# Servicios Rest

María Isabel Alfonso Galipienso `<<eli@ua.es>>`

## Table of Contents

1. Introducción a REST. Diseño y creación de servicios RESTful ... 4
 1.1. ¿Qué es un servicio Web? ... 4
  Servicios Web RESTful ... 5
 1.2. Fundamentos de REST ... 5
  Recursos ... 6
  Representación de los recursos ... 7
  Direccionabilidad de los recursos: URI ... 8
  Uniformidad y restricciones de las interfaces ... 9
 1.3. Diseño de servicios Web RESTful ... 11
 1.4. Un primer servicio JAX-RS ... 12
  Modelo de objetos ... 12
  Modelado de URIs ... 13
  Definición del formato de datos ... 13
  Asignación de métodos HTTP ... 15
  Implementación del servicio: Creación del proyecto Maven ... 20
  Implementación del servicio: Recursos JAX-RS ... 23
  Construcción y despliegue del servicio ... 28
  Probando nuestro servicio ... 29
 1.5. Ejercicios ... 33
  Servicio REST ejemplo (0 puntos) ... 33
  Servicio REST saludo (1 punto) ... 33
  Servicio REST foro (1 punto) ... 35
2. Anotaciones básicas JAX-RS. El modelo de despliegue. ... 38
 2.1. ¿Cómo funciona el enlazado de métodos HTTP? ... 38
 2.2. La anotación @Path ... 39
  Expresiones @Path ... 40
  Parámetros *matrix* (*Matrix parameters*) ... 44
  Subrecursos (*Subresource Locators*) ... 44
 2.3. Usos de las anotaciones @Produces y @Consumes ... 49
  Anotación @Consumes ... 49
  Anotación @Produces ... 51
 2.4. Inyección de parámetros JAX-RS ... 51
  @javax.ws.rs.PathParam ... 52
  Interfaz UriInfo ... 53
  @javax.ws.rs.MatrixParam ... 55
  @javax.ws.rs.QueryParam ... 55
  @javax.ws.rs.FormParam ... 56
  @javax.ws.rs.HeaderParam ... 56
  @javax.ws.rs.core.Context ... 57
  @javax.ws.rs.BeanParam ... 58
  Conversión automática de tipos ... 59
  Valores por defecto (@DefaultValue) ... 60
 2.5. Configuración y despliegue de aplicaciones JAX-RS ... 60
  Configuración mediante la clase *Application* ... 60
  Configuración mediante un fichero *web.xml* ... 63

Configuración en un contenedor que no disponga de una implementación JAX-RS ..... 63
2.6. Ejercicios ..... 65
Creación de un recurso: creación y consulta de temas en el foro (0,5 puntos) ..... 65
Despliegue y pruebas del recurso (0,5 puntos) ..... 67
Múltiples consultas de los temas del foro (0,5 puntos) ..... 68
Creación de subrecursos (0,5 puntos) ..... 69
3. Manejadores de contenidos. Respuestas del servidor y manejo de excepciones. ..... 72
3.1. Proveedores de entidades ..... 72
Interfaz javax.ws.rs.ext.MessageBodyReader ..... 72
Interfaz javax.ws.rs.ext.MessageBodyWriter ..... 73
3.2. Proveedores de entidad estándar incluidos en JAX-RS ..... 73
javax.ws.rs.core.StreamingOutput ..... 74
java.io.InputStream, java.io.Reader ..... 75
java.io.File ..... 76
byte[] ..... 77
String, char[] ..... 78
MultivaluedMap<String, String> y formularios de entrada ..... 78
3.3. Múltiples representaciones de recursos ..... 79
3.4. Introducción a JAXB ..... 80
Clase JAXBContext ..... 87
Manejadores JAX-RS para JAXB ..... 88
JAXB y JSON ..... 89
3.5. Respuestas del servidor ..... 91
Códigos de respuesta por defecto ..... 91
Elaboración de respuestas con la clase Response ..... 93
3.6. Manejadores de excepciones ..... 98
La clase javax.ws.rs.WebApplicationException ..... 99
Mapeado de excepciones ..... 100
Jerarquía de excepciones ..... 101
3.7. Ejercicios ..... 104
Servicio REST ejemplo ..... 104
Plantillas que se proporcionan ..... 104
Uso de JAXB (0,5 puntos) ..... 104
Uso de manejadores de contenidos y clase Response (0,75 puntos) ..... 105
Manejo de excepciones (0,75 puntos) ..... 106
4. HATEOAS y Seguridad ..... 107
4.1. ¿Qué es HATEOAS? ..... 107
4.2. HATEOAS y Servicios Web ..... 108
Enlaces Atom ..... 108
Ventajas de utilizar HATEOAS con Servicios Web ..... 108
Enlaces en cabeceras frente a enlaces Atom ..... 111
4.3. HATEOAS y JAX-RS ..... 111
Construcción de URIs con UriBuilder ..... 112
URIs relativas mediante el uso de UriInfo ..... 115
Construcción de enlaces (Links) en documentos XML y en cabeceras HTTP ..... 116
4.4. Seguridad ..... 118
Autentificación en JAX-RS ..... 119
Autorización en JAX-RS ..... 121
Encriptación ..... 122
Seguridad programada ..... 124
4.5. Ejercicios ..... 126
Uso de Hateoas (1 puntos) ..... 126

Ejercicio seguridad (1 punto) ..... 127
5. Api cliente. Procesamiento JSON y Pruebas ..... 128
5.1. API cliente. Visión general ..... 128
Obtenemos una instancia Client ..... 129
Configuramos el *target* del cliente (URI) ..... 131
Construimos y Realizamos la petición ..... 133
Manejo de excepciones ..... 138
5.2. Procesamiento JSON ..... 140
5.3. Modelo de procesamiento basado en el modelo de objetos ..... 141
Creación de un modelos de objetos desde el código de la aplicación ..... 142
Navegando por el modelo de objetos ..... 143
Escritura de un modelo de objetos en un *stream* ..... 144
Modelo de procesamiento basado en *streaming* ..... 145
5.4. Pruebas de servicios REST ..... 147
Ciclo de vida de Maven y tests JUnit ..... 147
Anotaciones JUnit y aserciones AssertThat ..... 151
Observaciones sobre los tests y algunos ejemplos de tests ..... 154
5.5. Ejercicios ..... 158
Tests utilizando el API cliente y un mapeador de excepciones (1 punto) ..... 158
Tests utilizando el API Json y JUnit (1 punto) ..... 158
Implementación de los tests ..... 159

# 1. Introducción a REST. Diseño y creación de servicios RESTful

En esta sesión vamos a introducir los conceptos de servicio Web y servicio Web RESTful, que es el tipo de servicios con los que vamos a trabajar. Explicaremos el proceso de diseño del API de un servicio Web RESTful, y definiremos las URIs que constituirán los "puntos de entrada" de nuestra aplicación REST. Finalmente ilustraremos los pasos para implementar, desplegar y probar un servicio REST, utilizando *Maven*, *IntelliJ*, y el servidor de aplicaciones *Wildfly*. También nos familiarizaremos con *Postman*, una herramienta para poder probar de forma sencilla los servicios web directamente desde el navegador.

