**Introduction To Web Services. What is a Web Service?**

Una aplicación web retorna datos al interactuar con ella en forma de HTML, que no es el mejor formato para consumir desde otra aplicación: esta aplicación tiene las capas DATA, BUSINESS y WEB. ¿Podemos reutilizar las capas DATA y BUSINESS si creamos un JAR solo con esas capas? ¿Pero entonces una aplicación en Visual Basic va a poder hacer uso de esas capas si están en Java? Pero si se crea un servicio, y la capa de presentación hace uso de él, ya dará igual con qué está hecho el cliente.

Web services están pensados para la interoperabilidad entre aplicaciones a través de una red (misma máquina, misma red o a través de internet), independientemente de sus tecnologías.

Important How Questions related to Web Services

Para la interoperabilidad es imprescindible que el medio de comunicación sea agnóstico de las tecnologías empleadas tanto en el servicio como en los clientes: protocolo http con mensajes XML (SOAP), o info. transportada en JSON para REST.

Es necesario que la aplicación que consume el servicio sepa el formato de la información que recibe, para poder procesarlo adecuadamente. Esto se resuelve con el concepto de Definición de Servicio. Sobre todo, tiene sentido en SOAP, donde tiene que estar muy definido tanto el tipado de datos como las operaciones.

1. Request and response format.
2. Request Structure
3. Response Structure
4. Endpoint

Web Services - Key Terminology

Terminología: request es input, response es output. Message Exchange Format es XML o JSON. Service provider y service consumer: la aplicación consume el servicio web. Service definition: contrato entre el consumidor y el servicio: endpoint, estructura de la petición, estructura de la respuesta y formato de los datos.

Transporte: ¿http sobre internet, o está el servicio comunicado a través de una cola MQ? Cliente y servicio se pueden comunicar a través de una cola en ambos sentidos. De esa forma ambos son publisher y consumer. Entre el cliente y la cola MQ (por ejemplo, WebSphere MQ) está la capa SOAP. Lo mismo entre el service provider y la cola.

**Introduction to SOAP Web Services**

SOAP y REST no son comparables uno contra uno. SOAP tiene mensajes XML con envoltorio, cabecera y cuerpo. En ocasiones la cabecera es opcional.

Format: XML request y response. Transport: sobre MQ y sobre HTTP. Service definition WSDL (web service defition language): tiene todas las operaciones disponibles y los datos.

REST: hace el mejor uso de HTTP, replica su funcionalidad para pedir recursos de más naturalezas que simplemente html. Servicios que hacen uso de los conceptos ya presentes en http. Recurso-URI, representations: XML, HTML, JSON... Un recurso puede tener distintas representaciones. Se piensa en términos de recursos. Transport es SOLO http, no hay un standard para la definición del servicio (WADL, Swagger). No hay restricciones en el formato del intercambio de información (aunque JSON es muy popular).

Section 3: Introduction to Spring Framework in 10 Steps 16. Step 4 : Using Spring to Manage Dependencies - **@Component**, **@Autowired**

Spring: ¿qué beans maneja?@**Component**. ¿Qué dependencias tienen? @**Autowired** ¿Dónde buscar los beans?@**ComponentScan**. La anotación @**SprinbBootApplication** ya hace el component scan auto. Los beans que se declaran pasan a estar disponibles en el ApplicationContext.

Desde la aplicación (clase con @**SpringBootApplication**) podemos obtener el contexto fácilmente porque lo devuelve el método run que se invoca en public static main(). Y con el contexto podemos obtener los beans manejados por el contexto.

Se puede pedir un bean especificando una clase concreta, pero eso no será lo habitual, debido a que se busca un **bajo acoplamiento**. Se trabaja con **Interfaces** habitualmente.

Dynamic auto wiring and Troubleshooting - **@Primary**

Cuando hay más de una implementación de la misma interfaz, o beans definidos en una clase de configuración que son del mismo tipo, clase, etc. o bien se les asigna un nombre y luego se invocan con su @**Qualifier**, o también se les puede asignar un determinado perfil (@**Profile**), de modo que si en la configuración del proyecto (application.properties o yml) se habilita un determinado perfil, el fw podrá resolver de qué dependencia se trata. En otro caso, al no saber qué clase instanciar, el fw retornará un error de **multiple beans**. También se puede poner la anotacion @**Primary** en uno de los beans, y entonces será ese el que por defecto se utilice.

Constructor and Setter Injection

El autowired puede ser por constructor o por setter. Cuando es por setter ni siquiera es necesario declarar el método set, sino que con tener el atributo en la clase anotado con Autowired es suficiente.

Antiguamente lo que se hacía es que todas las dependencias obligatorias se inyectaban por constructor, y las opcionales por setter. Ahora no hay diferencia, **si el contexto no es capaz de resolver un bean, el contexto no arranca.** No hay mucha diferencia, por no decir que no hay ninguna a nivel de fw.

Spring Modules

**OXM**: **object to xml**, spring tiene ese módulo, entiendo que puede ser bueno para servicios web SOAP. Transacciones: buenas herramientas para controlar la transaccionalidad de los métodos que constan de varias acciones atómicas.

Section 4: Introduction to Spring Boot in 10 Steps

Introduction to Spring Boot - Goals and Important Features

web applicacion: spring-mvc, spring-core, validation fw, logging fw, y configurarlo todo: dispatcher servlet, view resolvers, etc. Con starter projects es muy fácil, con los starters tienes casi todo esto incluido y preconfigurado. Con spring data, tienes ya configurada la conexión a bd, etc. Otra cosa, embedded servers. Con springboot se arranca en contexto en un tomcat embebido. Spring boot actuator para la monitorización, es otra de las características que vienen gratis con spring boot.

Spring también facilita mucho la configuración de distintos entornos. Dev, prod, etc. Creando properties files y siguiendo un convenio de nombres, y ya :)

Developing Spring Applications before Spring Boot

Antes de springboot había un montón de decisiones que tomar a la hora de configurar el proyecto. Por ejemplo, para un proyecto web necesitaríamos spring-mvc, un fw de validación, uno de logging, seguridad… etc. Además, había que elegir las versiones de cada uno también, teniendo en cuenta posibles problemas de compatibilidad. Después, un **dispatcher servlet** (que hace las veces de controlador front) y un **view resolver**, que en base a un nombre que se le pase sabe encontrar qué vista es la que se tiene que emplear en la respuesta. Spring boot ayuda a que muchas de estas configuraciones sean hechas de forma automática.

Antes había una faceta más que configurar (que con springboot es automática), la gestión de excepciones (**exception handling**).

En **applicationContext.xml** había que configurar beans para i18n, para viewresolver... y en **web.xml** el filtro de seguridad, para que todas las peticiones fueran examinadas por él.

What is Spring Boot Auto Configuration? Springboot: component scan + autoconfiguration.

Una de las dependencias de un proyecto spring boot es spring autoconfiguration. Ese modulo tiene autoconfiguraciones para muchas librerías, como por ejemplo servlet, validation, transaction, thymeleaf, session...etc.

En el startup, al tener autoconfiguration, se recorren las clases del classpath a ver sobre cuáles se pueden aplicar autoconfiguraciones.

Spring Boot Starter Projects - Starter Web and Starter JPA

**spring starter web** facilita la creación de aplicaciones web y de servicios REST.

Dependencias: tomcat, json, hibernate validation, spring web y spring web mvc, logging fw. Todos los starters de springboot heredan de springboot starter: tienen spring boot fw, spring-auto-confgure,spring-boot-starter-logging, spring core y yaml.

Spring web: spring mvc + REST: autoconfigura el dispatcher servlet, etc. Starter test te habilita a crear test unitarios con **junit y mockito**.

Si quieres ORM hay que añadir springboot starter data jpa: hibernate, hibernate common, spring common, spring orm, transaction-api.

Overview of different Spring Boot Starter Projects

**spring boot starter web services** para **SOAP**. **spring starter web** para **Web** y **RESTful** application. starter test: unit testing y integration testing. Spring jdbc para traditional JDBC. Para añadir HATEOAS (facilitar la navegación mediante enlaces dentro de un web service REST) a un proyecto con RESTful, existe un starter también: starter-hateoas. Starter security: basic, oauth... authorization + authentication. Data JPA para hibernate etc. Cache: starter cache. Starter data rest: expone servicios sencillos REST de los recursos.

Más sofisticados pueden ser los starters para distintos embedded servers y monitoring con actuator: se pueden ver los beans creados, las llamadas a servicios, las llamadas a servicios que han fallado, etc.

Spring boot actuator lo que hace es exponer un conjunto grande de servicios REST que se pueden consultar. HAL standard: hal browser (otra dependencia que hay que poner)

<dependency>

<groupId>org.springframework.data</groupId>

<artifactId>spring-data-rest-hal-browser</artifactId>

</dependency>

Spring Boot Actuator

**localhost:8080/actuator** mediante el hal browser vemos enlaces de interés.

Para exponer el mayor número de endpoints rest informativos de actuator, en application.properties se incluye:

**management.endpoints.web.exposure.include=\***

Accediendo tras el reinicio a localhost:8080/actuator vemos un número mayor de endpoints informativos de la salud de la aplicación en ejecución.

