate. Durante su configuración en la clase LoadBalancerAutoConfiguration se creará una instancia de RestTemplate a la que le asigna un RibbonLoadBalancerClient como interceptor, de forma que todas las peticiones realizadas con el RestTemplate pasarán por él. El interceptor se encargará de utilizar el balanceador configurado para obtener la instancia final a la que se enviará la petición.

```
@Bean @LoadBalanced
RestTemplate loadBalancedRestTemplate() { return new RestTemplate(); }
```

- Para la configuración, el concepto central en Ribbon es el de cliente nominado. Cada equilibrador de carga forma parte de un conjunto de componentes que trabajan en conjunto para contactar a un servidor remoto a demanda, y el conjunto tiene un nombre que le asignamos (por ejemplo, mediante la anotación @FeignClient). Bajo demanda, Spring Cloud crea un nuevo ApplicationContext para cada cliente nominado utilizando una RibbonClientConfiguration que contiene (entre otras cosas) un ILoadBalancer, un RestClient y un ServerListFilter.
- Una configuración se asocia a un cliente nominado mediante anotaciones @RibbonClient en la clase principal o asociar una configuración por defecto.

© JMA 2016. All rights reserved

Ribbon

```
@Configuration
class CustomRibbonConfig {
    @Bean
    public IRule ribbonRule() { return new BestAvailableRule(); }
    @Bean
    public IPing ribbonPing() { return new PingUrl(); }
    @Bean
    public ServerList<Server> ribbonServerList(IClientConfig config) {
        return new RibbonClientDefaultConfigurationTestsConfig.BazServiceList(config);
    }
    @Bean
    public ServerListSubsetFilter serverListFilter() { return new ServerListSubsetFilter(); }
}

@SpringBootApplication
@RibbonClient(name = "custom", configuration = CustomRibbonConfig.class)
@RibbonClients(defaultConfiguration = DefaultRibbonConfig.class)
public class MsEurekaServiceDiscoveryApplication {
```

© JMA 2016. All rights reserved

Ribbon

```
@Configuration
public class AppConfig {
	@LoadBalanced
	@Bean
	public RestTemplate restTemplate(RestTemplateBuilder builder) {
		return builder.build();
	}
}

@RestController
public class BalanceadoResource {
	@Autowired
	private RestTemplate rest;

	@GetMapping(path = "/peticion")
	public RespuestaDTO get() {
		return rest.getForObject( "http://APPNAME-SERVICE/servicio", RespuestaDTO.class);
	}
}
```

© JMA 2016. All rights reserved

Zuul

- Zuul 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 Zuul. 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.
- Zuul utiliza Hystrix (tolerancia a fallos y resiliencia) y Ribbon (balanceador de carga)

© JMA 2016. All rights reserved

Zuul

- Añadir proyecto:
  - Zuul, Eureka Discovery Client
- Anotar la aplicación:

  @EnableZuulProxy
  @EnableDiscoveryClient
  @EnableEurekaClient
- Configurar:

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

© JMA 2016. All rights reserved

Zuul

- Configurar enrutado:

```
zuul.routes.api.serviceId=API-GATEWAY
zuul.routes.api.path=/api/**
#zuul.routes.api.url=http://localhost:8001

zuul.routes.catalogo-service.serviceId=CATALOGO-SERVICE
zuul.routes.catalogo-service.path=/catalogo/**
#zuul.routes.catalogo-service.url=http://localhost:8002
#zuul.routes.catalogo-service.stripPrefix=true

zuul.routes.cliente.serviceId=CLIENTES-SERVICE
zuul.routes.cliente.path=/clientes/**

zuul.routes.google.path=/search/**
zuul.routes.google.url=https://www.google.com/
```

© JMA 2016. All rights reserved

MONITORIZACIÓN Y RESILIENCIA

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Spring Boot 2.x Actuator

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

© JMA 2016. 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 2016. 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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Hystrix

- Hystrix es una librería que implemente el patrón CircuitBreaker.
- Hystrix nos permite gestionar las interacciones entre servicios en sistemas distribuidos añadiendo lógica de latencia y tolerancia a fallos.
- Su finalidad es mejorar la fiabilidad global del sistema, para ello Hystrix aísla los puntos de acceso de los microservicios, impidiendo así los fallos en cascada a través de los diferentes componentes de la aplicación, proporcionando alternativas de “fallback”, gestionando timeouts, pools de hilos…
- Hystrix encapsula las peticiones a sistemas “externos” para gestionar diversos aspectos de éstas tales como: timeouts, estadísticas de éxito y fallo, semáforos, lógica de gestión de error, propagación de errores en cascada…
- Así por ejemplo si una petición a un servicio alcanza un cierto límite (20 fallos en 5 segundos por defecto), Hystrix abrirá el circuito de forma que no se realizarán más peticiones a dicho servicio, lo que impedirá la propagación de los errores en cascada.