## 1.1. ¿Qué es un servicio Web?

El diseño del software tiende a ser cada vez más modular. Las aplicaciones están formadas por una serie de componentes reutilizables (**servicios**), que pueden encontrarse distribuidos a lo largo de una serie de máquinas conectadas en red.

El **WC3** (***W**orld **W**ide **W**eb Consortium*) define un servicio Web como un sistema software diseñado para soportar interacciones máquina a máquina a través de la red. Dicho de otro modo, los servicios Web proporcionan una forma estándar de interoperar entre aplicaciones software que se ejecutan en diferentes plataformas. Por lo tanto, su principal característica su gran **interoperabilidad** y **extensibilidad** así como por proporcionar información fácilmente procesable por las máquinas gracias al uso de XML. Los servicios Web pueden combinarse con muy bajo acoplamiento para conseguir la realización de operaciones complejas. De esta forma, las aplicaciones que proporcionan servicios simples pueden interactuar con otras para "entregar" servicios sofisticados añadidos.

### Historia de los servicios Web

Los servicios Web fueron "inventados" para solucionar el problema de la **interoperabilidad** entre las aplicaciones. Al principio de los 90, con el desarrollo de Internet/LAN/WAN, apareció el gran problema de integrar aplicaciones diferentes. Una aplicación podía haber sido desarrollada en C++ o Java, y ejecutarse bajo Unix, un PC, o un computador mainframe. No había una forma fácil de intercomunicar dichas aplicaciones. Fué el desarrollo de XML el que hizo posible compartir datos entre aplicaciones con diferentes plataformas hardware a través de la red, o incluso a través de Internet. La razón de que se llamasen servicios Web es que fueron diseñados para residir en un servidor Web, y ser llamados a través de Internet, típicamente via protocolos HTTP, o HTTPS. De esta forma se asegura que un servicio puede ser llamado por cualquier aplicación, usando cualquier lenguaje de programación, y bajo cualquier sistema operativo, siempre y cuándo, por supuesto, la conexión a Internet esté activa, y tenga un puerto abierto HTTP/HTTPS, lo cual es cierto para casi cualquier computador que disponga de acceso a Internet.

A nivel **conceptual**, un servicio es un componente software proporcionado a través de un **endpoint** accesible a través de la red. Los servicios productores y consumidores utilizan mensajes para intercambiar información de invocaciones de petición y respuesta en forma de documentos auto-contenidos que hacen muy pocas asunciones sobre las capacidades tecnológicas de cada uno de los receptores.

### ¿Qué es un *endpoint*?

Los servicios pueden interconectarse a través de la red. En una arquitectura orientada a servicios, cualquier interacción punto a punto implica dos **endpoints**: uno que proporciona un servicio, y otro de lo consume. Es decir, que un **endpoint** es cada uno de los "elementos", en nuestro caso nos referimos a servicios, que se sitúan en ambos "extremos" de la red que sirve de canal de comunicación entre ellos. Cuando hablamos de servicios Web, un *endpoint* se especifica mediante una URI.

A nivel **técnico**, los servicios pueden implementarse de varias formas. En este sentido, podemos distinguir dos tipos de servicios Web: los denominados servicios Web "grandes" (*"big" Web Services*), los llamaremos servicios Web SOAP, y servicios "ligeros" o servicios Web RESTful.

Los servicios Web SOAP se caracterizan por utilizar mensajes XML que siguen el estándar SOAP (***S**imple **O**bject **A**ccess **P**rotocol*). Además contienen una descripción de las operaciones proporcionadas por el servicio, escritas en WSDL (***W**eb **S**ervices **D**escription **L**anguage*), un lenguaje basado en XML.

Los servicios Web RESTful, por el contrario, pueden intercambiar mensajes escritos en diferentes formatos, y no requieren el publicar una descripción de las operaciones que proporcionan, por lo que necesitan una menor "infraestructura" para su implementación. Nosotros vamos a centrarnos en el uso de estos servicios.

### Servicios Web RESTful

Son un tipo de Servicios Web, que se adhieren a una serie de restricciones arquitectónicas englobadas bajo las siglas de **REST**, y que utilizan estándares Web tales como URIs, HTTP, XML, y JSON.

El API Java para servicios Web RESTful (**JAX-RS**) permite desarrollar servicios Web RESTful de forma sencilla. La versión más reciente del API es la 2.0, cuya especificación está publicada en el documento **JSR-339**, y que podemos descargar desde https://jcp.org/en/jsr/detail?id=339. A lo largo de estas sesiones, veremos cómo utilizar JAX-RS para desarrollar servicios Web RESTful. Dicho API utiliza anotaciones Java para reducir los esfuerzos de programación de los servicios.

## 1.2. Fundamentos de REST

El término REST proviene de la tesis doctoral de Roy Fielding, publicada en el año 2000, y significa ***RE**presentational **S**tate **T**ransfer* (podemos acceder a la tesis original en: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm). REST es un conjunto de restricciones que, cuando son aplicadas al diseño de un sistema, crean un estilo arquitectónico de software. Dicho estilo arquitectónico se caracteriza por seguir los siguientes principios:

- Debe ser un sistema **cliente-servidor**
- Tiene que ser **sin estado**, es decir, no hay necesidad de que los servicios guarden las sesiones de los usuarios (cada petición al servicio tiene que ser independiente de las demás)
- Debe soportar un sistema de **cachés**: la infraestructura de la red debería soportar caché en diferentes niveles

- Debe ser un sistema **uniformemente accesible** (con una interfaz uniforme): Esta restricción define cómo debe ser la interfaz entre clientes y servidores. La idea es simplificar y desacoplar la arquitectura, permitiendo que cada una de sus partes puede evolucionar de forma independiente. Una interfaz uniforme se debe caracterizar por:

    # Estar basada en **recursos**: La abstracción utilizada para representar la información y los datos en REST es el **recurso**, y cada recurso debe poder ser accedido mediante una URI (**U**niform **R**esource **I**dentifier).

    # Orientada a **representaciones**: La interacción con los servicios tiene lugar a través de las **representaciones** de los **recursos** que conforman dicho servicio. Un **recurso** referenciado por una URI puede tener diferentes formatos (*representaciones*). Diferentes plataformas requieren formatos diferentes. Por ejemplo, los navegadores necesitan HTML, JavaScript requiere JSON (**J**ava**S**cript **O**bject **N**otation), y una aplicación Java puede necesitar XML.