Si vamos a localhost:8080 se abre el hal browser y es una forma un poco más visual de navegar por todos los enlaces expuestos por actuator: beans, health, security, conditions (dependencias autoconfiguradas por estar en el classpath). Loggers, metrics: memoria

En metrics se va a mostrar una lista de métricas como la memoria usada, y muchas más medidas interesantes. Concatenando el nombre de la métrica a la url podemos acceder a los valores de las métricas. Ejemplo: **jvm.memory.used**:

http://localhost:8080/actuator/metrics/jvm.memory.max

y el resultado:

{ "name": "jvm.memory.max", "description": "The maximum amount of memory in bytes that can be used for memory management", "baseUnit": "bytes", "measurements": [{ "statistic": "VALUE", "value": 5565317119 }], "availableTags": [{ "tag": "area", "values": ["heap", "nonheap"] }, { "tag": "id", "values": ["Compressed Class Space", "PS Survivor Space", "PS Old Gen", "Metaspace", "PS Eden Space", "Code Cache"] }] }

Otra característica interesante es **httptrace**: se ven todas las peticiones que ha recibido el servidor, y los resultados. Mappings también es interesante porque muestra todos los mappings a urls del proyecto.

**The Actuator HTTP Trace and Auditing features are not enabled by default anymore, since the default repositories implementations are in-memory and may consume too many resources and are not cluster friendly. For robust, production-grade HTTP tracing we recommend the use of Spring Cloud Sleuth or similar.**

Spring Boot Developer Tools

Dev tools reinicia el contexto de spring cuando detecta un cambio, de modo que no hay que parar y arrancar manualmente. Es más rápido que el reincio completo de la aplicación, pero no sirve cuando hay un cambio en el conjunto de dependencias, por ejemplo.

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

</dependency>

**SOAP Web Services with Spring and Spring Boot**

Initialize a Spring Web Services application with Spring Boot

Para servicios web soap se debe incluir el **starter-web-services**: este módulo tiene en su pom dependencias con **starter web, con spring-oxm y con spring-core**.

Contract first approach: Service definition: xml, xsd (**xml schema definition**).

El servicio acepta request xml y devuelve response xml conforme a los schema de las definition.

Dentro del body va el auténtico contenido de la petición. En una petición al servicio, en el body, puede ir algo como

<ns2:GetCourseDetailsRequest xmlns:"http://in28min...">

<ns2:id>Course1</ns2:id>

</ns2:GetCourseDetailsRequest>

y en la respuesta

<ns2:GetCourseDetailsResponse xmlns:"http://in28min...">

<ns2:id>Course1</ns2:id>

<ns2:name>Course Fantastic</ns2:name>

<ns2:author>Daniel</ns2:author>

</ns2:GetCourseDetailsRequest>

Overview of creating SOAP Web Service using Contract First Approach

Cuando tenemos claro cómo han de ser la petición y la respuesta, es momento de crear el XSD: qué estructuras tienen la petición y la respuesta. Después se genera el WSDL (lo genera el fw).

Para que el servicio maneje la petición, JAXB la convierte a objeto Java, y para que el resultado de la ejecución del servicio Java, que está compuesto por objetos, viaje de vuelta como respuesta del servicio web, JAXB lo convierte en XML de vuelta.

JAXB plugin XML binding: a partir de los XSD genera objetos.

Define XML Schema Definition (XSD) for Response – GetCourseDetailsResp

<?xml version="1.0" encoding="UTF-8"?>

<schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="http://carregal.com/courses"

    xmlns:tns="http://carregal.com/courses" elementFormDefault="qualified">

    <element name="GetCourseDetailsRequest">

        <complexType>

            <sequence>

                <element name="id" type="integer"></element>

            </sequence>

        </complexType>

    </element>

</schema>

En el código vemos cómo se define targetNamespace, que se ha de usar después en el atributo xsi:schemaLocation="namespace nombrexsd.xsd" **del xml del mensaje** (ya sea de petición o de respuesta, el que corresponda).

* complexType se usa para definir un 'objeto' con atributos.
* sequence define el conjunto ordenado de atributos.
* element contiene cada uno de los atributos del objeto xml (distinguir el uso de la palabra atributo, que en este caso no se refiere a un atributo de tag xml, si no como sinónimo de 'campo' del objeto).
* schema es el tag que envuelve toda la definición.

More about XML Schema Definition and Implementing XSD Best Practices

En los xsd se pueden definir objetos reutilizables en más puntos del esquema, como vemos en el ejemplo de la Respuesta:

<element name="GetCourseDetailsResponse">

    <complexType>

        <sequence>

            <element name="CourseDetails" type="tns:CourseDetails"></element>

        </sequence>

    </complexType>

</element>

<complexType name="CourseDetails">

    <sequence>

        <element type="integer" name="id"></element>

        <element type="string" name="name"></element>

        <element type="string" name="description"></element>

    </sequence>

</complexType>

BuenaPractica1: A la hora de crear un xsd y definir un namespace, en la etiqueta raíz tenemos

<schema xmlns="http://www.w3.org/2001/XMLSchema">

Esto no es una buena práctica, pues define un xmlns global. Es mejor poner un prefijo para los tags, de modo que sean así: <xs:element></xs:element>

Una vez definido el prefijo, se usa tanto en los tags como en los tipos de datos. Ejemplo: <xs:element name="id" type="xs:integer">

La manera de conseguir declarar el namespace para poder usarlo es:

<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>

Introduction to Java API for XML Binding (JAXB) and Configuring JAXB

Para que el servicio convierta el cuerpo de la petición en un objeto Java que pueda manejar, y para que la repuesta obtenida por la ejecución pueda ser transformada de nuevo en xml para poder viajar de vuelta en la http response, hacemos uso de JAXB. Pues JAXB necesita el XSD para hacer las transformaciones.

Hasta ahora, para que los documentos xml response y request reconocieran el xsd, los teníamos ubicados en la misma carpeta física. Colocamos el xsd en src/main.resources que está en el classpath.

Existe un plugin que se llama jaxb maven plugin, que nos puede ayudar a crear los pojos que se asemejen a los objetos definidos en el xsd. Hay que definirlo en el pom jaxb2 maven

<!-- JAXB2 Maven Plugin --><!-- XSD source folder --><!-- Java Object Source Folder --><!-- clear folder: false -->

<plugin>

    <groupId>org.codehaus.mojo</groupId>

    <artifactId>jaxb2-maven-plugin</artifactId>

    <version>1.6</version>

    <executions>

        <execution>

            <id>xjc</id>

            <goals>

                <goal>xjc</goal>

            </goals>

        </execution>

    </executions>

    <configuration>        <!-- The package of your generated sources -->

        <outputDirectory>${project.basedir}/src/main/java</outputDirectory>

        <packageName>com.example.myschema</packageName>

        <schemaDirectory>${project.basedir}/src/main/resources</schemaDirectory>

        <clearOutputDir>false</clearOutputDir>

    </configuration>

</plugin>

Configuring an Endpoint for GetCourseDetailsRequest

Con el plugin jaxb2-maven, en configuración se puede definir a qué paquete se extraen/generan los java extraídos del xsd. Si no hay un atributo package definido en la configuración, se crea un paquete cuyo nombre concuerda con el targetNamespace definido en el xsd.

<configuration>    <!-- The package of your generated sources -->

    <outputDirectory>${project.basedir}/src/main/java</outputDirectory>

    <packageName>com.carregal.jaxb</packageName>

    <schemaDirectory>${project.basedir}/src/main/resources</schemaDirectory>

    <clearOutputDir>false</clearOutputDir>

</configuration>

Aparentemente, el namespace declarado como targetNamespace juega un papel en la url del servicio, así como el nombre del objeto que trae la petición (parámetro de entrada del endpoint). Tenemos declarado:

http://carregal.com/courses GetCourseDetailsRequest

Entonces, el método de la clase que define el endpoint y que se invoca para obtener la respuesta del servicio va anotado con:

**@PayloadRoot(namespace="http://carregal.com/courses", localPart="GetCourseDetailsRequest")**

El método recibe un parámetro de tipo GetCourseDetailsRequest, que va anotado con **@RequestPayload**. A su vez, la cabecera va anotada como **@ResponsePayload**.

Spring Web Services Configuration - Generating WSDL

El XSD define los formatos en el contexto de la aplicación servidora del servicio, pero el servicio como todo necesita **exponer un documento WSDL** para que el cliente lo pueda consultar y conocer así el contrato con el servicio.

Hay entonces que montar una clase de **@Configuration** que va a declarar unos cuantos **@Bean** de interés.

**@Configuration @EnableWs**

Estas dos anotaciones en una clase de configuración habilitan la aplicación como servicio web. Los beans son:

**ServletRegistrationBean**: es necesario configurarlo porque a partir de él se detectan los **@Endpoint** por el fw. Se configura a partir de una instancia de **MessageDispatcherServlet** que recibe el contexto de la aplicación, y del **path** para nuestro servicio.

MessageDispatcherServlet mds = new MessageDispatcherServlet();

mds.setApplicationContext(context);

mds.setTransformWsdlLocations(true);

return new ServletRegistrationBean(mds, "/ws/\*");

Lo próximo que necesitamos es exponer nuestro **WSDL**. No lo creamos a mano, si no que **Spring** lo crea y expone por nosotros. Lo crea **a partir de nuestro xsd**.

