

ESCUELA:
TECNOLOGÍA



MICROSERVICIOS CON SPRING



indra



ÍNDICE

- Nociones esenciales
- Arquitectura
- Acceso a datos con Spring Data: SQL y NoSQL
- Comunicaciones inter servicio
- Configuración de microservicios con XML y anotaciones
- Creación de microservicios con Spring Boot
- Ejemplos de sistemas cliente



- **NOMBRE APELLIDO PROFESOR**
Javier Martín

- **VER PERFIL COMPLETO:**

 linkedin.com/company/icono-training-consulting

- **CONTACTO**

 training@iconotc.com

Microservicios con
Spring

Contenidos

- Arquitectura de microservicios
- Spring con Spring Boot
- IoC con Spring Core
- Acceso a datos con Spring Data
- Servicios Rest con Spring
- Clientes de los Servicios Rest
- Implantación y Despliegue

© 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/>
- 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>
 - <https://github.com/spring-projects/spring-integration-samples>

© 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 implementar 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 manera, 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 interdisciplinarios, 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 tubos mudos**.
- 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 en lugar de complejos protocolos como WS o BPEL o orquestación de una herramienta central.

© JMA 2016. All rights reserved

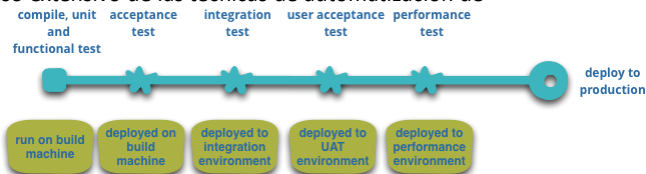
Gestión descentralizada de datos

- 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, es que las aplicaciones deben diseñarse de manera que puedan tolerar el fallo de los servicios. Cualquier llamada de 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 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 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: Debemos 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 actualizar la producción.
- Requiere un compromiso a largo plazo con una pila de tecnología: obliga a casarse con una tecnología (y, en algunos casos, con una versión particular de esa tecnología) que se eligió al inicio del desarrollo, puede que hace mucho tiempo. Puede ser difícil adoptar de manera incremental una tecnología más nueva. No permite utilizar otros lenguajes o entornos de desarrollo. Además, si la aplicación utiliza una plataforma que posteriormente se vuelve obsoleta, puede ser un desafío migrar gradualmente la aplicación a un marco más nuevo y mejor.

© JMA 2016. All rights reserved

Microservicios: Beneficios

- 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.
- 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.
- 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 de servicios diferentes.
- Mayor consumo de memoria. 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 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

Cómo descomponer la aplicación en servicios

- Descomponer por capacidad empresarial y definir servicios que correspondan a las capacidades empresariales.
- Descomponer por subdominio DDD.
- 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)
- 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

S.O.L.I.D.

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

© JMA 2016. All rights reserved

S.O.L.I.D.

S

- Single Responsibility Principle (SRP)
- Principio de responsabilidad única

O

- Open/Closed Principle (OCP)
- Principio de abierto-cerrado

L

- Liskov Substitution Principle (LSP)
- Principio de sustitución de Liskov

I

- Interface Segregation Principle (ISP)
- Principio de segregación de interfaces

D

- Dependency Inversion Principle (DIP)
- Principio de inversión de dependencias

© JMA 2016. All rights reserved

S.O.L.I.D.

- Principio de Responsabilidad Única

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

© JMA 2016. All rights reserved

S.O.L.I.D.

- Principio de Abierto/Cerrado

- “You should be able to extend a classes behavior, without modifying it.”
- El segundo principio de SOLID lo formuló Bertrand Meyer en 1988 en su libro “Object Oriented Software Construction” y dice: “Deberías ser capaz de extender el comportamiento de una clase, sin modificarla”. En otras palabras: las clases que usas deberían estar abiertas para poder extenderse y cerradas para modificarse.
- En su blog Robert C. Martin defendió este principio que a priori puede parecer una paradoja.
- Es importante tener en cuenta el Open/Closed Principle (OCP) a la hora de desarrollar clases, librerías, frameworks, microservicios.

© JMA 2016. All rights reserved

S.O.L.I.D.

- Principio de Sustitución de Liskov

“Derived classes must be substitutable for their base classes.”

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

© JMA 2016. All rights reserved

S.O.L.I.D.

- Principio de Segregación de la Interfaz

“Make fine grained interfaces that are client specific.”

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

© JMA 2016. All rights reserved

S.O.L.I.D.

- Principio de Inversión de Dependencias
 - “Depend on abstractions, not on concretions.”
 - Llegamos al último principio: “Depende de abstracciones, no de clases concretas”.
 - Así, Robert C. Martin recomienda:
 - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Ambos deberían depender de abstracciones.
 - Las abstracciones no deberían depender de los detalles. Los detalles deberían depender de las abstracciones.
 - El objetivo del Dependency Inversion Principle (DIP) consiste en reducir las dependencias entre los módulos del código, es decir, alcanzar un bajo acoplamiento de las clases.

© 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 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:
 - Composición de API: <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

- Construye un almacenamiento de datos separado por cada microservicio.
- Construye y despliega cada microservicio por separado.
- Centralizar la configuración
- Desplegar con contenedores
- Automatizar el despliegue
- Cultura Cloud: compra lo que necesites, cuando lo necesites y paga por lo que uses.
- Cultura anti-burocracia: libertad con responsabilidad

© 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 está 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

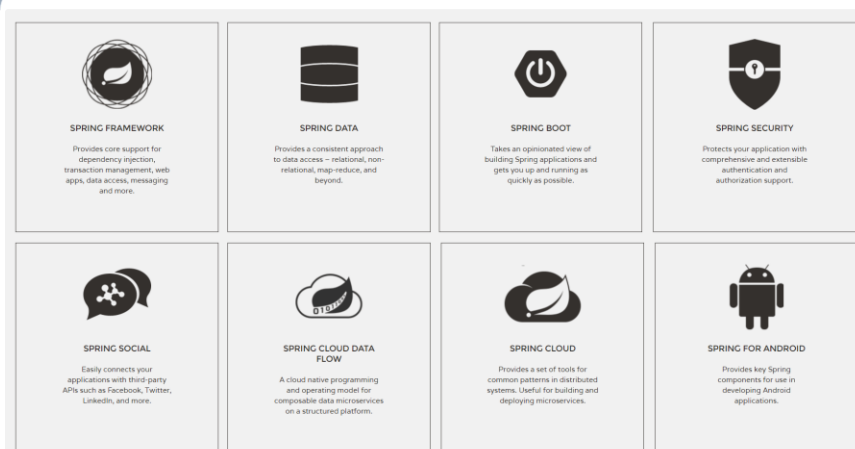
© JMA 2016. All rights reserved

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 se 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