    # Interfaz **restringida**: Se utiliza un pequeño conjunto de métodos bien definidos para manipular los recursos.

    # Uso de mensajes **auto-descriptivos**: cada mensaje debe incluir la suficiente información como para describir cómo procesar el mensaje. Por ejemplo, se puede indicar cómo "parsear" el mensaje indicando el tipo de contenido del mismo (xml, html, texto,…)

    # Uso de Hipermedia como máquina de estados de la aplicacion (**HATEOAS**): Los propios formatos de los datos son los que "dirigen" las transiciones entre estados de la aplicación. Como veremos más adelante con más detalle, el uso de HATEOAS (**H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate), va a permitir transferir de forma explícita el estado de la aplicacion en los mensajes intercambiados, y por lo tanto, realizar interacciones con estado.

- Tiene que ser un sistema por **capas**: un cliente no puede "discernir" si está accediendo directamente al servidor, o a algún intermediario. Las "capas" intermedias van a permitir soportar la escalabilidad, así como reforzar las políticas de seguridad

A continuación analizaremos algunas de las abstracciones que constituyen un sistema RESTful: recursos, representaciones, URIs, y los tipos de peticiones HTTP que constituyen la interfaz uniforme utilizada en las transferencias cliente/servidor

## Recursos

Un recurso REST es cualquier cosa que sea direccionable (y por lo tanto, accesible) a través de la Web. Por direccionable nos referimos a recursos que puedan ser accedidos y transferidos entre clientes y servidores. Por lo tanto, un recurso es una **correspondencia lógica y temporal** con un concepto en el **dominio** del problema para el cual estamos implementando una solución.

Algunos ejemplos de recursos REST son:

- Una noticia de un periódico
- La temperatura de Alicante a las 4:00pm
- Un valor de IVA almacenado en una base de datos
- Una lista con el historial de las revisiones de código en un sistema CVS
- Un estudiante en alguna aula de alguna universidad

- El resultado de una búsqueda de un ítem particular en Google

Aun cuando el mapeado de un recurso es único, diferentes peticiones a un recurso pueden devolver la misma representación binaria almacenada en el servidor. Por ejemplo, consideremos un recurso en el contexto de un sistema de publicaciones. En este caso, una petición de la "última revisión publicada" y la petición de "la revisión número 12" en algún momento de tiempo pueden devolver la misma representación del recurso: cuando la última revisión sea efectivamente la 12. Por lo tanto, cuando la última revisión publicada se incremente a la versión 13, una petición a la última revisión devolverá la versión 13, y una petición de la revisión 12, continuará devolviendo la versión 12. En definitiva: cada uno de los recursos puede ser accedido directamente y de forma independiente, pero diferentes peticiones podrían "apuntar" al mismo dato.

Debido a que estamos utilizando HTTP para comunicarnos, podemos transferir cualquier tipo de información que pueda transportarse entre clientes y servidores. Por ejemplo, si realizamos una petición de un fichero de texto de la CNN, nuestro navegador mostrará un fichero de texto. Si solicitamos una película flash a YouTube, nuestro navegador recibirá una película flash. En ambos casos, los datos son transferidos sobre TCP/IP y el navegador conoce cómo interpretar los *streams* binarios debido a la cabecera de respuesta del protocolo HTTP *Content-Type*. Por lo tanto, en un sistema RESTful, la representación de un recurso depende del tipo deseado por el cliente (tipo MIME), el cual está especificado en la petición del protocolo de comunicaciones.

## Representación de los recursos

La representación de los recursos es lo que se envía entre los servidores y clientes. Una representación muestra el estado temporal del dato real almacenado en algún dispositivo de almacenamiento en el momento de la petición. En términos generales, es un *stream* binario, juntamente con los metadatos que describen cómo dicho *stream* debe ser consumido por el cliente y/o servidor (los metadatos también puden contener información extra sobre el recurso, como por ejemplo información de validación y encriptación, o código extra para ser ejecutado dinámicamente).

A través del ciclo de vida de un servicio web, pueden haber varios clientes solicitando recursos. Clientes diferentes son capaces de consumir diferentes representaciones del mismo recurso. Por lo tanto, una representación puede tener varias formas, como por ejemplo, una imagen, un texto, un fichero XML, o un fichero JSON, pero tienen que estar disponibles en la misma URL.

Para respuestas generadas para humanos a través de un navegador, una representación típica tiene la forma de página HTML. Para respuestas automáticas de otros servicios web, la legibilidad no es importante y puede utilizarse una representación mucho más eficiente como por ejemplo XML.

El lenguaje para el intercambio de información con el servicio queda a elección del desarrollador. A continuación mostramos algunos formatos comunes que podemos utilizar para intercambiar esta información:

### Table 1. Ejemplos de formatos utilizados por los servicios REST

| Formato | Tipo MIME |
|---|---|
| Texto plano | text/plain |
| HTML | text/html |
| XML | application/xml |
| JSON | application/json |

De especial interés es el formato **JSON**. Se trata de un lenguaje ligero de intercambio de información, que puede utilizarse en lugar de XML (que resulta considerablemente más pesado) para aplicaciones AJAX. De hecho, en Javascript puede leerse este tipo de formato simplemente utilizando el método `eval()`.

## Direccionabilidad de los recursos: URI

Una URI, o **Uniform Resource Identifier**, en un servicio web RESTful es un hiper-enlace a un recurso, y es la única forma de intercambiar representaciones entre clientes y servidores. Un servicio web RESTful expone un conjunto de recursos que identifican los objetivos de la interacción con sus clientes.

El conjunto de restricciones REST no impone que las URIs deban ser hiper-enlaces. Simplemente hablamos de hiper-enlaces porque estamos utilizando la Web para crear servicios web. Si estuviésemos utilizando un conjunto diferente de tecnologías soportadas, una URI RESTful podría ser algo completamente diferente. Sin embargo, la idea de direccionabilidad debe permanecer.

En un sistema REST, la URI **no cambia a lo largo del tiempo**, ya que la implementación de la arquitectura es la que gestiona los servicios, localiza los recursos, negocia las representaciones, y envía respuestas con los recursos solicitados. Y lo que es más importante, si hubiese un cambio en la estructura del dispositivo de almacenamiento en el lado del servidor (por ejemplo, un cambio de servidores de bases de datos), nuestras URIs seguirán siendo las mismas y serán válidas mientras el servicio web siga estando "en marcha" o el contexto del recurso no cambie.

**Sin** las restricciones REST, los recursos se acceden por su localización: las direcciones web típicas son URIs fijas. Si por ejemplo renombramos un fichero en el servidor, la URI será diferente; si movemos el fichero a un directorio diferente, la URI también será diferente.

El formato de una URI se estandariza como sigue:

```
scheme://host:port/path?queryString#fragment
```

En donde:

- `scheme` es el protocolo que estamos utilizando para comunicarnos con el servidor. Para servicios REST, normalmente el protocolo será **http** o **https**.
- El término `host` es un nombre DNS o una dirección IP.
- A continuación se puede indicar de forma opcional un puerto (mediante `:port`), que es un valor numérico. El **host** y el **port** representan la localización de nuestro recurso en la red.
- Seguidamente aparece una expresión `path`, que es un conjunto de segmentos de texto delimitados por el carácter `\` (pensemos en la expresión *path* como en una lista de directorios de un fichero en nuestra máquina).
- Esta expresión puede ir seguida, opcionalmente por una `queryString`. El carácter `?`) separa el *path* de la *queryString*. Esta última es una lista de parámetros representados como pares nombre/valor. Cada par está delimitado por el carácter `&`.