Hay que definir un bean **DefaultWsdl11Definition**, y para eso es necesario inyectarle un **XsdSchema** que lo creamos en otro Bean que devuelve una instancia de **SimpleXsdSchema**. Para crear la instancia hace falta el **classpath resource** que apunte a nuestro xsd. Entonces, para el DefaultWsdl11Definition seteamos setPortType:

   @Bean(name = "courses")

    public DefaultWsdl11Definition defaultWsdlDefinition(XsdSchema schema) {

        definition.setPortTypeName("CoursePort");

        definition.setSchema(schema);

        definition.setTargetNamespace("http://carregal.com/courses");

        definition.setLocationUri("/ws");

        return definition;

    }

    @Bean

    XsdSchema getCoursesSchema() {

        return new SimpleXsdSchema(new ClassPathResource("course-details.xsd"));

    }

Hace falta la dependencia **wsdl4j** para poder crear automáticamente el wsdl:

<dependency>

    <groupId>wsdl4j</groupId>

    <artifactId>wsdl4j</artifactId>

</dependency>

48. Step 14 - Quick introduction to different parts of a WSDL

**Resumiendo** lo visto en capítulos anteriores: se ha creado una clase anotada con **@Endpoint** que contiene los métodos que procesan las distintas peticiones que se pueden hacer al servicio web.  Cada petición distinta tiene un método anotado con **@PayloadRoot**, anotación que tiene los atributos **namespace** y **localPart**. El localPart es el que tiene su entrada en el wsdl como **operation**. Estos métodos también están anotados con **@ResponsePayload**. El parámetro de entrada, con el que se les informa, está anotado con **@RequestPayload**.

Es típico no tener lógica de conversión o de manejo de los datos en la capa de control (en su definición de endpoints), por lo que se hace uso de capas de negocio como los Service para obtener la información, y de mapeadores para transformar objetos de negocio en objetos de presentación. En nuestro código de proyecto de pruebas hemos inyectado al **@Endpoint** un **@Service** y un **@Mapper**. Se definió el pojo de negocio y ese es el que devuelve el **@Service**. El **@Endpoint** devuelve objetos **JAXB** adecuados al XSD, transformados desde objeto de negocio a objeto de presentación mediante el **@Mapper** (MapStruct).

**wsdl**: types, messages, portType, binding y service. Estructura común.

**Types**: tipos presentes en el esquema. El **xsd**. Estructuras.

**Messages**: request y responses. Diferentes requests y responses de las operations. Solo se pueden enviar el requests o recibir en responses los tipos de Types que están en messages.

En **portType** es donde están definidas las operaciones. De momento en este servicio están definidas dos operaciones. En cada operación se mapea cual es la petición y cuál es la respuesta (input message - output message). PortType es como una Interface, muestra los métodos disponibles.

**Binding** es como la implementación. Se define, por ejemplo, **transport** (si es http, o MQ, etc). Se define el **style=document** para indicar que estamos intercambiando peticiones y respuestas completas **xml**. Otra alternativa sería **RPC**, entonces necesitaríamos definir los parámetros de entrada y salida. "Como exponemos las operaciones".

**Service**: location address.

50. Step 16 - Improving the DeleteCourseDetailsRequest - Using an Enum for Status

@PayloadRoot(namespace = "http://carregal.com/courses", localPart = "DeleteCourseDetailsRequest")

@ResponsePayload

public DeleteCourseDetailsResponse deleteCourseDetailsResquest(@RequestPayload DeleteCourseDetailsRequest request) {

    DeleteCourseDetailsResponse response = new DeleteCourseDetailsResponse();

    response.setResult(statusMapper.statusToOpStatus(service.deleteById(request.getId())));

    return response;

}

-----

package com.carregal.SpringSoap.beans.mappers;

import org.mapstruct.Mapper;

import com.carregal.SpringSoap.beans.Status;

import com.carregal.SpringSoap.jaxb.OpStatus;

@Mapper(componentModel = "spring")

public interface StatusMapper {

    public OpStatus statusToOpStatus(Status status);

}

51. Step 17 - Exception Handling and SOAP Fault Responses

Control de excepciones: se debe devolver un mensaje indicativo cuando el servicio falla. **Fault Response**. En el body de la respuesta se debe enviar un fault message en caso de un error. Si se produce una excepción no capturada en el **@PayloadRoot**, el servicio responde, pero no del todo correctamente:

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

    <SOAP-ENV:Header />

    <SOAP-ENV:Body>

        <SOAP-ENV:Fault>

            <faultcode>SOAP-ENV:Server</faultcode>

            <faultstring xml:lang="en">java.lang.NullPointerException</faultstring>

        </SOAP-ENV:Fault>

    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>

Realmente esto ya es algo, y es que spring se encarga de por lo menos producir una respuesta con un mensaje soap. Pero esto se puede mejorar.

**Una buena práctica** es crear una **Exception** que extienda de **RuntimeException**, y disponer del constructor con mensaje para el error. De esta forma, cuando se produzca la situación anterior podemos elevar la excepción y la respuesta del servicio será más informativa. Para que la respuesta no sugiera que se trata de un error en el servidor (por ejemplo cuando se han pasado parámetros de entrada no válidos), podemos anotar la clase de la excepcion custom como **@SoapFault** con distintos atributos: el primero es **faultCode=FaultCode.CLIENT** para indicar que se trata de un fallo de la aplicación consumidora. Otra alternativa es usar el atributo anterior pero con valor **faultCode=FaultCode.CUSTOM**, y entonces declarar además el atributo **customFaultCode="{aqui un namespace, una url}codigo\*random\*mio"**

Anotando la excepción con **@SoapFault** y con el atributo **faultCode=FaultCode.CUSTOM**, **customFaultCode="{namespace}randomErrorCode"** tenemos el siguiente resultado:

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

    <SOAP-ENV:Header />

    <SOAP-ENV:Body>

        <SOAP-ENV:Fault>

            <faultcode xmlns:ns0="http://in28minutes.com/courses">ns0:ERR\_001\_NOT\_FOUND</faultcode>

            <faultstring xml:lang="en">Identificador inválido:23</faultstring>

        </SOAP-ENV:Fault>

    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>

52. Step 18 - Implementing Security for SOAP Web Services with WS Security

Implementando seguridad en servicio SOAP con **WS-security**. Transport independent. Authentication, digital signatures, certificates. **XWSS** es una implementación de WS-Security. Vamos a instaurar una *política* de seguridad con user y password.

Dependencias: spring-ws-security, pero con exclussions: spring-security-core:

<dependency>

    <groupId>org.springframework.ws</groupId>

    <artifactId>spring-ws-security</artifactId>

    <exclusions>

        <exclusion>

            <groupId>org.springframework.security</groupId>

            <artifactId>spring-security-core</artifactId>

        </exclusion>

    </exclusions>

</dependency>

Se incluye también la dependencia xws security y javax activation, con la exclusión de javax.xml.crypto

Una vez con las dependencias, hay que añadir cambios a la **@Configuration**. Hay que declarar un **@Bean** **XwsSecurityInterceptor** y después añadirlo a la lista de interceptores de Spring. El interceptor necesita un callback handler, la lógica que se ejecuta al interceptar una petición, chequear si el usuario es válido.

Para poder añadir el interceptor a la lista de interceptores de Spring hay que establecer que la clase de configuración extienda de **WsConfigurerAdapter**:

@Configuration

@EnableWs

public class WebServiceConfig extends WsConfigurerAdapter {

@Override public void addInterceptors(List<EndpointInterceptor> interceptors) {

    interceptors.add(securityInterceptor());

    super.addInterceptors(interceptors);

}

Cuando se crea el interceptor, hay que definir un Callback Handler, y para el interceptor de seguridad vamos a definirlo como un **SimplePasswordValidationCallbackHandler**. A este handler hay que pasarle un map de usuarios para que pueda comprobar la validez.

Otra cosa necesaria es definir un xml llamado **securityPoliciy.xml**. Este se setea como **interceptor.setPolicyConfiguration(new ClassPathResource("securityPolicy.xml"));**

El xml define la necesidad de enviar usuario y password, pero también de que no se hacen operaciones de seguridad adicionales más complejas (en nuestro caso).

Si tratamos en enviar una petición como hasta ahora, sin usuario ni password, el servicio responde con un error:

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">

    <SOAP-ENV:Header />

    <SOAP-ENV:Body>

        <SOAP-ENV:Fault>

            <faultcode>SOAP-ENV:Client</faultcode>

            <faultstring xml:lang="en">com.sun.xml.wss.XWSSecurityException: Message does not conform to configured policy [ AuthenticationTokenPolicy(S) ]: No Security Header found; nested exception is com.sun.xml.wss.XWSSecurityException: com.sun.xml.wss.XWSSecurityException: Message does not conform to configured policy [ AuthenticationTokenPolicy(S) ]: No Security Header found</faultstring>

        </SOAP-ENV:Fault>

    </SOAP-ENV:Body>

</SOAP-ENV:Envelope>

El siguiente fragmento muestra las cabeceras de seguridad:

<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/">

    <Header>

        <Security xmlns="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">

            <UsernameToken>

                <Username>danielcro</Username>

                <Password Type="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-username-token-profile-1.0#PasswordText">dani01</Password>

            </UsernameToken>

        </Security>

    </Header>

    <Body>

        <GetCourseDetailsRequest xmlns="http://carregal.com/courses">

            <id>3</id>

        </GetCourseDetailsRequest>

    </Body>

</Envelope>

**Section 6: RESTful Web Services with Spring and Spring Boot**

Initializing a RESTful Services Project with Spring Boot: proyecto con **spring-starter-web** (mvc+rest), **devtools**, **h2**, y **jpa**

Un recurso tiene URI, y representaciones. Al mapear recursos a URIs hay que tener en cuenta la precedencia y la pertenencia.