Con Eclipse

- Descargar Hibernate:
 - <http://hibernate.org/orm/downloads/>
- Descargar e instalar JDK:
 - <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- Descargar y descomprimir Eclipse:
 - <https://www.eclipse.org/downloads/>
- Añadir a Eclipse las Hibernate Tools
 - Help > Eclipse Marketplace: JBoss Tools
- Crear una User Librarie para Hibernate
 - Window > Preferences > Java > Build Path > User Libraries > New
 - Add External JARs: \lib\required
- Descargar y registrar la definición del driver JDBC
 - Window > Preferences > Data Management > Connectivity > Driver Definition > Add

© JMA 2016. All rights reserved

Instalación

- <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 → Spring Boot → Spring Started Project
- Dependencias
 - Web
 - JPA
 - JDBC (o proyecto personalizado)

© JMA 2016. All rights reserved

Dependencias opcionales

- Serialización XML a cliente

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

© 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 settings
`spring.datasource.url=jdbc:mysql://localhost:3306/sakila`
`spring.datasource.username=root`
`spring.datasource.password=root`
`spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver`

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

`server.port=8080`
- Repetir con `src/test/resources/application.properties`

© JMA 2016. All rights reserved

Oracle Driver con Maven

- <http://www.oracle.com/technetwork/database/features/jdbc/index-091264.html>
- Instalación de Maven:
 - Descargar y descomprimir (<https://maven.apache.org>)
 - Añadir al PATH: `C:\Program Files\apache-maven\bin`
 - Comprobar en la consola de comandos: `mvn -v`
- Descargar el JDBC Driver de Oracle (ojdbc6.jar):
 - <https://www.oracle.com/technetwork/apps-tech/jdbc-112010-090769.html>
- Instalar el artefacto ojdbc en el repositorio local de Maven
 - `mvn install:install-file -Dfile=Path/to/your/ojdbc6.jar -DgroupId=com.oracle -DartifactId=ojdbc6 -Dversion=11.2.0 -Dpackaging=jar`
- En el fichero `pom.xml`:

```
<dependency>
  <groupId>com.oracle</groupId>
  <artifactId>ojdbc6</artifactId>
  <version>11.2.0</version>
</dependency>
```

© JMA 2016. All rights reserved

Configuración del proxy: Maven

- Crear fichero setting.xml o editar %MAVEN_ROOT%/conf/setting.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0 http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <localRepository>C:\directorio\local\.m2\repository</localRepository>
  <proxies>
    <proxy>
      <id>optional</id>
      <active>true</active>
      <protocol>http</protocol>
      <username>usuario</username>
      <password>contraseña</password>
      <host>proxy.dominion.com</host>
      <port>8080</port>
      <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
    </proxy>
  </proxies>
</settings>
```
- Referenciarlo en Window → Preferences → Maven → User setting → User setting , browse..., seleccionar fichero recién creado, aceptar, update setting, aplicar y cerrar.

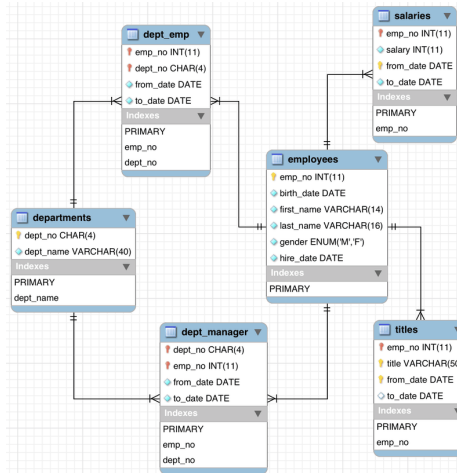
© JMA 2016. All rights reserved

Instalación de MySQL

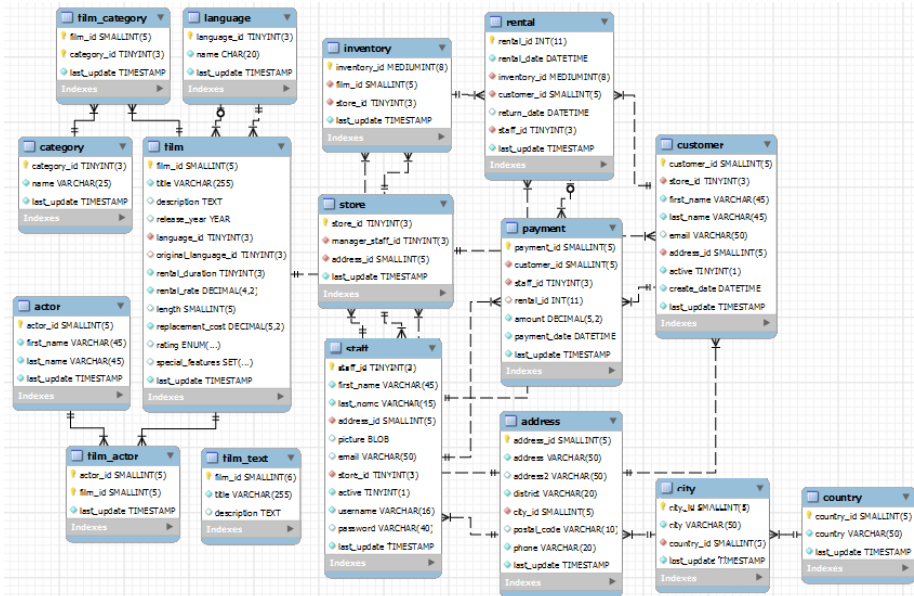
- Descargar e instalar:
 - <https://mariadb.org/download/>
- Incluir en la seccion [mysqld] de %MYSQL_ROOT%/data/my.ini
 - default_time_zone='+01:00'
- Descargar bases de datos de ejemplos:
 - <https://dev.mysql.com/doc/index-other.html>
- Instalar bases de datos de ejemplos:
 - mysql -u root -p < employees.sql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

© JMA 2016. All rights reserved

Modelos de datos



© JMA 2016. All rights reserved



© 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

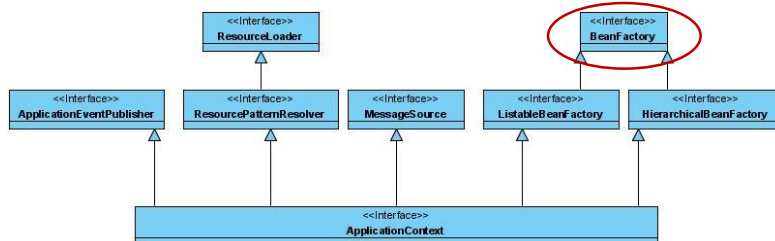
- Los beans se corresponden a los objetos reales que conforman la aplicación y que requieren ser inyectables: los objetos de la capa de servicio, los objetos de acceso a datos (DAO), los objetos de presentación (como las instancias Action de Struts), los objetos de infraestructura (como Hibernate SessionFactories,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

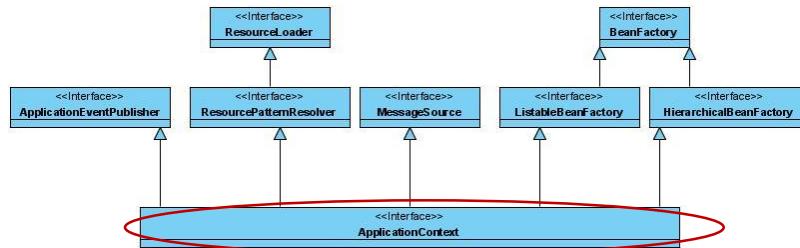
- Denominamos Bean Factory al contenedor Spring.
- Cualquier Bean Factory permite la configuración y la unión de objetos mediante la inyección de dependencia.
- Este Bean Factory también permite una gestión del ciclo de vida de los beans instanciados en él.
- Todos los contenedores Spring (Bean Factory) 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
 - @Scope
 - @Component
 - @Repository
 - @Service
 - @Controller
- 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 `@Autowire`:
 - En atributos:
`@Autowire`
`private MyBeans myBeans;`
 - En propiedades (setter):
`@Autowire`
`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.
`@Autowire(required=false) private MyBeans myBeans;`
- Se puede completar `@Autowire` 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`, `.yaml`, `.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 contenedor.
- 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

ACCESO A DATOS CON SPRING DATA

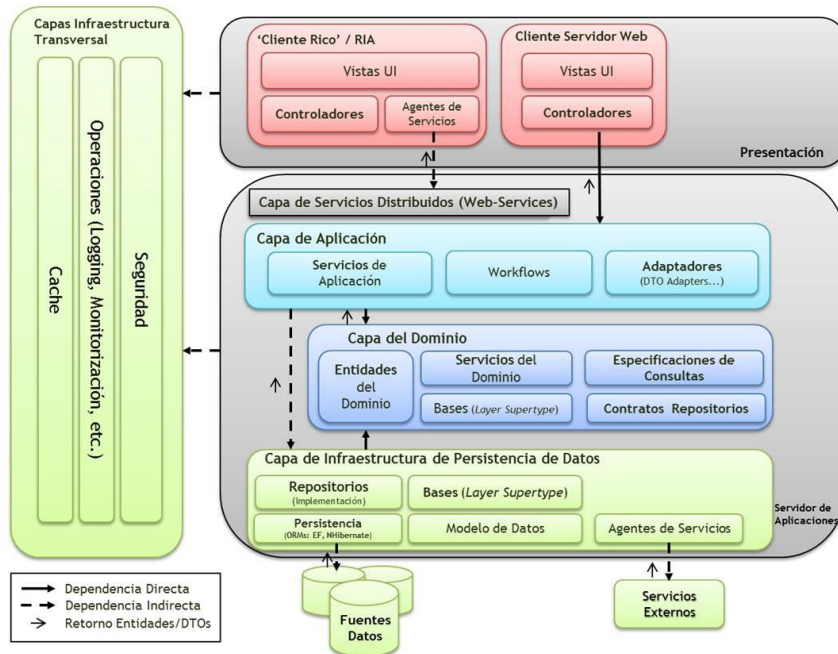
© JMA 2016. All rights reserved

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

Arquitectura N-Capas con Orientación al Dominio



Modelos: Entidades

- Una entidad 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

Anotaciones JPA

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

© JMA 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 sincronización 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

Mapecto 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 a necesitar una interfaz que extienda uno de los siguientes interfaces:
 - `CrudRepository<T,ID>`
 - `count()`, `delete(T entity)`, `deleteAll()`, `deleteAll(Iterable<? extends T> entities)`, `deleteById(ID id)`, `existsById(ID id)`, `findAll()`, `findAllById(Iterable<ID> ids)`, `findById(ID id)`, `save(S entity)`, `saveAll(Iterable<S> entities)`
 - `PagingAndSortingRepository<T,ID>`
 - `findAll(Pageable pageable)`, `findAll(Sort sort)`
 - `JpaRepository<T,ID>`
 - `deleteAllInBatch`, `deleteInBatch`, `flush`, `findOne`, `saveAll`, `saveAndFlush`
- 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 específica para crear dichos nombres:

```
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, TopNumFilas y FirstNumFilas
- Expresión de propiedad: *ByPropiedad*
 - Operador (Between, LessThan, GreaterThan, Like, ...) por defecto equal.
 - Se pueden concatenar varias con And y Or
 - Opcionalmente admite el indicador IgnoreCase y AllIgnoreCase.
- Opcionalmente, *OrderByPropiedadAsc* para ordenar,
 - se puede sustituir Asc por Desc, admite varias expresiones de ordenación.
- Parámetros:
 - un parámetro por cada operador que requiera valor y debe ser del tipo apropiado
- Parámetros opcionales:
 - Pageable, Sort

© 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, findByFirstnames, 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.firstname) = 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

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());  
    }  
}
```

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)
```

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 (múltiples relaciones requieren asignar nombres únicos).

```
@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

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.
- 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  
private Validator validator;  
Set<ConstraintViolation<@Valid Persona>> constraintViolations =  
    validator.validate( persona );  
Set<ConstraintViolation<@Valid Persona>> constraintViolations =  
    validator.validateProperty( persona, "nombre" );
```

© JMA 2016. All rights reserved

Validaciones

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

© JMA 2016. All rights reserved

Validaciones

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

© JMA 2016. All rights reserved

Validaciones

- @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

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

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 Agregado (Aggregate)

- Una Agregación es un grupo de objetos asociados 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

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, ...

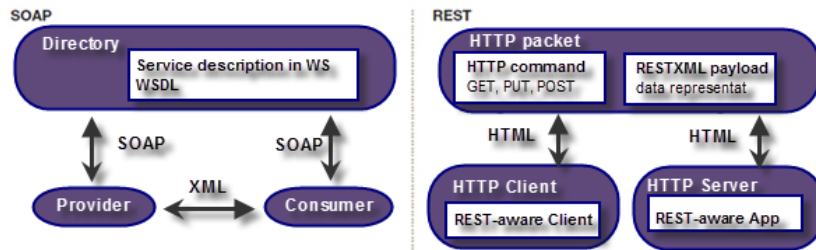
© JMA 2016. All rights reserved

Introducción a los Servicios REST

- REST y SOAP son muy diferentes.
 - SOAP:
 - Es un protocolo de mensajería
 - REST:
 - Es un estilo de arquitectura de software para sistemas hipermedia distribuidos.
 - Sistemas en los que el texto, gráficos, audio y otros medios de comunicación se almacenan en una red e interconectados a través de hipervínculos .
- WWW es un sistema REST (la web estática, la web dinámica no).
 - En la Web, HTTP es a la vez un protocolo de transporte y un sistema de mensajería (las peticiones y respuestas HTTP son mensajes).
- Estos requisitos REST son:
 - Se publican Recursos (Un dato, una operación, un numero de empleado, el empleado 44, etc.)
 - Los servicios REST no publican un conjunto de métodos u operaciones, publican RECURSOS.
 - Cada recurso dispone de un identificador único.
 - Cada recurso debe de tener una o varias representaciones de su estado (XML, HTML, PDF, etc)