La última parte de la URI es el `fragment`, delimitado por el carácter `#`. Normalmente se utiliza para "apuntar" a cierto "lugar" del documento al que estamos accediendo.

En una URI, no todos los caracteres están permitidos, de forma que algunos caracteres se codificarán de acuerdo a las siguientes reglas:

- Los caracteres a-z, A-Z, 0-9, ., -, *, y _, permanecen igual
- El caracter "espacio" se convierte en el carácter `+`
- El resto de caracteres se codifican como una secuencia de bits siguiendo un esquema de codificación hexadecimal, de forma que cada dos dígitos hexadecimales van precedidos por el carácter `%`.

Un ejemplo de URI podría ser éste:

```
http://expertojava.ua.es/recursos/clientes?
apellido=Martinez&codPostal=02115
```

En el ejemplo anterior el **host** viene dado por *expertojava.ua.es*, el **path** o ruta de acceso al recurso es */recursos/clientes*, y hemos especificado los **parámetros** *apellido* y *codPostal*, con los **valores** *Martinez* y *02115* respectivamente.

Si por ejemplo, en nuestra aplicación tenemos información de clientes, podríamos acceder a la lista correspondiente mediante una URL como la siguiente:

```
http://expertojava.ua.es/recursos/clientes
```

Esto nos devolverá la lista de clientes en el formato que el desarrollador del servicio haya decidido. Hay que destacar, por lo tanto, que en este caso debe haber un entendimiento entre el consumidor y el productor del servicio, de forma que el primero comprenda el lenguaje utilizado por el segundo.

La URL anterior nos podría devolver un documento como el siguiente:

```
<?xml version="1.0"?>
<clientes>
  <cliente>http://expertojava.ua.es/recursos/clientes/1"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/clientes/2"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/clientes/4"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/clientes/6"<cliente/>
</clientes>
```

En este documento se muestra la lista de clientes registrados en la aplicación, cada uno de ellos representado también por una URL. Accediendo a estas URLs, a su vez, podremos obtener información sobre cada curso concreto o bien modificarlo.

## Uniformidad y restricciones de las interfaces

Ya hemos introducido los conceptos de recursos y sus representaciones. Hemos dicho que los **recursos** son correspondencias (*mappings*) de los estados reales de las entidades que son intercambiados entre los clientes y servidores. También hemos dicho que las **representaciones** son negociadas entre los clientes y servidores a través del protocolo de comunicación en tiempo de ejecución (a través de HTTP). A continuación veremos con detalle

lo que significa el intercambio de estas representaciones, y lo que implica para los clientes y servidores el realizar acciones sobre dichos recursos.

El desarrollo de servicios web REST es similar al desarrollo de aplicaciones web. Sin embargo, la diferencia fundamental entre el desarrollo de aplicaciones web tradicionales y las más modernas es cómo pensamos sobre las acciones a realizar sobre nuestras abstracciones de datos. De forma más concreta, el desarrollo moderno está centrado en el concepto de **nombres** (intercambio de recursos); el desarrollo tradicional está centrado en el concepto de verbos (acciones remotas a realizar sobre los datos). Con la primera forma, estamos implementando un servicio web RESTful; con la segunda un servicio similar a una llamada a procedimiento remoto- RPC). Y lo que es más, un servicio RESTful **modifica el estado de los datos a través de la representación de los recursos** (por el contrario, una llamada a un servicio RPC, oculta la representación de los datos y en su lugar envía comandos para modificar el estado de los datos en el lado del servidor). Finalmente, en el desarrollo moderno de aplicaciones web limitamos la ambigüedad en el diseño y la implementación debido a que tenemos **cuatro** acciones específicas que podemos realizar sobre los recursos: *Create, Retrieve, Update y Delete* (CRUD). Por otro lado, en el desarrollo tradicional de aplicaciones web, podemos tener otras acciones con nombres o implementaciones no estándar.

A continuación indicamos la correspondencia entre las acciones CRUD sobre los datos y los métodos HTTP asociados:

### Table 2. Operaciones REST sobre los recursos:

| Acción sobre los datos | Protocolo HTTP equivalente |
|---|---|
| CREATE | POST |
| RETRIEVE | GET |
| UPDATE | PUT |
| DELETE | DELETE |

El principio de uniformidad de la interfaz de acceso a recursos es fundamental, y quizá el más difícil de seguir por los programadores acostumbrados al modelo RPC (**R**emote **P**rocedure **C**all). La idea subyacente es utilizar únicamente un conjunto finito y claramente establecido de operaciones para la interacción con los servicios. Esto significa que no tendremos un parámetro "acción" en nuestra URI y que sólo utilizaremos los métodos HTTP para acceder a nuestros servicios. Cada uno de los métodos tiene un propósito y significado específicos, que mostramos a continuación:

**GET**

GET es una operación **sólo de lectura**. Se utiliza para "recuperar" información específica del servidor. También se trata de una operación **idempotente** y **segura**. **Idempotente** significa que no importa cuántas veces invoquemos esta operación, el resultado (que observaremos como usuarios) debe ser siempre el mismo. **Segura** significa que una operación GET no cambia el estado del servidor en modo alguno, es decir, no debe exhibir ningún efecto lateral en el servidor. Por ejemplo, el hecho de "leer" un documento HTML no debería cambiar el estado de dicho documento.

**PUT**

La operación PUT solicita al servidor el almacenar el cuerpo del mensaje enviado con dicha operación en la dirección proporcionada en el mensaje HTTP. Normalmente se modela como una inserción o actualización (nosotros la utilizaremos solamente como actualización). Es una propiedad **idempotente**. Cuando se utiliza PUT, el cliente conoce

la identidad del recurso que está creando o actualizando. Es idempotente porque enviar el mismo mensaje PUT más de una vez no tiene ningún efecto sobre el servicio subyacente. Una analogía podría ser un documento de texto que estemos editando. No importa cuántas veces pulsemos el "botón" de grabar, el fichero que contiene el documento lógicamente será el mismo documento.

**DELETE**
Esta operación se utiliza para eliminar recursos. También es **idempotente**

**POST**
Post es la única operación HTTP que no es idempotente ni segura. Cada petición POST puede modificar el servicio de forma exclusiva. Se puede enviar, o no, información con la petición. También podemos recibir, o no, información con la respuesta. Para implementar servicios REST, es deseable enviar información con la petición y también recibir información con la respuesta.