**Pej:** un usuario crea un post: **POST** users/{id}/post

Temas que vamos a tratar: validation, i18n, exception handling, versioning, content negociation, documentation, monitoring

Lo básico: Se define una clase como controladora de tráfico entrante de peticiones mediante la anotación **@RestController**, y en ella habrá distintos métodos para las peticiones entrantes. Esto se consigue mediante anotaciones **@GetMapping**, **@PostMapping**, **@RequestMapping** y otras.

Al usar **@RequestMapping** se deben usar atributos de la anotación: **method= RequestMethod.GET, path="/hello-world"**

Si la clase estuviera marcada con la anotación **@Controller**, el return sería de un string con el nombre de la vista que se devolvería al navegador. En caso de **@RestController** lo que se devuelve con return puede ser cualquier objeto, y se devuelve de forma por defecto en una cadena json.

path se usa en RequestMapping si se incluye más de un atributo. **RequestMapping** es genérico, pero necesita que se especifique el método

HTTP 57. Step 04 - Enhancing the Hello World Service to return a Bean

Poner **ResponseBody** en el método del controlador no es necesario porque **no** es un **@Controller**, sino un **@RestController**, por tanto, no es necesario.

Quick Review of Spring Boot Auto Configuration and Dispatcher Servlet

Estableciendo el nivel de logging para spring a debug, lo que quiere decir que se escribe todo a partir de info para arriba

**DispactcherServlet** es el **front controller** que sabe qué **@Controller** procesa la petición.

boot autoconfig también incluye los message converters, responsables de la conversión a json del objeto retornado por el servicio. Jackson beans detrás de los message converters.

El **dispatcher servlet** maneja todas las peticiones, se ve mapeado a "/"

La autoconfiguración de spring boot **pone un dispatcher servlet escuchando en la ruta / por defecto**. Se sigue el patrón llamado **front controller**. El dispatcher servlet conoce todos los mappings de la aplicación.  
  
Para admitir parámetros en la propia URI, el método que la gestiona tiene que declarar parámetros anotados con **@PathVariable o @RequestParam**. Esta anotación tiene **atributos como name, o required.**

En el **arranque** se leen todos los **mappings** de los **controller**, de modo que el **front controller** sabe qué bean (controller) gestiona cierta URL. **@RestController** ya incluye **@ResponseBody**, de ahí que no sea necesaria.

61. Step 08 - Implementing GET Methods for User Resource

**spring.jackson.date-format=YYYY-MM-dd (hay otros mecanismos como DataBinder)**

**spring.jackson.serialization.write-dates-as-timestamps=false**

@InitBinder

// ver DOC:inicializa org.springframework.web.bind.WebDataBinder

**public** **void** initBinder(WebDataBinder binder) {

SimpleDateFormat dateFormat = **new** SimpleDateFormat("dd/MM/yyyy");

binder.registerCustomEditor(Date.**class**, **new** CustomDateEditor(dateFormat, **false**));

}

En **application.properties** podemos configurar como se escribe el json al serializar, y también como se lee al deserializar si fuese necesario.

62. Step 09 - Implementing POST Method to create User Resource

En las peticiones POST al servicio REST, que se utilizan para registrar nuevos usuarios en nuestro caso particular, la información para la creación del usuario se recibe en el cuerpo de la petición. Se emplea como parámetro del método una anotación que habilita el mapeo del body de la petición a un objeto de la clase que esperamos recibir para poder realizar la acción.

**@PostMapping(path="/users") public String createUser(@RequestBody User user){ ... }**

Peticiones POST: idealmente queremos ver la respuesta **HTTP 201 Created** en vez de la 200 OK, porque es más específica de la acción que queremos realizar. También queremos que el servicio devuelva la URI del recurso recién creado.

@PostMapping(path = "/users")

**public** String createNewUser(@RequestBody User user) {

User savedUser = usersDao.save(user);

**return** "localhost:8080/users/" + savedUser.getId();}

Esta primera aproximación NO envía una respuesta con status adecuado y además la URI devuelta está generada "a mano". Veremos esto también, cómo hacerlo bien.

63. Step 10 - Enhancing POST Method to return correct HTTP Status Code and Location

Vemos ahora como devolver el estado http CREATED, y la URI del recurso. En el framw hay una class **ResponseEntity** que modela la respuesta del servicio REST. Se compone sobre un genérico, y permite asociar el estado http en función del resultado de la ejecución.

**ResponseEntity<T>.created(location);**

Para conseguir la URI, buscamos la clase **ServletUriComponentsBuilder** y el método **fromCurrentRequest()** para obtener la URI de una petición POST, necesaria para devolver ResponseEntity<Object>

URI location=ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}").buildAndExpand(savedUser.getId()).toUri();

En la response del servicio, en las cabeceras, tenemos el atributo location que contiene la URI del recurso creado, y date con la fecha y hora de creación del recurso. Asimismo, el status de la respuesta **HTTP es 201 Created**.

64. Step 11 - Implementing Exception Handling - 404 Resource Not Found

Para elegir el tipo de status http que se devuelve con un determinado error (controlado por una excepción al efecto), se utiliza la anotación **@ResponseStatus(value = HttpStatus.NOT\_FOUND)**

@ResponseStatus(value = HttpStatus.***NOT\_FOUND***)

**public** **class** UserNotFoundException **extends** RuntimeException {

**public** UserNotFoundException(String message) {

**super**(message);

}

}

El servicio REST con SpringBoot tiene un *default exception handling* que es el que retorna cosas como el mensaje, path, error, etc. Es buena práctica tener una estructura que homogeneice los mensajes de error para todo: definir una estructura estandarizada y que la sigan TODOS tus RESTful services.

**Validación + estructura estándar para gestionar**

65. Step 12 - Implementing Generic Exception Handling for all Resources

Para estandarizar la gestión de excepciones se puede definir un pojo que contenta los campos que se quieren devolver con la información relativa al error: mensaje, timestamp, details... Esto debería ir no solo a nivel de proyecto, sino de organización. La clase del pojo puede llamarse **ExceptionResponse**.

**public** **class** ExceptionResponse {

**private** String message;

**private** Date timestamp;

**private** String details;

}

Tenemos otra clase del framework spring, en este caso relacionada con la respuesta en caso de un error: **ResponseEntityExceptionHandler**. Es una clase abstracta. La extendemos y así podemos tener una exception handling customizada. El comportamiento definido por la implementación de esta clase se puede aplicar a todos los **controller** de la aplicación.

Esta clase tiene que tener las anotaciones **@Controller** (o **@RestController**) y **@ControllerAdvice**, que es una anotación que tienen que tener todas las clases que declaren métodos con anotación @**ExceptionHandler**.

@ControllerAdvice

@RestController

public class ResponseEntityExceptionHandlerImpl extends ResponseEntityExceptionHandler {

    @ExceptionHandler(Exception.class)

    @Nullable

    public final ResponseEntity<Object> handleAllException(Exception ex, WebRequest request) throws Exception {

        ExceptionResponse exceptionResponse = new xceptionResponse(ex.getMessage(), new Date(),

                request.getDescription(false));

        return new ResponseEntity(exceptionResponse, HttpStatus.INTERNAL\_SERVER\_ERROR);

    }

@ExceptionHandler({ NoUserPostsException.class, UserNotFoundException.class, NoExistingUserPostException.class })

@Nullable

public final ResponseEntity<Object> handleUserNotFoundException(Exception ex,WebRequest request) throws Exception {

        ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(), new Date(),

                request.getDescription(false));

        return new ResponseEntity(exceptionResponse, HttpStatus.NOT\_FOUND);

    }

    @Override

    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

            HttpHeaders headers, HttpStatus status, WebRequest request) {

        ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(), new Date(),

                ex.getBindingResult().toString());

        return new ResponseEntity(exceptionResponse, HttpStatus.BAD\_REQUEST);

    }

}

Al definir una clase que extiende **ResponseEntityExceptionHandler**, la excepción que teníamos declarada a la que le poníamos una anotación **@ResponseStatus(value = x)**ya no determina el status de la respuesta, sino que lo hace nuestra implementación de **ResponseEntityExceptionHandler**en cada método anotado con **@ExceptionHandler(Exception.class)**. Hay un método de estos sobrescrito para cada subconjunto de excepciones que queramos manejar. En esos métodos se crea la **ResponseEntity**, que lleva el **status** http que se asigne en cada caso.

Creando un método con **@ExceptionHandler(UserNotFoundException.class)**, que es la exception que elevamos al no encontrar el user, podemos asignar el **httpStatus.NOT\_FOUND**a la **ResponseEntity**, como esperaríamos.

Una buena práctica puede ser crear un **jar** con las **respuestas** que queremos dar a las **excepciones** **comunes**.

66. Step 13 - Exercise: User Post Resource and Exception Handling

Ejercicio 1: después de haber dado una implementación para findAllUsers, findUserById y para createUser, qué mejoras podemos añadir para el manejo de excepciones? Hacer lo propio para los posts, por ejemplo, indicar que el usuario no existe al querer crear un post a un userId.

Ejercicio 2: crear métodos para los casos de uso de posts de usuarios de la red social (temática del servicio REST ejemplo para este curso):

* GET /users/{id}/posts
* POST /users/{id}/posts
* GET /users/{id}/posts/{post\_id}

67. Step 14 - Implementing DELETE Method to delete a User Resource

Añadir el recurso para borrar un usuario es sencillo. Se debe crear el método anotado con **@DeleteMapping("users/{id}")**

Una respuesta 200OK es perfectamente válida, pero también se puede devolver un **ResponseEntity** con No content().