© JMA 2016. All rights reserved

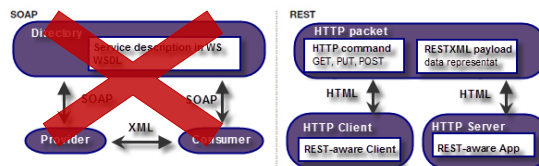
Introducción a los Servicios REST



© JMA 2016. All rights reserved

REST (REpresentation State Transfer)

- Nos permite utilizar cualquier interfaz web simple que utiliza HTTP, sin las abstracciones/restricciones de los protocolos basados en patrones de intercambio de mensajes.
- Los servicios que siguen los principios de REST habitualmente se denominan RESTful.
- REST se basa en varios componentes principales:
 - Recursos
 - URI
 - Representaciones
 - Solicitudes HTTP



© JMA 2016. All rights reserved

HTTP

- HTTP es una pieza fundamental en World Wide Web, y especifica como intercambiar entre cliente y servidor recursos web.
- Es un protocolo idóneo para implementar servicios web, ya que sigue los principios REST.
- Características de HTTP:
 - Es un protocolo de nivel de aplicación y algo de presentación.
 - Está diseñado para ser ejecutado sobre TCP o sobre TLS/SSL.
 - Se basa en un paradigma sencillo de petición/respuesta, es decir, es un protocolo stateless.

© 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)

The diagram illustrates the structure of an HTTP request message. It is divided into three main sections, each highlighted with a red box and a numbered blue circle:

- 1** (Top): The request line, containing the HTTP method, the URI, and the protocol version. In the example, it is `POST /server/payment HTTP/1.1`.
- 2** (Middle): The headers, which are key-value pairs. In the example, they include `Host: www.myserver.com`, `Content-Type: application/x-www-form-urlencoded`, `Accept: application/json`, `Accept-Encoding: gzip, deflate, sdch`, `Accept-Language: en-US,en;q=0.8`, `Cache-Control: max-age=0`, and `Connection: keep-alive`.
- 3** (Bottom): The body of the request, which contains the data being sent. In the example, it is `orderId=34fry423&payment-method=visa&card-number=2345123423487648&sn=345`.

© 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.

```
HTTP/1.1 201 Created
Content-type: application/json;charset=utf-8
Location: https://www.myserver.com/services/payment/3432
Cache-Control: max-age=21600
Connection: close
Date: Mon, 23 Jul 2012 14:20:19 GMT
ETag: "2ec8-3e3073913b100"
Expires: Mon, 23 Jul 2012 20:20:19 GMT

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

© 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

© JMA 2016. All rights reserved

URI (Uniform Resource Identifier)

- Los URI son los identificadores globales de recursos en la web, y actúan de manera efectiva como UUIDs REST.
- Hay 2 tipos de URIs : URL y URN
 - URLs Identifican un recurso de red mediante una IP o un DNS
 - URNs son simples UUIDs lógicos con un espacio de nombres asociados
- URI es una cadena de caracteres corta, que identifica inequívocamente un recurso y que tienen el siguiente formato
 <esquema>://<host>:puerto/<ruta><querystring><fragmento>
 - Esquema = Indican que protocolo hay que utilizar para usar el recurso (http o https)
 - Host = Indica el lugar donde encontraremos el recurso (por IP o por dominio)
 - Puerto = Puerto por donde se establece la conexión (80 o 443)
 - Ruta = Ruta del recurso dentro del servidor, está separado por /
 - queryStrng = Parámetros adicionales, separados por ? o por &
 - Fragmento = Separado por #

© JMA 2016. All rights reserved

URI (Uniform Resource Identifier)

- 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.
- 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

© JMA 2016. All rights reserved

Tipos MIME

- Para negociar el formato entre el cliente y el servidor se utilizan las cabeceras:
 - Petición
 - En la cabecera ACCEPT se envía una lista de tipos MIME que el cliente entiende.
 - En caso de enviar contenido en el cuerpo, la cabecera CONTENT-TYPE indica en que formato MIME está codificado.
 - Respuesta
 - El servidor selecciona el tipo que más le interese de entre todos los especificados en la cabecera ACCEPT, y devuelve la respuesta indicando con la cabecera CONTENT-TYPE el formato del cuerpo.
- La lista de tipos MIME se especifica en la cabecera (ACCEPT) mediante lo que se llama una lista separada por comas de tipos (media range). También pueden aparecer expresiones de rango, por ejemplo
 - */* indica cualquier tipo MIME
 - image / * indica cualquier formato de imagen
- 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 <i>Página no encontrada</i>
405	Method Not Allowed	El servidor no permite el uso del método utilizado por la petición, por ejemplo por utilizar el método GET cuando el servidor sólo permite el método POST
406	Not Acceptable	El tipo de contenido solicitado por el navegador no se encuentra entre la lista de tipos de contenidos que admite, por lo que no se envía en la respuesta
407	Proxy Authentication Required	Similar al código 401, indica que el navegador debe obtener autorización del proxy antes de que se le pueda enviar el contenido solicitado
408	Request Timeout	El navegador ha tardado demasiado tiempo en realizar la petición, por lo que el servidor la descarta

© JMA 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 de 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

Uso de la cabecera

- Request: Método /uri?parámetros
 - GET: Recupera el recurso
 - Todos: Sin parámetros
 - Uno: Con parámetros
 - POST: Crea un nuevo recurso
 - PUT: Edita el recurso
 - DELETE: Elimina el recurso
- Accept: Indica al servidor el formato o posibles formatos esperados, utilizando MIME.
- Content-type: Indica en que formato está codificado el cuerpo, utilizando MIME
- Response: Código de estado con el que el servidor informa del resultado de la petición.

© JMA 2016. All rights reserved

Objetivos de los servicios REST

- Desacoplar el cliente del backend
- Mayor escalabilidad
 - Sin estado en el backend.
- Separación de problemas
- División de responsabilidades
- API uniforme para todos los clientes
 - Disponer de una interfaz uniforme (basada en URIs)

© 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

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
}
```