Adicionalmente, podemos utilizar otras dos operaciones HTTP:

**HEAD**
Es una operación exactamente igual que GET, excepto que en lugar de devolver un "cuerpo de mensaje", solamente devuelve un código de respuesta y alguna cabecera asociada con la petición.

**OPTIONS**
Se utiliza para solicitar información sobre las opciones disponibles sobre un recurso en el que estamos interesados. Esto permite al cliente determinar las capacidades del servidor y del recurso sin tener que realizar ninguna petición que provoque una acción sobre el recurso o la recuperación del mismo.

**PATCH**
Se utiliza para para realiza reemplazos (actualizaciones) parciales de un documento, ya que la operación PUT sólo permite una actualización completa del recurso (y requiere indicar una representación completa del mismo) . Es útil cuando el recurso a modificar es complejo y solamente queremos actualizar parte de su contenido. En este caso solo necesitamos indicar la parte que queremos cambiar.

## 1.3. Diseño de servicios Web RESTful

El diseño de servicios RESTful no es muy diferente del diseño de aplicaciones web tradicionales: tenemos requerimientos de negocio, tenemos usuarios que quieren realizar operaciones sobre los datos, y tenemos restricciones *hardware* que van a condicionar nuestra arquitectura *software*. La principal diferencia reside en el hecho de que tenemos que "buscar", a partir de los requerimientos, cuáles van a ser los **recursos** que van a ser accedidos a través de los servicios, "sin preocuparnos" de qué **operaciones** o acciones específicas van a poderse realizar sobre dichos recursos (el proceso de diseño depende de los "nombres", no de los "verbos").

Podemos resumir los principios de diseño de servicios web RESTful en los siguientes cuatro pasos:

1. Elicitación de requerimientos y creación del **modelo de objetos**: Este paso es similar al diseño orientado a objetos. El resultado del proceso puede ser un modelo de clases UML
2. Identificación de **recursos**: Este paso consiste en identificar los "objetos" de nuestro modelo sin preocuparnos de las operaciones concretas a realizar sobre dichos objetos

3. Definición de las **URIs**: Para satisfacer el principio de "direccionabilidad" de los recursos, tendremos que definir las URIs que representarán los *endpoints* de nuestros servicios, y que constituirán los "puntos de entrada" de los mismos
4. Definición de la **representación de los recursos**: Puesto que los sistemas REST están orientados a la representación, tendremos que definir el formato de los datos que utilizaremos para intercambiar información entre nuestros servicios y clientes
5. Definición de los **métodos de acceso** a los recursos: Finalmente, tendremos que decidir qué métodos HTTP nos permitirán acceder a las URIs que queremos exponer, así como qué hará cada método. Es muy importante que en este paso, nos ciñamos a las restricciones que definen los principios RESTful que hemos indicado en apartados anteriores.

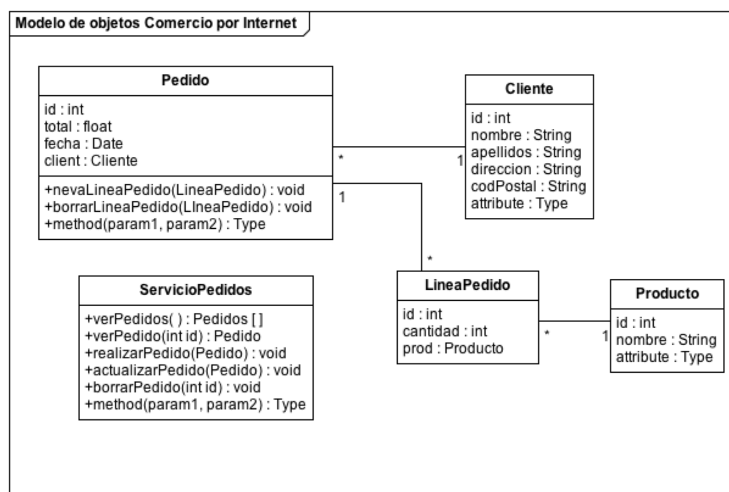
## 1.4. Un primer servicio JAX-RS

Vamos a ilustrar los pasos anteriores con un ejemplo, concretamente definiremos una interfaz RESTful para un sistema sencillo de gestión de pedidos de un hipotético comercio por internet. Los potenciales clientes de nuestro sistema, podrán realizar compras, modificar pedidos existentes en nuestro sistema, así como visualizar sus datos personales o la información sobre los productos que son ofertados por el comercio.

### Modelo de objetos

A partir de los requerimientos del sistema, obtenemos el modelo de objetos. El modelo de objetos de nuestro sistema de ventas por internet es bastante sencillo. Cada pedido en el sistema representa una única transacción de compra y está asociada con un cliente particular. Los pedidos estarán formados por una o más líneas de pedido. Las líneas de pedido representan el tipo y el número de unidades del producto adquirido.

Basándonos en esta descripción de nuestro sistema, podemos extraer que los objetos de nuestro modelo son: **Pedido**, **Cliente**, **LineaPedido**, y **Producto**. Cada objeto de nuestro modelo tiene un identificador único, representado por la propiedad `id`, dada por un valor de tipo entero. La siguiente figura muestra un diagrama UML de nuestro modelo:




Estamos interesados en consultar todos los pedidos realizados, así como cada pedido de forma individual. También queremos poder realizar nuevos pedidos, así como actualizar

pedidos existentes. El objeto ServicioPedidos representa las operaciones que queremos realizar sobre nuestos objetos *Pedido*, *Cliente*, *LineaPedido* y *Producto*.

## Modelado de URIs

Lo primero que haremos para crear nuestra interfaz distribuida, es definir y poner nombre a cada uno de los *endpoints* de nuestro sistema. En un sistemam RESTful, los *endpoints* serán los **recursos** del sistema, que identificaremos mediante URIs.

En nuestro modelo de objetos queremos poder interactuar con *Pedidos*, *Clientes*, y *Productos*. Éstos serán, por lo tanto, nuestros recursos de nivel más alto. Por otro lado, estamos interesados en obtener una lista de cada uno de estos elementos de alto nivel, así como interactuar con los elementos indiduales de cada tipo. El objeto *LineaPedido* es un objeto *agregado* del objeto *Pedido* por lo que no lo consideraremos com un recurso de nivel superior. Más adelante veremos que podremos exponerlo como un **subrecurso** de un *Pedido* particular, pero por ahora, asumiremos que está "oculto" por el formato de nuestros datos. Según esto, una posible lista de URIs que expondrá nuestro sistema podría ser:

- /pedidos
- /pedidos/{id}
- /productos
- /productos/{id}
- /clientes
- /clientes/{id}

Fíjate que hemos representado como URIs los **nombres** en nuestro modelo de objetos. Recuerda que las URIS no deberían utilizarse como mini-mecanismos de RPC ni deberían identificar operaciones. En vez de eso, tenemos que utilizar una combinación de métodos HTTP y de datos (recursos) para modelar las operaciones de nuestro sistema RESTful

## Definición del formato de datos

Una de las cosas más importantes que tenemos que hacer cuando definimos la interfaz RESTful es determinar cómo se representarán los recursos que serán accedidos por los usuarios de nuestro API REST. Quizá XML sea el formato más popular de la web y puede ser procesado por la mayor parte de los lenguajes de programación. Como veremos más adelante, JSON es otro formato popular, menos "verboso" que XML, y que puede ser interpretado directamente por *JavaScript* (lo cual es perfecto para aplicaciones Ajax por ejemplo). Por ahora, utilizaremos el formato XML en nuestro ejemplo.