**return ResponseEntity.noContent().build().**

Como los métodos del **dao** elevan excepciones manejadas por el framework al haber definido un ResponseEntityExceptionHandler, el manejo de excepciones controla bien el caso de querer borrar un recurso que no existe.

69. Step 15 - Implementing Validations for RESTful Services

Validación. Existe una API de Java sobre validación**. Javax validation API**: @Valid anotacion.  Este API viene provista por **starter web**.

La anotación **@Valid** se utiliza en el **@RequestBody**, o sea, en el objeto que llega al servicio para ser procesado.

Se añaden más anotaciones en lo atrib de la clase a la que se mapea el requestbody:

**@Size(min = 2) private String name**;

**@Past private Date birthDate**;

Con este mecanismo tan sencillo, si no se cumple alguna de las restricciones, la respuesta es **400 Bad request**. Esto no es completo, pues no aporta un mensaje con el error concreto.

Para poder sobrescribir el comportamiento en caso de error de validación, tenemos que ir a nuestra clase **ResponseEntityExceptionHandlerImpl** y hacer **@Override** del método:

    @Override

    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,

            HttpHeaders headers, HttpStatus status, WebRequest request) {

        ExceptionResponse exceptionResponse = new ExceptionResponse(ex.getMessage(), new Date(),

                request.getDescription(false));

        return new ResponseEntity(exceptionResponse, HttpStatus.BAD\_REQUEST);

    }

Ahora bien, en la casuística de fallo de validación, los detalles del error, **details** en nuestro pojo ExceptionResponse, se saca de la **excepcion**, de su atributo **BindingResult()**

En el objeto **BindingResult()** está disponible toda la **colección de errores**, de modo que se puede iterar por ellos.

En el paquete **javax.validation API** hay muchas más validaciones:

**AssertFalse, AssertTrue, DecimalMax, DecimalMin, Digits, Future, Max, Min, NotNull, Null, Pattern (regex)...\*.**

Implementación: **hibernate validation.**

Resumen: en los pojos de recursos, en sus atributos, se hace uso de **las anotaciones de validación**. En el **controller**, se anotan los parámetros de los métodos con **@Valid**, para habilitar la validación.

71. Step 16 - Implementing HATEOAS for RESTful Services

**HATEOAS: hypermedia state**. Hay un starter para eso: **starter-hateoas**.

Este starter permite **añadir enlaces** a otros recursos haciendo uso del nombre de los métodos, de modo que, si se cambiase el **path** en aquel método, también se **refleja** en el enlace **provisto por HATEOAS**.

La idea es crear un recurso, que se construye sobre un genérico, en el caso de nuestro ejemplo será un usuario, y **devolver el recurso** **en vez del pojo User**. Sobre el recurso **podemos añadir enlaces** haciendo uso de los nombres de métodos que ofrecen los otros recursos. Si hacemos **import static** de una clase, podemos tener **acceso** a todos sus métodos estáticos sin referenciar el nombre de la clase.

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

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

Resource<User> resource = new Resource<User>(user);

ControllerLinkBuilder linkTo = linkTo(methodOn(this.getClass()).retrieveAllUsers());

resource.add(linkTo.withRel("all-users"));

{ "id": 1,

"name": "Daniel",

"birthDate": "2020-04-04 11:23:07",

**"\_links": { "all-users": {"href": "http://localhost:8080/users"} }**

}

73. Step 18 - Internationalization for RESTful Services

i18n: Vamos a habilitar un mecanismo por el que cuando **recibamos una cabecera http "Accept-language"** devolvamos los textos traducidos.

Configuración: **LocaleResolver**, **ResourceBundleMessageSource**

Uso: **@Autowired** **MessageSource**

**@RequestHeader(value="Accept-Language", required=false) String locale** será uno de los **parámetros** del método de controlador, o recurso. Entonces se recupera el texto traducido como:

**String mensajeHola = messageSource.getMessage("helloWorld.message", null, locale);**

**org.springframework.web.servlet.i18n.SessionLocaleResolver:** org.springframework.web.servlet.LocaleResolver implementation that uses a locale attribute in the user's session in case of a custom setting, with a fallback to the specified default locale or the request's accept-header locale.

This is most appropriate if the application needs user sessions anyway, i.e. when the HttpSession does not have to be created just for storing the user's locale. The session may optionally contain an associated time zone attribute as well; alternatively, you may specify a default time zone.

Custom controllers can override the user's locale and time zone by calling #setLocale(Context) on the resolver, e.g. responding to a locale change request. As a more convenient alternative, consider using org.springframework.web.servlet.support.RequestContext.changeLocale.

Se configura un **@Bean** **ResourceBundleMessageSource** y se establece el *basename* como "messages", ya que se han definido los archivos de properties messages.properties y messages\_fr.properties.

    @Bean

    ResourceBundleMessageSource messageSource() {

        ResourceBundleMessageSource messageSource = new ResourceBundleMessageSource();

        messageSource.setBasename("messages");

        return messageSource;

    }

**org.springframework.context.MessageSource:**

Strategy interface for resolving messages, with support for the parameterization and internationalization of such messages.

Se define un **@Bean** que devuelve un **LocaleResolver** implementado por **SessionLocaleResolver()**, y un **@Bean** que devuelve un **MessageSource** implementado por **ResourceBundleMessageSource**.

El primero lo declaramos para que resuelva un Locale por defecto: **Locale.US.**

El segundo es el que se inyecta en el **controller** o donde sea preciso, para tener acceso a las properties traducidas.

74. Step 18 - Part 2 - Internationalization

Una simplificación es obtener el Locale desde la clase **LocalContextHolder.getLocale()**. De esta manera no es necesario recoger el locale de las cabeceras de las peticiones **para cada método** del controller.

@GetMapping(path = "/hello-locale")

public HelloBean getHelloLocale(@RequestHeader(name = "Accept-Language", required = false) Locale locale) {

   return new HelloBean(messageSource.getMessage("hello.message", null, locale));

}

@GetMapping(path = "/hello-locale-2")

public HelloBean getHelloLocale2() {

   return new HelloBean(messageSource.getMessage("hello.message", null, **LocaleContextHolder.getLocale()**));

}

Otra simplificación puede ser cambiar el **SessionLocaleResolver** por un **AcceptHeaderLocaleResolver**, de ese modo **según si se recibe un locale en las cabeceras** y existe el lenguaje en un properties, se hace uso del lenguaje elegido.

La tercera simplificación es incluir en **application.properties** la línea:

**spring.messages.basename = messages**

De este modo **no es necesario** configurar un **ResourceBundleMessageSource** (implementación de **MessageSource**). Se puede seguir haciendo **@Autowire** de una instancia de **MessageSource** en el controlador para poder invocar **getMessage(args),** pero la implementación la pone el framework sin que hayamos definido nosotros el bean**.<-- solución más sencilla.**

75. Step 19 - Content Negotiation - Implementing Support for XML

Para poder permitir representaciones además de **json**, por ejemplo **xml**, tenemos que tener disponibles las librerías de dependencias necesarias en el proyecto.

<!-- https://mvnrepository.com/artifact/com.fasterxml.jackson.dataformat/jackson-dataformat-xml -->

<dependency>

    <groupId>com.fasterxml.jackson.dataformat</groupId>

    <artifactId>jackson-dataformat-xml</artifactId>

    <version>2.10.3</version>

</dependency>

<item>

    <List>

        <id>1</id>

        <name>Daniel</name>

        <birthDate>2020-04-05 00:18:57</birthDate>

    </item>

    <item>

        <id>2</id>

        <name>Veronica</name>

        <birthDate>2020-04-05 00:18:57</birthDate>

    </item>

</List>

76. Step 20 - Configuring Auto Generation of Swagger Documentation

**Swagger es para REST lo que Wizdler para SOAP**, una forma de presentar el contrato del servicio web al cliente. Debemos añadir la dependencia al pom. Hay que crear también una clase **@Configuration** para configurar Swagger.

La clase tiene que tener las anotaciones **@Configuration** y **@EnableSwagger**

Hay que definir un **@Bean Docket**, y se devuelve **new Docket(DocumentationType.SWAGGER2)**

En la URL: **localhost:8080/swagger-ui.html** tenemos la visualización de los endpoints expuestos por nuestro servicio REST, como esperábamos.

En la URL: **localhost:8080/api-docs** tenemos el servicio documentado en formato json.

79. Step 22 - Enhancing Swagger Documentation with Custom Annotations

En la información que exponemos en Swagger, podemos configurar ciertos datos para que aparezcan: por ejemplo, podemos poner información sobre las **restricciones** que impongamos en la **validación**. También podemos poner una persona de contacto en la sección Info. Empezando por Info:

En la clase con la configuración de swagger, y vemos el Bean Docket que tenemos definido. Al crear el Docket haciendo uso del método apiInfo podemos **pasar un bean con nuestra nueva Info de Api**. Así de sencillo:

**new Docket(DocumentationType.SWAGGER\_2).apiInfo(DEFAULT\_API\_INFO);**

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.Configuration;

import springfox.documentation.service.ApiInfo;

import springfox.documentation.service.Contact;

import springfox.documentation.spi.DocumentationType;

import springfox.documentation.spring.web.plugins.Docket;

import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration

@EnableSwagger2

public class SwaggerConfiguration {

    public static final Contact DEFAULT\_CONTACT = new Contact("Daniel Carregal", "https://github.com/danicarregal",

            "dani\_carregal@hotmail.com");

    public static final ApiInfo DEFAULT\_API\_INFO = new ApiInfo("Api Title In28minutes", "Api REST Docu", "1.luv",

            "url términos/servicios", DEFAULT\_CONTACT, "Apache 2.luv", "http://www.apache.org/licenses/LICENSE-2.0");

    private static final Set<String> DEFAULT\_API\_PRODUCES = new HashSet(

            Arrays.asList("application/json", "application/xml"));

    @Bean

    public Docket api() {

        return new Docket(DocumentationType.SWAGGER\_2).apiInfo(DEFAULT\_API\_INFO).produces(DEFAULT\_API\_PRODUCES)

                .consumes(DEFAULT\_API\_PRODUCES);

    }

}

Podemos copiar el objeto apiInfo de la clase Docket y modificarlo.

http://localhost:8080/v2/api-docs

También en la info general mostrada por Swagger, debemos actualizar los formatos consumidos y producidos por el servicio.