© JMA 2016. All rights reserved

Hystrix

- Encapsula las peticiones a sistemas “externos” y las ejecuta dentro de un hilo separado. Esto es un ejemplo del “command pattern”.
- Cancela las peticiones que exceden el timeout que se les haya definido.
- Gestiona semáforos / pools de hilos para cada petición a sistema “externo”, de forma que si no hay hilos disponibles la petición será rechazada en lugar de encolada previniendo así esperas innecesarias (“failing fast instead of queueing”).
- Gestiona la propagación de errores en cascada.
- Mide estadísticas de peticiones exitosas, fallidas, timeouts…
- Gestiona las peticiones para que en caso de que un sistema externo exceda una cuota de error definida no se le deriven más peticiones.
- Gestiona la ejecución del “fallback” en caso de error en una petición para proteger al usuario del fallo.
- Proporciona un dashboard que integra las métricas capturadas en tiempo real de cada una de las peticiones gestionadas por Hystrix. Estas métricas en tiempo real nos permiten optimizar el “time-to-discovery” en caso de fallos.
- El flujo de datos de métricas generado por Hystrix puede ser interpretado por Turbine para agregar en un único dashboard las peticiones gestionadas por Hystrix de todos los microservicios de nuestro ecosistema

© JMA 2016. All rights reserved

Hystrix

- Añadir al proyecto:
  - Spring Cloud Circuit Breaker > Hystrix
- Anotar la clase principal con @EnableCircuitBreaker
- Cada método que realice peticiones a otro microservicio y quiera contar con un interruptor de circuito debe ser anotado @HystrixCommand donde se indica el método con la acción alternativa o fallback (catch) que se ejecuta cuando el circuito está abierto.
- También se puede indicar el tiemout de espera antes de considerar que la llamada ha fallado con la propiedad, los valores para abrir el circuito: cuando el número de llamadas supera un umbral (requestVolumeThreshold) y un porcentaje de fallos (errorThresholdPercentage). Estos son los básicos para utilizar este patrón de tolerancia a fallos. Tiene algunos valores adicionales más que se pueden configurar para adaptar el patrón a los valores óptimos de la aplicación.

```
@HystrixCommand(fallbackMethod = "getFallback", commandProperties = {
        @HystrixProperty(name = "circuitBreaker.requestVolumeThreshold", value = "4"),
        @HystrixProperty(name = "circuitBreaker.errorThresholdPercentage", value = "50"),
        @HystrixProperty(name = "execution.isolation.thread.timeoutInMilliseconds", value = "1000") })
@GetMapping(path = "/lento")
public RespuestaDTO getLento() {
    return rest.getForObject("http://ESCENARIOS-SERVICE/lento", RespuestaDTO.class);
}
private RespuestaDTO getFallback() {
    return new RespuestaDTO("Fallback");
}
```

© JMA 2016. All rights reserved

Hystrix Dashboard

- Para monitorizar en tiempo real el estado del sistema y de los circuitos se ofrece un dashboard en el que visualizan el número de peticiones que se están realizando, las fallidas, el estado de los circuitos, las que fallan por timeout o las que fallan con error.
- Hystrix ofrece Hystrix Stream que proporciona métricas en tiempo real del estado de los circuit breakers (Hystrix commands) de una aplicación. Para explotar esta información de forma gráfica, proporciona una interfaz llamada Hystrix Dashboard y un agregador de métricas conocido como Turbine.
- Para tener acceso a esta página hay que incluir la dependencia:
  - Spring Cloud Circuit Breaker > Hystrix Dashboard
- Anotar la clase principal con @EnableHystrixDashboard
- La página del sashboard está disponible en la dirección:
  - http://localhost:XXXX/hystrix.
- El dahsboard que ofrece Hystrix es muy básico y con un diseño mejorable, se pueden exponer las métricas de Hystrix en plataformas de análisis y monitorización como Grafana con Prometheus.