Generalmente, tendríamos que definir un esquema XML para cada representación que queramos transimitir a traves de la red. Un esquema XML define la gramática del formato de datos. Por simplicidad, vamos a omitir la creación de esquemas, asumiendo que los ejemplos que proporcionamos se adhieren a sus correspondientes esquemas.

A continuación distinguiremos entre **dos formatos de datos**: uno para las operaciones de lectura y actualización, y otro para la operación de creación de recursos.

### Formato de datos para operaciones de lectura y modificación de los recursos

Las representaciones de los recursos *Pedido*, *Cliente*, y *Producto* tendrán un **elemento XML en común**, al que denominaremos `link`:

```
<link rel="self" href="http://org.expertojava/..."/>
```

El elemento (o etiqueta) `link` indica a los clientes que obtengan un documento XML como representación del recurso, dónde pueden interaccionar en la red con dicho recurso en particular. El atributo `self` le indica al cliente qué relación tiene dicho enlace con la URI del recurso al que apunta (información contenida en el atributo `href`). El valor `self` indica que está "apuntando" a sí mismo. Más adelante veremos la utilidad del elemento **link** cuando agreguemos información en documentos XML "más grandes".

El formato de representación del recurso **Cliente** podría ser:

```
<cliente id="8">
   <link rel="self"
         href="http://org.expertojava/clientes/8"/>
   <nombre>Pedro</nombre>
   <apellidos>Garcia Perez</apellidos>
   <direccion>Calle del Pino, 5<direccion>
   <codPostal>08888</codPostal>
   <ciudad>Madrid</ciudad>
</cliente>
```

El formato de representación del recurso **Producto** podría ser:

```
<producto id="34">
   <link rel="self"
         href="http://org.expertojava/productos/34"/>
   <nombre>iPhone 6</nombre>
   <precio>800</precio>
   <cantidad>1</cantidad>
</producto>
```

Finalmente, el formato de la representación del recurso **Pedido** podría ser:

```
<pedido id="233">
  <link rel="self" href="http://org.expertojava/pedidos/233"/>
  <total>800</total>
  <fecha>December 22, 2014 06:56</fecha>
  <cliente id="8">
     <link rel="self"
                href="http://org.expertojava/clientes/8"/>
     <nombre>Pedro</nombre>
     <apellidos>Garcia Perez</apellidos>
     <direccion>Calle del Pino, 5<direccion>
     <codPostal>08888</codPostal>
     <ciudad>Madrid</ciudad>
  </cliente>
  <lineasPedido>
    <lineaPedido id="1">
      <producto id="34">
         <link rel="self"
             href="http://org.expertojava/productos/34"/>
         <nombre>iPhone 6</nombre>
```

```
        <precio>800</precio>
        <cantidad>1</cantidad>
      </producto>
    <lineaPedido/>
  <lineasPedido/>
</pedido>
```

El formato de datos de un `Pedido` tiene en un primer nivel la información del `total`, con el importe total del pedido, así como la `fecha` en la que se hizo dicho pedido. *Pedido* es un buen ejemplo de composición de datos, ya que un pedido incluye información sobre el *Cliente* y el *Producto/s*. Aquí es donde el elemento `<link>` puede resultar particularmente útil. Si el usuario está interesado en interaccionar con el *Cliente* que ha realizado el pedido, o en uno de los *productos* del mismo, se proporciona la URI necesaria para interactuar con cada uno de dichos recursos. De esta forma, cuando el usuario del API consulte un pedido, podrá además, acceder a información adicional relacionada con la consulta realizada.

### Formato de datos para operaciones de creación de los recursos

Cuando estamos creando nuevos *Pedidos*, *Clientes* o *Productos*, no tiene mucho sentido incluir un atributo **id** y un elemento **link** en nuestro documento XML. El servidor será el encargado de crear los **ids** cuando inserte nuestro nuevo objeto en la base de datos. Tampoco conocemos la URI del nuevo objeto creado, ya que será el servidor el encargado de generarlo. Por lo tanto, para crear un nuevo *Producto*, el formato de la información podría ser el siguiente:

```
<producto>
   <link rel="self"
         href="http://org.expertojava/clientes/8"/>
   <nombre>iPhone</nombre>
   <precio>800</precio>
</producto>
```

## Asignación de métodos HTTP

Finalmente, tendremos que decidir qué métodos HTTP expondremos en nuestro servicio para cada uno de los recursos, así como definir qué harán dichos métodos. Es muy importante no asignar funcionaldad a un método HTTP que "sobrepase" los límites impuestos por la especificación de dicho método. Por ejemplo, una operación GET sobre un recurso concreto debería ser de sólo lectura. No debería cambiar el estado del recurso cuando invoquemos la operación GET sobre él. Si no seguimos de forma estricta la semántica de los métodos HTTP, los clientes, así como cualquier otra herramienta administrativa, no pueden hacer asunciones sobre nuestros servicios, de forma que nuestro sistema se vuelve más complejo.

Veamos, para cada uno de los métodos de nuestro modelo de objetos, cuales serán las URIs y métodos HTTP que usaremos para representarlos.

### Visualización de todos los *Pedidos*, *Clientes* o *Productos*

Los tres objetos de nuestro modelo: *Pedidos*, *Clientes* y *Productos*, son accedidos y manipulados de forma similar. Los usuarios pueden estar interesados en ver **todos** los *Pedidos*, *Clientes* o *Productos* en el sistema. Las siguientes URIs representan dichos objetos como un grupo:

- /pedidos

- /productos
- /clientes

Para obtener una lista de *Pedidos*, *Clientes* o *Productos*, el cliente remoto realizara una llamada al método HTTP GET sobre la URI que representa el grupo de objetos. Un ejemplo de petición podría ser la siguiente:

```
GET /productos HTTP/1.1
```

Nuestro servicio responderá con los datos que representan todos los *Pedidos* de nuestro sistema. Una respuesta podría ser ésta:

```
HTTP/1.1 200 OK
Content-Type: application/xml

<productos>
   <producto id="111">
      <link rel="self" href="http://org.expertojava/productos/111"/>
      <nombre>iPhone</nombre>
      <precio>648.99</precio>
   </producto>
   <producto id="222">
      <link rel="self" href="http://org.expertojava/productos/222"/>
      <nombre>Macbook</nombre>
      <precio>1599.99</precio>
   </producto>
    ...
</productos>
```

Un problema que puede darse con esta petición es que tengamos miles de *Pedidos*, *Clientes* o *Productos* en nuestro sistema, por lo que podemos "sobrecargar" a nuestro cliente y afectar negativamente a los tiempos de respuesta. Para mitigar esta problema, permitiremos que el usuario especifique unos parámetros en la URI para limitar el tamaño del conjunto de datos que se va a devolver:

```
GET /pedidos?startIndex=0&size=5 HTTP/1.1
GET /productos?startIndex=0&size=5 HTTP/1.1
GET /clientes?startIndex=0&size=5 HTTP/1.1
```

En las órdenes anteriores, hemos definido dos parámetros de petición: `startIndex`, y `size`. El primero de ellos es un índice numérico que representa a partir de qué posición en la lista de *Pedidos*, *Clientes* o *Productos*, comenzaremos a enviar la información al cliente. El parámetro `size` especifica cuántos de estos objetos de la lista queremos que nos sean devueltos.