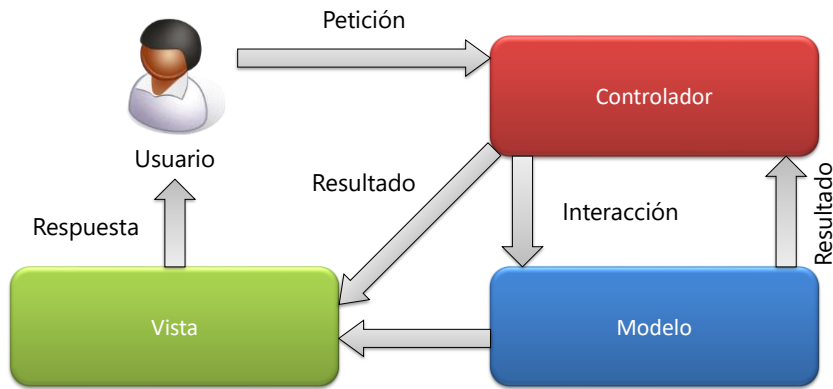
© JMA 2016. All rights reserved

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

El patrón MVC



© JMA 2016. All rights reserved

El patrón MVC



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



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



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

© JMA 2016. All rights reserved

Recursos

- Son clases Java con la anotación `@RestController` 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 = "/países/{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

- `consumes`: formatos MIME permitidos del encabezado `Content-type`

```
@PostMapping(path = "/pets", consumes = "application/json")
public void addPet(@RequestBody Pet pet) {
```
- `produces` : formatos MIME permitidos del encabezado `Accept`

```
@GetMapping(path = "/pets/{petId}", produces =
"application/json;charset=UTF-8")
@ResponseBody
public Pet getPet(@PathVariable String petId) {
```
- `params`: valores permitidos de los `QueryParams`
- `headers`: valores permitidos de los encabezados

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

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

© JMA 2016. All rights reserved

Inyección de Parámetros

Anotación	Descripción
@RequestBody	Para acceder al cuerpo de la solicitud HTTP. El contenido del cuerpo se convierte al tipo de argumento del método declarado utilizando implementaciones <code>HttpMessageConverter</code> .
@RequestPart	Para acceder a una parte en una solicitud multipart/form-data, convertir el cuerpo de la parte con un <code>HttpMessageConverter</code> .
@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 <code>@SessionAttributes</code> 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` 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 diferentes 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`, etc.
- Spring HATEOAS ahora proporciona una `ControllerLinkBuilder` que permite crear enlaces apuntando a clases de controladores:
`import static org.springframework.hateoas.mvc.ControllerLinkBuilder.*;`
- Para añadir una referencia a si mismo:
`personaDTO.add(linkTo(PersonaResource.class).withSelfRel());`
`personaDTO.add(linkTo(PersonaResource.class).slash(personaDTO.getId()).withSelfRel());`
- Para añadir una referencia a si mismo como método:
`personaDTO.add(linkTo(PersonaResource.class.getMethod("get", Long.class), personaDTO.getId()).withSelfRel());`
- Para crear una referencia a un elemento interno:
`personaDTO.add(linkTo(PersonaResource.class).slash(personaDTO.getId()).slash("direcciones").withRel("direcciones"));`

© JMA 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

Swagger

<https://swagger.io/>

- Swagger 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

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 RESTFul

- **@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(@PathVariable int id) throws NotFoundException {
```
- **@ApiResponse**, **@ApiResponses**: 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

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 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", rel="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 getPersonald();`
`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

Seguridad

- 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 de servicios de seguridad aplicables para sistemas basados en la arquitectura JEE, enfocado particularmente sobre proyectos contruidos 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

Seguridad: Configuración

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class SecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsService userDetailsService;
    @Autowired
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(userDetailsService).passwordEncoder(passwordEncoder());
    }
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.csrf().disable()
            .authorizeRequests()
                .antMatchers("/**").permitAll()
                .antMatchers("/privado/**").authenticated() // @PreAuthorize("authenticated")
                .antMatchers("/admin/**").hasRole("ADMIN") // @PreAuthorize("hasRole('ROLE_ADMIN')")
                .and().httpBasic();
    }
    @Bean
    public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
}
```

© JMA 2016. All rights reserved

Seguridad: UserDetailsService

```
@Service
@Transactional
public class UserDetailsServiceImpl implements UserDetailsService {
    @Override
    public UserDetails loadUserByUsername(final String username) throws UsernameNotFoundException {
        // Se accede a UserDao para buscar el usuario y obtener su clave y roles
        if ("user".equals(username)) {
            return this.userBuilder(username, new BCryptPasswordEncoder().encode("123456"), "USER");
        } else if ("manager".equals(username)) {
            return this.userBuilder(username, new BCryptPasswordEncoder().encode("123456"), "MANAGER");
        } else if ("admin".equals(username)) {
            return this.userBuilder(username, new BCryptPasswordEncoder().encode("123456"), "USER", "MANAGER",
                "ADMIN");
        } else {
            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

JWT: JSON Web Tokens

<https://jwt.io>

- JSON Web Token (JWT) es un estándar abierto (RFC-7519) basado en JSON para crear un token que sirva para enviar datos entre aplicaciones o servicios y garantizar que sean válidos y seguros.
- El caso más común de uso de los JWT es para manejar la autenticación en aplicaciones móviles o web. Para esto cuando el usuario se quiere autenticar manda sus datos de inicio de sesión al servidor, este genera el JWT y se lo manda a la aplicación cliente, posteriormente en cada petición el cliente envía este token que el servidor usa para verificar que el usuario 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

CLIENTES DE LOS SERVICIOS REST

© 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 endpoint REST en una sola línea. Para incorporarlo en Maven:

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

- Para poder inyectar la dependencia:

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

© JMA 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
 - UnknownHttpStatusException: en caso de un estado HTTP desconocido.
- Para vigilar las excepciones:

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

© JMA 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();
```

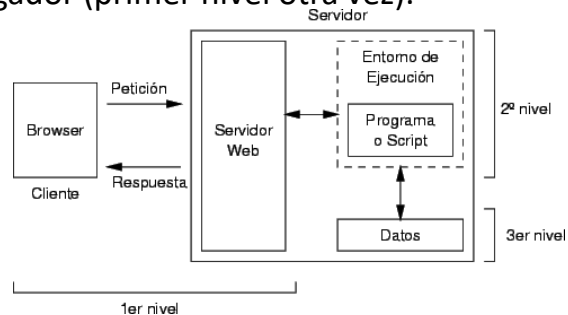
© JMA 2016. All rights reserved

ASYNCHRONOUS JAVASCRIPT AND XML

© JMA 2016. All rights reserved

Introducción

- Una aplicación Web típica recogerá datos del usuario (primer nivel), los enviará al servidor, que ejecutará un programa (segundo y tercer nivel) y cuyo resultado será formateado y presentado al usuario en el navegador (primer nivel otra vez).