En el Docket, en apiInfo, hacemos uso de los métodos **consumes** y **produces** para pasarle dos HashSet de Strings con los formatos. Se puede observar en localhost:8080/v2/api-docs el resultado:

"consumes": [ "application/xml", "application/json" ], "produces": [ "application/xml", "application/json" ],

El siguiente objetivo es mostrar las restricciones de validación en la documentación del api. Hay que emplear la anotación **@ApiModel(description="")** en la cabecera de las clases 'de dominio', y la anotación **@ApiModelProperty(notes = "La fecha debe ser del pasado para ser válida")**  en los atributos con restricciones declaradas por otras anotaciones de **javax.validation**.

Hay muchas anotaciones de Swagger que pueden resultar de utilidad a la hora de documentar nuestro api. Si hacemos control click en la anotación **ApiModelProperty**, esto nos lleva a un paquete dentro del jar llamado Swagger anotations.

80. Step 23 - Monitoring APIs with Spring Boot Actuator

Monitorización de servicios con actuator. Se debe incluir la dependencia en pom. También la de hal browser. Las url para acceder al panel de control de actuator han ido cambiando con el tiempo: **localhost:8080/actuator**, o **localhost:8080/application**.

Para configurar que actuator muestre mucha información hay que establecer una property:

**management.endpoints.web.exposure.include =\*;**

Si vamos **a http://localhost:8080/** se accede al hal browser, que muestra la información en un formato web. Tiene una barra de navegación en la que podemos navegar a **/actuator**, y ver gráficamente **métricas** expuestas por actuator. Cosas interesantes que se pueden ver:

beans manejados, mappings de urls, usuarios logueados y otros que han fallado la validación, métricas como la memoria empleada, uso de cpu...

**httptrace** muestra las peticiones recibidas, con sus respuestas. Esto ya no está activo porque tiene gran impacto y no es seguro. Ahora se usan cosas como **Zipkin/Grafana**, etc

81. Step 24 - Implementing Static Filtering for RESTful Service

Otra faceta lógica de los servicios REST es la del **filtrado**. Un recurso puede contener más atributos de los que se quiere transmitir, de modo que se han de habilitar mecanismos para elegir qué información se comparte. Por otro lado, dependiendo del tipo de petición, podría considerarse un conjunto distinto de campos en cada caso. En el segundo caso estaríamos hablando de **filtrado dinámico**. El filtrado estático, en el cual siempre se envía el mismo conjunto de datos de un recurso, se acomete fácilmente mediante **@JsonIgnore** en los campos del recurso que NO queremos enviar en la respuesta.

Alternativamente se puede usar la anotación **@JsonIgnoreProperties(value={"at1","at2"})** con una lista de campos **hardcodeados**. Esto es peor si cambiar el nombre de los atributos y no se cambia el hardcodeo de las propiedades.

82. Step 25 - Implementing Dynamic Filtering for RESTful Service

El filtrado estático se configura **sobre las clases DTO** o de dominio, mediante anotaciones del estilo de **@JsonIgnore**. Se indica con ello qué atributos **no** formarán parte del json devuelto por el servicio.

El **filtrado dinámico**, por el cual podemos decidir qué campos devolver en función de la URL solicitada (del método del controller que gestione la petición http), o de otros criterios, se realiza mediante la clase:

**MappingJacksonValue**, que crea un **wrapper** sobre el DTO sobre el que se pueden establecer filtros. Ahora lo vemos:

Patrón DTO y mappers:

Este es el fragmento de pom para incluir la dependencia a mapstruct, una librería para proveer mappers entre entities y DTOs.

<dependency>

    <groupId>org.mapstruct</groupId>

    <artifactId>mapstruct-jdk8</artifactId>

    <version>1.3.0.Beta2</version>

</dependency>

MappingJacksonValue es el **tipo de objeto que devolvemos** si tenemos **filtrado dinámico**. Los pasos son:

1.-Crear el filtro con alguno de los métodos estáticos de **SimpleBeanPropertyFilter** --> filter

2.-Crear un objeto **FilterProvider** que contendrá todos los filtros, y **añadir el filtro creado**.

3.-Crear el objeto **MappingJacksonValue** alrededor del pojo, y setear los filtros contenidos en el FilterProvider.

MappingJacksonValue mapping = new MappingJacksonValue(resource);

SimpleBeanPropertyFilter filter = SimpleBeanPropertyFilter.filterOutAllExcept("id", "name");

FilterProvider filters = new SimpleFilterProvider().addFilter("userFilter\*ID\*NAME", filter);

mapping.setFilters(filters);

El método de controller retorna el **MappingJacksonValue** en vez del DTO. Y, también **necesario**, es:

4.-En el DTO o pojo con la información, a nivel de **clase** hay que usar la anotación **@JsonFilter("nombre del filtro")**. Ese nombre del filtro es el mismo empleado en la llamada al método para añadir el filtro en el provider.

**@JsonFilter(**"userFilter\*ID\*NAME"**)**

**public class User { …**

83. Step 26 - Versioning RESTful Services - Basic Approach with URIs

Versionado, soluciones empleadas por distintas grandes empresas:

* **GITHUB** usa el tipo de *media* (content negotiation or accept-header)
* Custom Headers (**Microsoft**)
* Versiones en URI (**Twitter**)
* Versiones en parámetros (**Amazon**)

Factores a tener en cuenta:

Polución de URIs, mal uso de las cabeceras HTTP, Caching, documentación... no hay solución favorita.

// ------------ simple way of versioning---------

@GetMapping("person/v1")

public **Personv1** getPersonUrl() {

return new Personv1(new Name("Daniel", "Carregal"));

}

@GetMapping("person/v2")

public **Personv2** getPersonUrl2() {

 return new Personv2("Daniel", "Carregal", "M");

}

Para tener varias versiones del servicio, una de las formas fáciles que vienen a la mente es mapeando diferentes URIs según la versión. Esta es la forma básica de versionado:

**GET v1/person**

**GET v2/person**

Y suponiendo que tenemos DTOs distintos por versión, haciendo la prueba se reciben distintas versiones del recurso.

84. Step 27 - Versioning RESTful Services - Header and Content Negotiation Approach

// ------------ headers way of versioning---------

@GetMapping(value = "person/header", headers = "X\_API\_VERSION**=1**")

public Personv1 getPersonHeader() {

    return new Personv1(new Name("Daniel", "Carregal"));

}

@GetMapping(value = "person/header", headers = "X\_API\_VERSION**=2**")

public Personv2 getPersonHeader2() {

    return new Personv2("Daniel", "Carregal", "M");

}

// ------------ produces way of versioning---------

@GetMapping(value = "person/produces", produces = "application/vnd.company.app-**v1**+json")

public Personv1 getPersonProduces() {

    return new Personv1(new Name("Daniel", "Carregal"));

}

@GetMapping(value = "person/produces", produces = "application/vnd.company.app-**v2**+json")

public Personv2 getPersonProduces2() {

    return new Personv2("Daniel", "Carregal", "M");

}

El de produces es como el versionado usando el mime type, o media. Lleva en @GetMapping el atributo **produces** y el valor es el tipo de dato que acepta. Es una forma un poco artificiosa de indicar la versión del servicio.

Con parámetros es muy directa, pero se puede producir una **cierta polución de urls**. Pasa lo mismo con tener varios mapeos de URL según la versión del recurso que se desea consumir.

Con cabeceras como **X\_API\_VERSION** la petición no se puede hacer desde un navegador, además de que las cabeceras nunca se pensaron para pasar información como la versión del servicio que se desea invocar.

Consideraciones también se pueden hacer sobre los métodos de versionado que usan cabeceras, pues al tener **la misma URL** (pero distintas cabeceras) se **inhabilita** el **cacheo** de peticiones, pues la última **sobrescribiría** a la anterior aun en el caso de que llevasen cabeceras **diferentes**.

85. Step 28 - Implementing Basic Authentication with Spring Security

Para añadir seguridad básica al API debemos incluir un starter más: **spring-starter-security**.

Al arrancar la aplicación, en el log del Tomcat (Console en Eclipse) se puede ver:

Using generated security password: **f33599d8-6da4-4421-8fc5-c3222d7e2e6d**

Esta es una pasword default que se ha de usar con el ***username*** default, que es: **user**. Estos valores se pueden sobrescribir en **applicacion.properties:**

spring.security.user.name=daniel

spring.security.user.password=unapasswordcool

Una vez **habilitado** el starter **security**, se ha de usar la **auth básica** para las peticiones. Esto es: usuario y password. En otro caso, la respuesta es **401 unauthorized** en algunas versiones, o una página con un formulario para loguearse, en otras versiones de springboot.