Estos parámetros serán opcionales, de forma que el cliente no tiene que especificarlos en su URI.

### Obtención de *Pedidos*, *Clientes* o *Productos* individuales

Ya hemos comentado previamente que podríamos utilizar las siguientes URIs para obtener *Pedidos*, *Clientes* o *Productos*:

- /pedidos/{id}
- /productos/{id}
- /clientes/{id}

En este caso usaremos el método HTTP GET para recuperar objetos individuales en el sistema. Cada invocación GET devolverá la información del correspondiente objeto. Por ejemplo:

```
GET /pedidos/233 HTTP/1.1
```

Para esta petición, el cliente está interesado en obtener una representación del *Pedido* con identificador 233. Las peticiones GET para *Productos* y *Clientes* podrían funcionar de forma similar. El mensaje de respuesta podría parecerse a algo como esto:

```
HTTP/1.1 200 OK
Content-Type: application/xml

<pedido id="233">...</pedido>
```

El **código de respuesta** es **200 OK**, indicando que la petición ha tenido éxito. La cabecera `Content-Type` especifica el formato del cuerpo de nuestro mensaje como XML, y finalmente obtenemos la representación del *Pedido* solicitado.

### Creación de un *Pedido*, *Cliente* o *Producto*

Para crear un *Pedido*, *Cliente* o *Producto* utilizaremos el método POST. En este caso, el cliente envía una representación del nuevo objeto que se prentende crear a la URI "padre" de su representación, y por lo tanto, podremos omitir el identificador del recurso. Por ejemplo:

#### Petición POST para crear un pedidio

```
POST /pedidos HTTP/1.1
Content-Type: application/xml

<pedido>
   <total>199.02</total>
   <fecha>December 22, 2008 06:56</fecha>
   ...
</pedido>
```

El servicio recibe el mensaje POST, procesa la XML, y crea un nuevo pedido en la base de datos utilizando un identificador generado de forma única. Si bien esta aproximación "funciona" perfectamente, se le pueden plantear varias cuestiones al usuario. ¿Qué ocurre si el usuario quiere visualizar, modificar o eliminar el pedido que acaba de crear? ¿Cuál es el identificador del nuevo recurso? ¿Cuál es la URI que podemos utilizar para interactuar con el nuevo recurso? Para resolver estas cuestiones, añadiremos alguna información al mensaje de respuesta HTTP. El cliente podría recibir un mensaje similar a éste:

#### Respuesta de una petición POST para crear un pedido

```
HTTP/1.1 201 Created
```

```
Content-Type: application/xml
Location: http://org.expertojava/pedidos/233

<pedido id="233">
    <link rel="self" href="http://org.expertojava/pedidos/233"/>
    <total>199.02</total>
    <fecha>December 22, 2008 06:56</fecha>
    ...
</pedido>
```

HTTP requiere que si POST crea un nuevo recurso, se debe responder con un código `201 Created`. También se requieer que la cabecera `Location` en el mensaje de respuesta proporcione una URI al usuario que ha hecho la petición para que éste pueda interactuar con la *Petición* que acaba de crear (por ejemplo, para modificar dicho *Pedido*). Es **opcional** por parte del servidor devolver en la respuesta la representación del nuevo recurso creado. En nuestro ejemplo optamos por devolver una representación XML de la *Peticion* creada con el identificador del atributo así como el elemento `link`.

## Actualización de un *Pedido*, *Cliente* o *Producto*

Para realizar modificaciones sobre los recursos que ya hemos creado utilizaremos el método PUT. En este caso, un ejemplo de petición podría ser ésta:

**Petición PUT para modificar un pedidio**

```
PUT /pedidos/233 HTTP/1.1
Content-Type: application/xml

<producto id="111">
   <nombre>iPhone</nombre>
   <precio>649.99</precio>
</producto>
```

Tal y como he hemos indicado anteriormente, la operación PUT es **idempotente**. Lo que significa que no importa cuántas veces solicitemos la petición PUT, el producto subyacente sigue permaneciendo con el mismo estado final.

Cuando un recurso se modifica mediante PUT, la especificación HTTP requiere que el servidor envíe un código de respuesta `200 OK`, y un cuerpo de mensaje de respuesta, o bien el código `204 No Content`, sin ningún cuerpo de mensaje en la respuesta.

En nuestro caso, devolveremos un código de estado **204** y un mensaje sin cuerpo de respuesta.

**RECUERDA: Es importante NO confundir POST con PUT**

Muchas veces se confunden los métodos PUT y POST. El significado de estos métodos es el siguiente:

- **POST**: Publica datos en un determinado recurso. El recurso debe existir previamente, y los datos enviados son añadidos a él. Por ejemplo, para añadir nuevos pedidos con POST hemos visto que debíamos hacerlo con el recurso lista de pedidos (/pedidos), ya que la URI del nuevo pedido todavía no existe. La operación **NO es idempotente**, es decir, si

> añadimos varias veces el mismo alumno aparecerá repetido en nuestra lista de pedidos con URIs distintas.
>
> - **PUT**: Hace que el recurso indicado tome como contenido los datos enviados. El recurso podría no existir previamente, y en caso de que existiese sería sobrescrito con la nueva información. A diferencia de POST, **PUT es idempotente**: Múltiples llamadas idénticas a la misma acción PUT siempre dejarán el recurso en el **mismo estado**. La acción se realiza sobre la URI concreta que queremos establecer (por ejemplo, /pedidos/215), de forma que varias llamadas consecutivas con los mismos datos tendrán el mismo efecto que realizar sólo una de ellas.

### Borrado de un *Pedido*, *Cliente* o *Producto*

Modelaremos el borrado de los recursos utilizando el método HTTP DELETE. El usuario simplemente invocará el método DELETE sobre la URI que representa el objeto que queremos eliminar. Este método hará que dicho recurso ya no exista en nuestro sistema.