© JMA 2016. All rights reserved

Introducción

- AJAX, acrónimo de Asynchronous JavaScript And XML (JavaScript asíncrono y XML), es una técnica de desarrollo web para crear aplicaciones interactivas o RIA (Rich Internet Applications). Estas aplicaciones se ejecutan en el cliente, es decir, en el navegador de los usuarios mientras se mantiene la comunicación asíncrona con el servidor en segundo plano. De esta forma es posible realizar cambios sobre las páginas sin necesidad de recargarlas, mejorando la interactividad, velocidad y usabilidad en las aplicaciones.
- Ajax es una tecnología asíncrona, en el sentido de que los datos adicionales se solicitan al servidor y se cargan en segundo plano sin interferir con la visualización ni el comportamiento de la página. Ajax no constituye una tecnología en sí, sino que es un término que engloba a un grupo de éstas que trabajan conjuntamente.
- JavaScript es el lenguaje interpretado (scripting language) en el que normalmente se efectúan las funciones de llamada de Ajax mientras que el acceso a los datos se realiza mediante XMLHttpRequest, objeto disponible en los navegadores actuales. En cualquier caso, no es necesario que el contenido asíncrono esté formateado en XML.
- Ajax es una técnica válida para múltiples plataformas y utilizable en muchos sistemas operativos y navegadores dado que está basado en estándares abiertos como JavaScript y Document Object Model (DOM).

© JMA 2016. All rights reserved

XMLHttpRequest

- XMLHttpRequest (XHR), también conocido como XMLHTTP (Extensible Markup Language / Hypertext Transfer Protocol), es una interfaz empleada para realizar peticiones HTTP y HTTPS a servidores Web.
- Para los datos transferidos se usa cualquier codificación basada en texto, incluyendo: texto plano, XML, JSON, HTML y codificaciones particulares específicas.
- El navegador implementa la interfaz como una clase de la que una aplicación cliente puede generar tantas instancias como necesite y permita el navegador para manejar el diálogo con el servidor.
- La primera versión de la interfaz XMLHttpRequest fue desarrollada por Microsoft que la introdujo en la versión 5.0 de Internet Explorer utilizando un objeto ActiveX. A partir de la versión 7 la interfaz se ofrece de manera integrada.
- El proyecto Mozilla incorporó la primera implementación integrada en la versión 1.0 de la Suite Mozilla en 2002. Esta implementación sería seguida por Apple a partir de Safari 1.2, Opera Software a partir del Opera 8.0 e iCab desde la versión 3.0b352.
- El World Wide Web Consortium presentó el 27 de septiembre de 2006 el primer borrador de una especificación estándar de la interfaz.

© JMA 2016. All rights reserved

Propiedades

Propiedad	Descripción	
readyState	0	No inicializado (objeto creado, pero no se ha invocado el método open)
	1	Cargando (objeto creado, pero no se ha invocado el método send)
	2	Cargado (se ha invocado el método send, pero el servidor aún no ha respondido)
	3	Interactivo (descargando, se han recibido algunos datos, aunque no se puede emplear la propiedad responseText)
	4	Completo (se han recibido todos los datos de la respuesta del servidor)
responseText	El contenido de la respuesta del servidor en forma de cadena de texto	
responseXML	El contenido de la respuesta del servidor en formato XML. El objeto devuelto se puede procesar como un objeto DOM	
status	El código de estado HTTP devuelto por el servidor (200 para una respuesta correcta, 404 para "No encontrado", 500 para un error de servidor, etc.)	
statusText	El código de estado HTTP devuelto por el servidor en forma de cadena de texto: "OK", "Not Found", "Internal Server Error", etc.	

© JMA 2016. All rights reserved

Métodos y eventos

Método	Descripción
abort()	Detiene la petición actual.
getAllResponseHeaders()	Devuelve una cadena de texto con todas las cabeceras de la respuesta del servidor.
getResponseHeader("cabecera")	Devuelve una cadena de texto con el contenido de la cabecera solicitada.
open("metodo", "url")	Establece los parámetros de la petición que se realiza al servidor. Los parámetros necesarios son el método HTTP empleado y la URL destino.
send(contenido)	Realiza la petición HTTP al servidor
setRequestHeader("cabecera", "valor")	Permite establecer cabeceras personalizadas en la petición HTTP. Se debe invocar entre el open() y el send().
onreadystatechange	Evento. Se invoca cada vez que se produce un cambio en el estado de la petición HTTP.

© JMA 2016. All rights reserved

Pasos a seguir

1. Obtener XMLHttpRequest
2. Crear y asignar el controlador del evento `onreadystatechange`.
3. Abrir conexión: `open(method, url, async)`:
4. Opcional. Añadir cabeceras.
5. Opcional. Serializar datos a enviar vía POST.
6. Enviar petición: `send()`

© JMA 2016. All rights reserved

Obtener XMLHttpRequest

- Los navegadores que siguen los estándares (Firefox, Safari, Opera, Internet Explorer 7+) implementan el objeto XMLHttpRequest de forma nativa, por lo que se puede obtener a través del objeto window. Los navegadores obsoletos (Internet Explorer 5 y 6) implementan el objeto XMLHttpRequest como un objeto de tipo ActiveX.

```
var xmlhttp;
if (window.XMLHttpRequest) {
    //El explorador implementa la interfaz de forma nativa
    xmlhttp = new XMLHttpRequest();
} else if (window.ActiveXObject) {
    //El explorador permite crear objetos ActiveX
    try {
        xmlhttp = new ActiveXObject("MSXML2.XMLHTTP");
    } catch (e) {
        try {
            xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
        } catch (e) {}
    }
}
if (!xmlhttp) {
    alert("No ha sido posible crear una instancia de XMLHttpRequest");
}
```

© JMA 2016. All rights reserved

Controlador del evento onreadystatechange

- readyState: cuando vale 4 (Completo: se han recibido todos los datos de la respuesta del servidor)
 - status: cuando vale 200 (OK: La petición se ha recibido correctamente y se está enviando la respuesta)
 - responseText: El contenido de la respuesta del servidor en forma de cadena de texto
 - responseXML: El contenido de la respuesta del servidor en formato XML.
- ```
xmlhttp.onreadystatechange = function () {
 if (xmlhttp.readyState == 4)
 if (xmlhttp.status == 200) {
 xmlDoc = xmlhttp.responseXML;
 // Tratamiento de los datos recibidos
 } else {
 // Tratamiento de excepción
 }
};
```