Section 7: Introduction to JPA in 10 Steps 86. Introduction to JPA - An Overview

Introducimos conceptos básicos de la OOP para notar que el diseño de datos en almacenamientos persistentes como las bases de datos relacionales tiene de base una naturaleza distinta a la forma en que se maneja la información en un lenguaje de programación orientado a objetos.

Existe pues, una **discordancia** entre ambos planos: nombres de atributos y de columnas diferentes, manera de materializar las relaciones, la herencia, y más cosas.

Sistemas de conexión entre relacional y objetos: legacy en caso de JDBC, spring JDBC e Ibatis (Mybatis). Estos 3 están basados en **queries** y **resultsets**, pero con diferencias.

87. Step 1 : Object Relational Impedence Mismatch

Conceptos de OOP:

**Abstraction.** Abstraction means using simple things to represent complexity. We all know how to turn the TV on, but we don’t need to know how it works in order to enjoy it. In Java, abstraction means simple things like **objects**, **classes**, and **variables** represent more complex underlying code and data. This is important because it lets avoid repeating the same work multiple times.

**Encapsulation.**This is the practice of keeping fields within a class private, then providing access to them via public methods. It’s a protective barrier that keeps the data and code safe within the class itself. This way, we can re-use objects like code components or variables without allowing open access to the data system-wide.

**Inheritance.**This is a special feature of Object Oriented Programming in Java. It lets programmers create new classes that share some of the attributes of existing classes. This lets us build on previous work without reinventing the wheel.

**Polymorphism.**This Java OOP concept lets programmers use the same word to mean different things in different contexts. One form of polymorphism in Java is **method overloading**. That’s when different meanings are implied by the code itself. The other form is **method overriding**. That’s when the different meanings are implied by the values of the supplied variables.

88. Step 2: World before JPA - JDBC, Spring JDBC and myBatis

JDBC, Spring JDBC y myBatis están basados en queries.

**JDBC**: conceptos como **Statement**, **PreparedStatement**, **ResultSet**.

El **PreparedStatement** es el patrón para inyectar los parámetros en las queries. Hay que escribir código para transformar la información devuelta en forma de **ResultSet** hacia objetos Java. Las queries tienen parámetros.

Con JDBC hay que configurar el **datasource** y obtenerlo, después establecer una **conexión**.

A partir de la conexión crear la **PreparedStatement** pasando como parámetro la **query** con comodines. Sobre la PreparedStatement ir asignando cada uno de los valores de las variables de entrada. Después **execute** de la PreparedStatement. Hacer **close** sobre la **PreparedStatement** y después **close** sobre la **Connection**.

Y todo esto ni siquiera tiene en cuenta el control de excepciones.

**SpringJDBC** provee, sobre lo anterior, **JdbcTemplate** para facilitar las cosas. Es mucho más fácil pasar los parámetros y mucho más fácil mapear el resultado a un objeto Java.

Se necesitan muchas menos líneas de código para la misma operación. Existen los **BeanPropertyRowMapper**. Con ellos, si los nombres concuerdan, **no es necesario crear un mapper** específico para las filas del resultSet. Si no, puede ser necesario crear el Mapper que implemente la Interface **RowMapper<T>**:

class TodoMapper implements RowMapper<Todo>{ @Override public Todo mapRow(ResultSet rs, int rowNum) throws SQLException{ Todo todo=new Todo(); todo.setId(rs.getInt("id")); .... return todo; } }

Con **SpringJDBC**, haciendo uso de **JdbcTemplate**, se accede a métodos para ejecutar queries de forma sencilla y con poco código:

return jdbcTemplate.queryForObject("SELECT \* FROM TODO where id=?", id, new TodoMapper());

**SpringJDBC**provee una capa sobre JDBC, haciendo uso de conceptos como la **JDBCTemplate**.

**MyBatis**elimina la necesidad de código para asignar los parámetros y para obtener los resultados. Ofrece conf. por **xml o anotaciones sobre los pojos** hacia la base de datos.

Las queries tienen **parámetros** que coinciden con los **atributos** de las **clases manejadas**. Como si se crease una **clase** para **contener los parámetros de cierta query o conjunto que queries** con los mismos parámetros.

89. Step 3 : Introduction to JPA

Estas tres maneras de acceder a la información de la base de datos (JDBC, SpringJDBC y MyBatis) hacen uso de queries. Es difícil mantener un proyecto si hay que rehacer cosas y leer y entender todas las queries. **JPA**trata de solventar ese problema, el de la mantenibilidad, el de la dependencia con la base de datos concreta, etc.

JPA permite **mapear clases a tablas**, haciendo uso de **Entities**, **relaciones**, el **entity manager** para guardar cosas en la base de datos. También ofrece el **API Criteria** y **JPQL** que tienen funcionalidad adicional.

El jpql se puede usar en sustitución del SQL para hacer consultas. Es algo parecido, pero más orientado a los **objetos** entity Java que a las columnas de una tabla de bd.

Anotaciones de JPA de las más comunes:

@Entity

@Table(name="Task")

public class Task{

@Id

@GeneratedValue

private int id;

@Column(name="description")

private String desc; ... }

Se pueden modelar las reaciones entre entities mediante anotaciones como:

**@OneToMany, @ManyToMany, @ManyToOne @OneToOne**

ejemplo:

public class Employee{ ...

@ManyToMany

private List<Task> tasks;

... }

Otro uso interesante, herencia de clases Java que mapean a una única tabla de base de datos:

Ejemplo: distintos tipos de empleado (fijo y temporal)

@Entity

@Inheritance(strategy = Inheritance.SINGLE\_TABLE)

@DiscriminatorColumn(name = "EMPLOYEE\_TYPE")

public class Employee { // Other employee atributes

}

public class FullTimeEmployee extends Employee{

    protected Integer salary;

}

public class PartTimeEmployee extends Employee{

    protected Float hourlyWage;

}

La columna **employee\_type** de la base de datos, determina del tipo de empleado del que se trata.

92. Step 5 : Defining a JPA Entity - User

Creación de project para experimentar. Initializr con dependencias: H2, JPA, WEB.

Comparación entre Hibernate y JPA. En breve, se puede asumir que JPA es la interfaz e Hibernate es una implementación. JPA define todas las anotaciones. Hibernate es un framework de ORM. JPA define la especificación, es una API: **entities**, **attributes**, **relaciones**, manejo de las **entities**. Hibernate entiende los mapeos entre objetos y tablas, y garantiza que los datos se leen o guardan en la BD en base a los mapeos. Hibernate ofrece más cosas de las que especifica JPA.

Para el ejemplo creamos una entity User. JPA necesita que las entities tengan constructor por defecto sin parámetros. Para convertir un pojo en una entity: **@Entity**

93. Step 6 : Defining a Service to manage the Entity - UserService and EntityManager

JPA define las anotaciones @**Repository**y @**Transactional**. La primera se usa para dar a ver al contexto de Spring que la clase es un DAO y que ha de estar disponible para inyectar (componente manejado).

La segunda indica que los métodos del repositorio están envueltos en transacción, de modo que, si la ejecución se rompe en alguno de ellos, las operaciones que se hubieran realizado son anuladas.

**EntityManager**: cuando un pojo se persiste con el EM, empieza a ser **gestionado** por el **PersistenceContext**, con lo que está vinculado a la base de datos mediante su Id, por ejemplo. Solo aquellos cambios que están en el PersistenceContext son trackeados por el EM.

El campo **EntityManager** en el DAO se inyecta por DI. Hay que darle la anotación @**PersistenceContext**.

94. Step 7 : Using a Command Line Runner to save the User to database

Para probar la clase UserDaoService, que es tan sencilla como vemos:

@Repository

@Transactional

public class UserDAOService {

    @PersistenceContext

    private EntityManager entityManager;

    public Integer insert(User user) {

        entityManager.persist(user);

        return user.getId();

    }

}

vamos a usar un **CommandLineRunner**de SpringBoot. Este **se lanza al inicio de la aplicación**.

Nueva clase **UserDaoServiceCommandLineRunner**, la creamos nosotros. Debe **implementar**CommandLineRunner, que tiene un método: **run**

Este CommandRunner se va a ejecutar al arrancar el contexto de Spring, al ejecutar la @**SpringBootApplication**.

95. Step 8 : Magic of Spring Boot and In Memory Database H2

Para sacar trazas a la consola podemos hacer uso de

private static Logger logger = LoggerFactory.getLogger(UserDaoServiceCommandLineRunner.class);

Dando implementación al método run de nuestro CommandLineRunner, de modo que se use el DAO y se inserte un usuario, observamos que realmente el sistema nos informa de que ha persistido un usuario. Lo ha hecho en la **bd en memoria H2**.

Para poder acceder a la consola de H2 desde el navegador (**localhost:8080/h2-console**) hace falta habilitar una opción en **applicacion.properties**.

La base de datos H2 se crea en base a los starters que tenemos definidos en el proyecto y a la autoconfiguración de SpringBoot.

La base de datos in memory H2 es creada al arrancar la aplicación: se crea el schema, se crean las tablas, se pueblan las tablas etc. Y cuando la aplicación se cierra, toda la base de datos desaparece también. Es volátil porque es en memoria. Existen opciones para persistirla en disco, no obstante.

Para ver en la **consola** los comandos y **queries** SQL que se invocan contra la base de datos, otra property:

**spring.jpa.show-sql=true**

Para poder acceder a la consola H2 desde el navegador, hay que habilitar la property en applicacion.properties:

**spring.h2.console.enabled=true**

Una vez se tiene acceso, la url de la bd es **jdbc:h2:mem:testdb** y **username: sa**

Entramos en la consola, conectamos con la base de datos y vemos que, aunque no establecimos un **@Table(name="users")** en la entity, por **defecto** ya se establece con el mismo nombre de la entity pero en mayúscula.