© JMA 2016. All rights reserved

Turbine

- Mirar los datos de Hystrix de una instancia individual no es muy útil en términos de monitorizar el estado general de la salud del sistema. Turbine es una aplicación que agrega todos los puntos finales /hystrix.stream relevantes en una combinación /turbine.stream para su uso en el Hystrix Dashboard. Las instancias individuales se encuentran a través de Eureka.
- Agregar al proyecto del Hystrix Dashboard:
  - Spring Cloud Circuit Breaker > Turbine
- Anotar la clase principal con @EnableTurbine
- La página del sashboard está disponible en la dirección:
  - http://localhost:XXXX/hystrix.
- Configurar:

```
server.port=${PORT:8099}
spring.application.name=monitor-server
spring.config.name=${spring.application.name}
turbine.appConfig=clientes-service
turbine.aggregator.clusterConfig=CLIENTES-SERVICE
eureka.instance.appname=${spring.application.name}
eureka.client.serviceUrl.defaultZone=http://localhost:8761/eureka/
eureka.client.registerWithEureka=false
eureka.client.fetchRegistry=true
```

© JMA 2016. All rights reserved

SEGURIDAD

© JMA 2016. 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 ciertamente 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.
- Debe 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

© JMA 2016. 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 2016. 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.cloud.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 2016. 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 2016. 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 2016. 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 que características:
  - "Access-Control-Allow-Origin", "*"
  - "Access-Control-Allow-Headers", "Origin, Content-Type, Accept, Authorization, X-XSRF-TOKEN"
  - "Access-Control-Allow-Methods", "GET, POST, PUT, DELETE, OPTIONS"
  - "Access-Control-Allow-Credentials", "true"
- Soporte: Chrome 3+ Firefox 3.5+ Opera 12+ Safari 4+ Internet Explorer 8+

© JMA 2016. 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 2016. 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 2016. 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 desee.
- El valor predeterminado del UserDetailsService tiene un solo usuario. El nombre del usuario es “user” y la contraseña se genera aleatoriamente al arrancar y se imprime como INFO:
  - Using generated security password: e4918bc4-d8ac-4179-9916-c37825c7eb55
- Puede cambiar el nombre de usuario y la contraseña proporcionando un spring.security.user.name y spring.security.user.password en application.properties.
- Las características básicas predeterminadas en una aplicación web son:
  - Un bean UserDetailsService con almacenamiento en memoria y un solo usuario con una contraseña generada.
  - Inicio de sesión basado en formularios o seguridad básica HTTP (según el tipo de contenido) para toda la aplicación (incluidos los endpoints).
  - Un DefaultAuthenticationEventPublisher para la publicación de eventos de autenticación.

© JMA 2016. 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 2016. 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.
- UserDetailsService es la interfaz utilizada para crear el objeto UserDetails.

© JMA 2016. 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 2016. 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 concuerdan con las esperadas, añadiéndole al objeto Authentication las autorizaciones asociadas a esa identidad o generandon una excepción de tipo AuthenticationException.
- El propio framework ya tiene implementado un gestor de autenticación que es válido para la mayoría de los casos, el ProviderManager. El bean AuthenticationManager es del tipo ProviderManager, lo que significa que actúa de proxy con el AuthenticationProvider.
- Este es el encargado de realizar la comprobación de la validez del nombre de usuario/contraseña asociada y de devolver las autorizaciones permitidas a dicho participante (roles asociados).
- Esta clase delega la autenticación en una lista que engloba a los proveedores y que, por tanto, es configurable. Cada uno de los proveedores tiene que implementar el interfaz AuthenticationProvider.

© JMA 2016. 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 mostrara un mensaje de error.

© JMA 2016. 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 2016. 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 2016. All rights reserved

Configuración de Autenticación

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

```
@Configuration
@EnableWebSecurity
@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 2016. 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 2016. 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 2016. 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 2016. All rights reserved

Configuración

- La sobreescritua 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 2016. 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 2016. 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 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("hasAuthority('ROLE_TELLER')")
@PreAuthorize("authenticated")
@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 2016. 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 agregará. Esto se puede personalizar modificando el defaultRolePrefix en DefaultWebSecurityExpressionHandler.
hasAnyRole([role1,role2])	Se devuelve true si el principal actual tiene alguno de los roles proporcionados (lista de cadenas separadas por comas).
hasAuthority([authority])	Devuelve true si el principal actual tiene la autoridad especificada.
hasAnyAuthority([authority1,authority2])	Se devuelve true si el principal actual tiene alguna de las autorizaciones proporcionadas (se proporciona como una lista de cadenas separadas por comas)
principal	Permite el acceso directo al objeto principal que representa al usuario actual.
authentication	Permite el acceso directo al objeto Authentication actual obtenido del SecurityContext

© JMA 2016. 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 2016. 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 2016. All rights reserved

OAuth 2

© JMA 2016. 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 del sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario este correctamente autenticado y saber quien es.
- Se puede usar con plataformas IDaaS (Identity-as-a-Service) como Auth0 que eliminan la complejidad de la autenticación y su gestión.
- También es posible usarlo para transferir cualquier 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 2016. 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 2016. All rights reserved

Servidor de Autenticación/Autorizacion

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

```
<dependency>
   <groupId>com.auth0</groupId>
   <artifactId>java-jwt</artifactId>
   <version>3.8.1</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 2016. 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 2016. 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 2016. 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 2016. All rights reserved