© JMA 2016. All rights reserved

## Enviar petición

- Abrir conexión: open(method, url, async):
  - method: Verbo HTTP (GET, POST, ...)
  - async: Opcional, marcar con false para comportamiento síncrono

```
xmlhttp.open("GET", "demo_get.asp?nocache=" + Math.random());
xmlhttp.open("POST", "demo_post.asp");
```
- Opcional. Añadir cabeceras.

```
xmlhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");
```
- Opcional. Serializar datos a enviar vía POST.
  - Reunir los datos a enviar en una cadena formada por pares "nombre=valor" concatenados con ampersand (&).

```
var nombre= document.getElementById("fldNombre");
var datos="nombre="+encodeURIComponent(nombre)
 +"&apellidos="+encodeURIComponent(apellidos);
```
- Enviar petición: send()

```
xmlhttp.send(datos);
```

© JMA 2016. All rights reserved

# Texto plano, HTML y JavaScript

- Texto plano

```
var rslt = http_request.responseText;
document.getElementById("myDiv").innerHTML=rslt;
```

- HTML

```
var rslt = http_request.responseText;
document.getElementById("myDiv").innerHTML=rslt;
```

- JavaScript

```
var respuesta = http_request.responseText;
eval(respuesta);
```

© JMA 2016. All rights reserved

## XML

- Serializar:

```
var datos="<Datos>";
datos+="<Nombre>"+nombre+"</Nombre>";
datos+="<Apellidos>"+apellidos+"</Apellidos>";
datos+="</Datos>";
```

- Recibir:

```
xmlDoc=xmlhttp.responseText;
txt="";
x=xmlDoc.getElementsByTagName("Provincias");
for (i=0;i<x.length;i++) {
 txt=txt + x[i].childNodes[0].nodeValue + "
";
}
document.getElementById("myDiv").innerHTML=txt;
```

© JMA 2016. All rights reserved

# JSON

- Serializar:

```
var datos= JSON.stringify(objeto_json);
var datos='{ "Datos": {';
datos+=' "Nombre":"' +nombre+' "';
datos+=' , "Apellidos":"' +apellidos+' "';
datos+=' } }';
```

- Recibir:

```
var respuesta = http_request.responseText;
var objeto_json= JSON.parse(respuesta).Provincias;
//var objeto_json = eval("(" +respuesta+")").Provincias;
txt="";
for (i=0;i<objeto_json.length;i++) {
 txt=txt + objeto_json[i].Nombre + "
";
}
document.getElementById("myDiv").innerHTML=txt;
```

© JMA 2016. All rights reserved

## Detener las peticiones

```
...
// Fijar un temporizador que aborte la petición
var temporizador = setTimeout(function() {
 xmlhttp.abort();
 alert(...);
}, 18000); // 2 minutos
xmlhttp.onreadystatechange = function () {
 if (xmlhttp.readyState == 4) {
 clearTimeout(temporizador);
 if (xmlhttp.status == 200) {
 // Eliminar el temporizador, innecesario
 }
 }
}
...
```

© JMA 2016. All rights reserved

# Indicador de descarga

```
...
var temporizador = setTimeout(function() {
 document.getElementById("trabajandoAJAX").style.display="none";
 xmlhttp.abort();
 alert(...);
}, 18000); // 30 segundos
// Muestra el indicador hasta ahora oculto
document.getElementById("trabajandoAJAX").style.display = "block";
xmlhttp.onreadystatechange = function () {
 if (xmlhttp.readyState == 4)
 try {
 if (xmlhttp.status == 200) {
 ...
 } finally {
 // Oculta el indicador
 document.getElementById("trabajandoAJAX").style.display="none";
 }
 }
 ...
}
```

© JMA 2016. All rights reserved

## IMPLANTACIÓN Y DESPLIEGUE

© 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.

© JMA 2016. All rights reserved

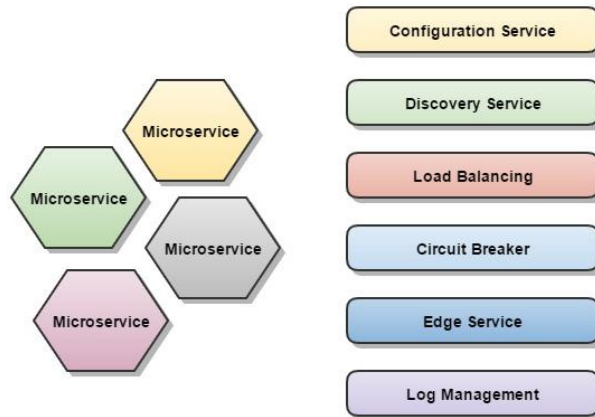
## Implantación

- 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.
- 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.
- 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.

© JMA 2016. All rights reserved

# Modelo de referencia

- 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.
- 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.
- Servidor de Autorización
  - Para implementar la capa de seguridad (recomendable en la capa de servicios API)
- Monitorización
  - Para poder disponer de mecanismos y dashboard para monitorizar aspectos de los nodos como, salud, carga de trabajo...

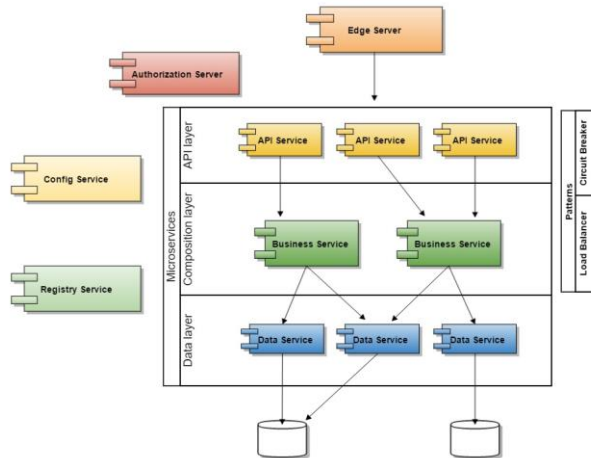
© JMA 2016. All rights reserved

# 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.

© JMA 2016. All rights reserved

Este diagrama ilustra la arquitectura de un sistema de microservicios, organizada en capas y componentes:

- Clientes:** El punto de entrada principal al sistema.
- Servidor perimetral:** Actúa como el primer punto de contacto, integrando **Zuul** para la gestión de rutas.
- Balanceo de carga y Tolerancia a fallos:** Se implementa mediante **Ribbon** y **Hystrix** para asegurar la disponibilidad y el rendimiento.
- Microsservicios:** El núcleo del sistema, dividido en:
  - Microsservicios públicos:** Utilizan tecnologías como **Spring**.
  - Microsservicios Privados:** Utilizan tecnologías como **Node JS** y **.NET**.
- Servicio de registro:** Utiliza **Eureka** para el descubrimiento de servicios.
- Servidor de configuración central:** Utiliza **Spring Cloud Config** para gestionar la configuración.
- Repositorio de configuración:** Almacena la configuración en **Git**.
- Centralización de logs:** Utiliza **Graylog** para la gestión de los registros.
- Monitorización:** Utiliza **Turbine** para la recolección y agregación de métricas.
- Bases de datos:** Se utilizan diferentes tipos de bases de datos como **H2**, **MySQL** y **MongoDB**.
- Servidor de Autorización:** Utiliza **Spring Cloud Security** para gestionar el acceso a los recursos.