Podemos ver algo de información sobre la bd creada:

**SELECT \* FROM INFORMATION\_SCHEMA.TABLES**

La columna TABLE\_TYPE = **TABLE** (por contraposición a **SYSTEM\_TABLE**)

La columna SQL del registro

**CREATE MEMORY TABLE "PUBLIC"."USER"( "ID" INTEGER NOT NULL, "NAME" VARCHAR(255), "ROLE" VARCHAR(255) )**

Todo lo relacionado con la base de datos en memoria, creación de la base de datos, el schema, las tablas, usuarios de acceso, la conexión, viene dado o autogenerado por **spring boot autoconfiguration**.

Podemos ver en la consola más información de todos los pasos ejecutados habilitando más nivel de detalle en el log:

**logging.level.org.springframework=debug**

En el log ahora se puede ver toda la configuración automática que se ha llevado a cabo en función de los starters y las dependencias del proyecto.

Por ejemplo, la configuración de la base de datos H2. Al tener tambien **Data-JPA** configura **Hibernate** (**HibernateEntityManager**). También el **TransacionManager**.

SpringBoot, en base a todos los jar que encuentra en el classpath, realiza la configuración más adecuada.

Un uso de las bases de datos in memory es **UnitTesting**, **mocking**, etc.

96. Step 9 : Introduction to Spring Data JPA

A raíz de haber escrito el código para un DAO service de una entity concreta, User, y en el momento de crear otro servicio para otra entity diferente, se comprueba que el **código sería prácticamente igual** salvo por la entity sobre la que aplica.

Este tipo de situación es la que hace pensar en el uso de **Java Generics**. **Spring-Data-JPA** lo tiene cubierto.

Si pensamos en un DAO para operaciones CRUD, todos los DAO de la aplicación serán parecidos.

Spring Data te habilita a **escribir solo las interfaces**, y provee las implementaciones **dado el genérico** indicado en la Interface.

Es más, las operaciones CRUD más habituales que vienen a la mente, **no hace falta ni declararlas**, ya vienen OOTB.

Hay que crear, en este caso, una Interface que extienda de **JpaRepository<User, Integer>**.

Están disponibles gran cantidad de métodos para obtener registros, para crear, actualizar, borrar...

En el caso de la inserción que probamos antes desde un CommandLineRunner, ahora sería como sigue:

**User savedUser = userRepo.save(user);**

El método **save** vale para **crear** y para **actualizar**. Hay una tabla en la documentación de Spring que explica un **convenio de nombres** para operaciones de SELECT: por nombres de columnas, por rango de fechas... Los nombres de métodos que siguen el convenio son del estilo **FindFirstByFechaAfter(Date date)** .

**JpaRepository** ofrece multitud de métodos para obtener información de la base de datos en forma de entities mapeadas con las tablas.

Por ejemplo:

**Optional<User> optuser = userRepository.findById(1);**

Es un *optional* porque podría no haber ningún resultado. Mediante el método get() del optional se accede a la **entity**, pero podría no existir, lo que derivaría en un error si tratamos de acceder a alguno de sus atributos.

Caused by: java.util.NoSuchElementException: No value present at java.util.Optional.get(Unknown Source) ~[na:1.8.0\_121]

...

97. Step 10 : More JPA Repository : findById and findAll

User userRetrieved = userRepo.findById(23).orElse(null);

logger.info(userRetrieved != null ? "Se ha encontrado un usuario" : "No existe el usuario");

List<User> users = userRepo.findAll();

logger.info("Usuarios: " + users);

salida (consola eclipse):

UserDaoServiceCommandLineRunner  : No existe el usuario

UserDaoServiceCommandLineRunner  : Usuarios: [User [id=1, name=Daniel, role=admin]]

Llega el momento de cambiar el proyecto anterior, el servicio REST de usuarios y posts, para adaptarlo a Spring-Data-Jpa. Hacemos tres cosas para empezar: deshabilitar spring-security, y habilitar:

spring.jpa.show-sql=true

spring.h2.console.enabled=true

Para que application.properties nos asista a la hora de escribir configuraciones, hay que tener instalado Spring Tools en el IDE.

Section 8: Connecting RESTful Web Service to JPA - Creating User Entity and some test data

Vemos que en el arranque se crea la tabla:

Hibernate: create table user (id integer not null, birth\_date timestamp, name varchar(255), primary key (id))

Ponemos el archivo sql en **resources** dentro del proyecto: data.sql

Insertamos datos en la tabla:

insert into user values (1,sysdate(),'Daniel');

insert into user values (2,sysdate(),'Vero');

insert into user values (3,sysdate(),'Chiquitin');

insert into user values (4,sysdate(),'Rosa');

insert into user values (5,sysdate(),'Jose Luis');

Se debe recordar que en el **script** SQL **no** se pueden usar comillas dobles.

101. Step 31 - Updating GET methods on User Resource to use JPA

La funcionalidad de filtrado, por la que definimos un **FilterProvider** que contiene un conjunto de filtros, necesita que en la entity se use la anotación **@JsonFilter**.

En caso de que se defina un **FilterProvider** en alguno de los **métodos de controller que retornan la entity** o algún objeto construido alrededor de la entity anotada, parece que es imprescindible que el **mismo** **filterProvider** se defina en **todos** los **endpoints** que retornen objetos relacionados con esa misma entity (caso de findAll y findById).

La prueba realizada consiste en haber habilitado filtrado en el método **get /users/{id}** y después probar el método **get /users**. Como en el segundo no se programó el FilterProvider, pero en la **entity** se hace uso de la anotación @JsonFilter, se produce un **error 500**:

**{ "message": "...Cannot resolve PropertyFilter with id 'userFilter\*ID\*NAME' ; no FilterProvider configured "timestamp": "2020-04-09 10:28:25", "details": "uri=/users" }**

No es necesario, aunque en el video el instructor lo usa, anotar la **interface** con **@Repository**, pues Spring ya **'sabe'** que una interfaz que extiende **JpaRepository ES un @Repository.**

Otro apunte al respecto es que las anotaciones **@Repository**, **@Service**, **@Component**, etc suelen estar en implementaciones, no en interfaces.

102. Step 32 - Updating POST and DELETE methods on User Resource to use JPA

Para borrar recursos existe tanto el método **delete(User)** como el **deleteById(id)**.

A la hora de ir a hacer save de un usuario, con userRepo.save(user) a mi me da error de ConstraintViolationException por el id. Yo lo envié sin valor asignado. Al instructor también le da el mismo error. La explicación, tiene que ver, como era de esperar, con **@GeneratedValue**. En el curso no tenía más **atributos** esa anotación. Yo, personalmente, tenía:

**strategy=GenerationType.AUTO.**

Hibernate genera una **Sequence** empezando por **1** (**PersistenceContext**). Esto no tiene mucho sentido para mí, pues Hibernate debería tener en cuenta la base de datos.

Como habíamos **insertado unos datos en el arranque** mediante un script **data.sql** insertando 5 usuarios, el primero de los cuales con id=1, se ha producido la **colisión**.

Parece lógico pensar en configurar Hibernate para que la sequence empiece en un valor adecuado, en vez de en 1, **o cambiar los ids en el script**.

103. Step 33 - Creating Post Entity and Many to One Relationship with User Entity

En las relaciones siempre podemos establecer el atributo **fetch**, para que solo se obtengan las entities relacionadas en caso de que se acceda a la propiedad que contiene esa información.

Si tenemos una relación definida bilateralmente, para evitar un bucle infinito de relaciones, uno de los dos lados de la relación tiene que tener **fetch=FetchType.LAZY**.

**@ManyToOne(fetch=FetchType.LAZY)**

**private User user;**

En las relaciones @**OneToMany**, para evitar tener una columna con la relación en ambas tablas, usamos el atributo **mappedBy**que especifica qué **campo de la otra entity** contiene la relación.

**@OneToMany(mappedBy = "user")**

**List<Post> post;**

Section 9: RESTful Web Services - Best Practices - Richardson Maturity Model

Maturity Model de Richardson. Cómo de **RESTful**es tu servicio REST:

**Nivel 0:** exponer servicios SOAP al estilo REST (**http** pero con **endpoints** indicando nombres de métodos, o con el verbo http equivocado como GET http://server/deletePost)

Como que no se exponen recursos, sino más bien **actions**. Retrieve esto, delete lo otro...etc.

**Nivel 1:** Se exponen **recursos** con las URI adecuadas, pero no se hace un buen uso de los métodos http.

**Nivel 2:** Como el nivel 1 pero con los métodos HTTP adecuados.

**Nivel 3:** Nivel 2 + **HATEOAS**. Datos más siguientes posibles acciones (uris para like, unlike, comment).

107. Step 37 - RESTful Web Services - Best Practices

Buenas prácticas de servicios REST:

1. **Consumer**first: buena **documentación**(swagger), **hateoas**, buen nombrado de los **recursos**.
2. Hacer buen uso de lo que ofrece **HTTP**: métodos, códigos de respuesta correctos (201, 404, 400, etc).
3. No enviar información **sensible**en las URIs.
4. Para los nombres de recursos, usar mejor **plurales**(/users/1 en vez de /user/1)
5. Usar **nombres**para los recursos, no verbos.
6. Cuando no sea posible usar nombres, seguir siempre un criterio **uniforme**.