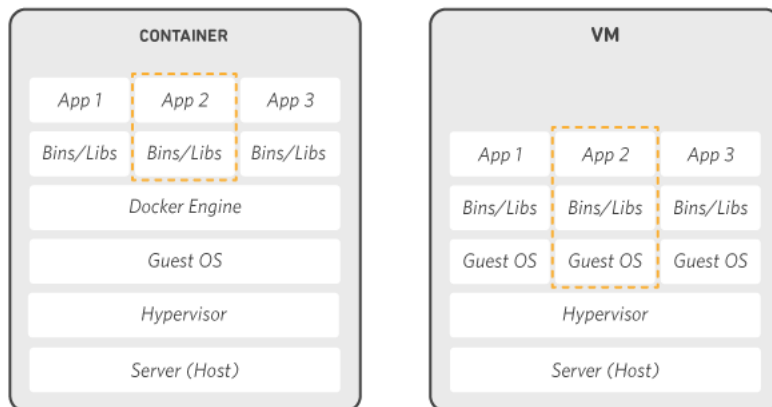
- El modelo de despliegue hace referencia al modo en que vamos a organizar y gestionar los despliegues de los microservicios, así como a las tecnologías que podemos usar para tal fin.
- El despliegue de los microservicios es una parte primordial de esta arquitectura. Muchas de las ventajas que aportan,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

# Modelo de despliegue

- Las imágenes 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

JavaScript Object Notation

<http://tools.ietf.org/html/rfc4627>

## JSON

© JMA 2016. All rights reserved

# Introducción

- JSON (JavaScript Object Notation) es un formato sencillo para el intercambio de información.
- El formato JSON permite representar estructuras de datos (arrays) y objetos (arrays asociativos) en forma de texto.
- La notación de objetos mediante JSON es una de las características principales de JavaScript y es un mecanismo definido en los fundamentos básicos del lenguaje.
- En los últimos años, JSON se ha convertido en una alternativa al formato XML, ya que es más fácil de leer y escribir, además de ser mucho más conciso.
- No obstante, XML es superior técnicamente porque es un lenguaje de marcado, mientras que JSON es simplemente un formato para intercambiar datos.
- La especificación completa del JSON es la RFC 4627, su tipo MIME oficial es application/json y la extensión recomendada es .json.

© JMA 2016. All rights reserved

# Estructuras

- JSON está constituido por dos estructuras:
  - Una colección de pares de nombre/valor. En los lenguajes esto es conocido como un objeto, registro, estructura, diccionario, tabla hash, lista de claves o un arreglo asociativo.
  - Una lista ordenada de valores. En la mayoría de los lenguajes, esto se implementa como tablas, arreglos, vectores, listas o secuencias.
- Estas son estructuras universales; virtualmente todos los lenguajes de programación las soportan de una forma u otra. Es razonable que un formato de intercambio de datos que es independiente del lenguaje de programación se base en estas estructuras.

© JMA 2016. All rights reserved

# Sintaxis

- Un array es un conjunto de valores separados por comas (,) que se encierran entre corchetes [ ... ]
- Un objeto es un conjunto de pares nombre:valor separados por comas (,) que se acotan entre llaves { ... }
- Los nombres son cadenas, entre comillas dobles (").
- El separador entre el nombre y el valor son los dos puntos (:)
- El valor debe ser un objeto, un array, un número, una cadena o uno de los tres nombres literales siguientes (en minúsculas):
  - true, false o null
- Se codifica en Unicode, la codificación predeterminada es UTF-8.

© JMA 2016. All rights reserved

# Valores numéricos

- La representación de números es similar a la utilizada en la mayoría de los lenguajes de programación.
- Un número contiene una parte entera que puede ser prefijada con un signo menos opcional, que puede ser seguida por una parte fraccionaria y / o una parte exponencial.
- La parte fraccionaria comienza con un punto (como separador decimal) seguido de uno o más dígitos.
- La parte exponencial comienza con la letra E en mayúsculas o minúsculas, lo que puede ser seguido por un signo más o menos, y son seguidas por uno o más dígitos.
- Los formatos octales y hexadecimales no están permitidos. Los ceros iniciales no están permitidos.
- No se permiten valores numéricos que no se puedan representar como secuencias de dígitos (como infinito y NaN).

© JMA 2016. All rights reserved



# Valores cadena

- La representación de las cadenas es similar a las convenciones utilizadas en la familia C de lenguajes de programación.
- Una cadena comienza y termina con comillas (").
- Se pueden utilizar todos los caracteres Unicode dentro de las comillas con excepción de los caracteres que se deben escapar: los caracteres de control (U + 0000 a U + 001F) y los caracteres con significado.
- Cuando un carácter se encuentra fuera del plano multilingüe básico (U + 0000 a U + FFFF), puede ser representado por su correspondiente valor hexadecimal. Las letras hexadecimales A-F puede ir en mayúsculas o en minúsculas.
- Secuencias de escape:
  - `\\, \/, \", \n, \r, \b, \f, \t`
  - `\u[0-9A-Fa-f]{4}`

© JMA 2016. All rights reserved

# Objeto con anidamientos

```
{
 "Image": {
 "Width": 800,
 "Height": 600,
 "Title": "View from 15th Floor",
 "Thumbnail": {
 "Url": "/image/481989943",
 "Height": 125,
 "Width": "100"
 },
 "IDs": [116, 943, 234, 38793]
 }
}
```

© JMA 2016. All rights reserved

# Array de objetos

```
[
 {
 "precision": "zip",
 "Latitude": 37.7668,
 "Longitude": -122.3959,
 "City": "SAN FRANCISCO",
 "State": "CA",
 "Zip": "94107"
 },
 {
 "precision": "zip",
 "Latitude": 37.371991,
 "Longitude": -122.026020,
 "City": "SUNNYVALE",
 "State": "CA",
 "Zip": "94085"
 }
]
```

© JMA 2016. All rights reserved

# JSON en JavaScript

- El Standard Built-in ECMAScript Objects define que todo interprete de JavaScript debe contar con un objeto JSON como miembro del objeto Global.
- El objeto debe contener, al menos, los siguientes miembros:
  - **JSON.parse** (Función): Convierte una cadena de la notación de objetos de JavaScript (JSON) en un objeto de JavaScript.
  - **JSON.stringify** (Función): Convierte un valor de JavaScript en una cadena de la notación de objetos JavaScript (JSON).

© JMA 2016. All rights reserved