Cuando eliminamos un recurso con DELETE, la especificación requiere que se envíe un código de respuesta `200 OK`, y un cuerpo de mensaje de respuesta, o bien un código de respuesta `204 No Content`, sin un cuerpo de mensaje de respuesta.

En nuestro caso, devolveremos un código de estado **204** y un mensaje sin cuerpo de respuesta.

### Cancelación de un *Pedido*

Hasta ahora, las operaciones de nuestro modelo de objetos "encajan" bastante bien en la especificación de los correspondientes métodos HTTP. Hemos utilzado GET para leer datos, PUT para realizar modificaciones POST para crear nuevos recursos, y DELETE para eliminarlos. En nuestro sistema, los *Pedidos* pueden eliminarse, o también cancelarse. Ya hemos comentado que el borrado de un recurso lo "elimina completamente" de nuestra base de datos. La operación de cancelación solamente cambia el estado del *Pedido*, y lo sigue manteniendo en el sistema. ¿Cómo podríamos modelar esta operación?

Cuando modelamos una interfaz RESTful para las operaciones de nuestro modelo de objetos, deberíamos plantearnos la siguiente pregunta: ¿la operación es un estado del recurso? Si la respuesta es sí, entonces deberíamos modelar esta operación "dentro" del formato de los datos.

La cancelación de un pedido es un ejemplo perfecto de esto que acabamos de decir. La clave está en que esta operación, en realidad es un estado específico del *Pedido*: éste puede estar cancelado o no. Cuando un usuario accede a un *Pedido*, puede desear conocer si el *Pedido* ha sido o no cancelado. Por lo tanto, la información sobre la cancelación debería formar parte del formato de datos de un *Pedido*. Así, añadiremos un nuevo elemento a la información del *Pedido*:

```
<pedido id="233">
    <link rel="self" href="http://org.expertojava/pedidos/233"/>
    <total>199.02</total>
    <fecha>December 22, 2008 06:56</fecha>
    <cancelado>false</cancelado>
    ...
```

```
</pedido>
```

Ya que el estado "cancelado" se modela en el propio formato de datos, modelaremos la acción de cancelación con una operación HTTP PUT, que ya conocemos:

```
PUT /pedidos/233 HTTP/1.1
Content-Type: application/xml

<pedido id="233">
    <link rel="self" href="http://org.expertojava/pedidos/233"/>
    <total>199.02</total>
    <fecha>December 22, 2008 06:56</fecha>
    <cancelado>true</cancelado>
    ...
</pedido>
```

En este ejemplo, modificamos la representación del *Pedido* con el elemento `<cancelado>` con valor `true`.

Este "patrón" de modelado, no siempre "sirve" en todos los casos. Por ejemplo, imaginemos que queremos ampliar el sistema de forma que "borremos" del sistema todos los pedidos cancelados. No podemos modelar esta operación de la misma manera que la de cancelación, ya que esta operación no cambia el estado de nuestra aplicación (no es en sí misma un estado de la aplicación).

Para resolver este problema, podemos modelar esta nueva operación como un "subrecurso" de */pedidos*, y realizar un borrado de los pedidos cancelados, mediante el método HTTP POST de dicho subrecurso, de la siguiente forma:

```
POST /pedidos/eliminacion HTTP/1.1
```

Un efecto interesante de lo que acabamos de hacer es que, puesto que ahora `eliminacion` es una URI, podemos hacer que la interfaz de nuestro servicios RESTful evolucionen con el tiempo. Por ejemplo, la orden **GET /pedidos/eliminacion** podría devolver la última fecha en la que se procedió a eliminar todos los pedidos cancelados, así como qué pedidos fueron cancelados. ¿Y si queremos añadir algún criterio a la hora de realizar el borrado de pedidos cancelados? Podríamos introducir parámetros para indicar que sólo queremos eliminar aquellos pedidos que estén cancelados en una fecha anterior a una dada. Como vemos, podemos mantener una interfaz uniforme y ceñirnos a las operaciones HTTP tal y como están especificadas, y a la vez, dotar de una gran flexiblidad a la interfaz de nuestro sistema RESTful. Hablaremos con más detalle de los subrecursos en la siguiente sesión.

## Implementación del servicio: Creación del proyecto Maven

Vamos a utilizar Maven para crear la estructura del proyecto que contendrá la implementación de nuestro servicio Rest. Inicialmente, podemos utilizar el mismo arquetipo con el que habéis trabajado en sesiones anteriores. Y a continuación modificaremos la configuración del fichero *pom.xml*, para implementar nuestros servicios.

Una opción es generar la estructura del proyecto directamente **desde línea de comandos**. El comando es el siguiente (recuerda que debes escribirlo en una misma línea. Los caracteres

"\" que aparecen en el comando no forman parte del mismo, simplemente indican que no se debe pulsar el retorno de carro):

```
mvn --batch-mode archetype:generate \
      -DarchetypeGroupId=org.codehaus.mojo.archetypes \
      -DarchetypeArtifactId=webapp-javaee7 \
      -DgroupId=org.expertojava -DartifactId=ejemplo-rest
```

En donde:

- `archetypeGroupId` y `archetypeArtifactId` son los nombres del *groupId* y *artifactId* del **arquetipo Maven** que nos va a generar la "plantilla" para nuestro proyecto
- `groupId` y `artifactId` son los nombres que asignamos como *groupId* y *artifactId* de **nuestro proyecto**. En este caso hemos elegido los valores *org.expertojava* y *ejemplo-rest*, respectivamente

## Si utilizamos IntelliJ para crear el proyecto tenemos que:

1. Crear un nuevo proyecto (**New Project**)
2. Elegir el tipo de proyecto **Maven**
3. Crear el proyecto Maven a partir de un **arquetipo** con las siguientes **coordenadas**:
    - GroupId: *org.codehaus.mojo.archetypes*
    - ArtifactId: *webapp-javaee7*
    - Version: *1.1*
4. Indicar las **coordenadas** de nuestro **proyecto**:
    - GroupId: *org.expertojava*
    - ArtifactId: *ejemplo-rest*
    - Version: *1.0-SNAPSHOT*
5. Confirmamos los datos introducidos
6. Para finalizar, especificamos el nombre de nuestro **proyecto en IntelliJ**:
    - Project Name: *ejemplo-rest* (este valor también identificará el nombre del módulo en IntelliJ)
7. Por comodidad, marcaremos *Enable autoimport* para importar automáticamente cualquier cambio en el proyecto

Una vez que hemos creado el proyecto con IntelliJ, el paso siguiente es cambiar la configuración del *pom.mxl* que nos ha generado el arquetipo, para incluir las *propiedades*, *dependencias*, *plugins*,…, que necesitaremos para implementar nuestros recursos REST.

Como ya sabemos, el fichero *pom.xml* contiene la configuración que utiliza Maven para construir el proyecto. A continuación indicamos las **modificaciones en el fichero pom.xml** generado inicialmente, para adecuarlo a nuestras necesidades particulares:

- Cambiamos las **propiedades** del proyecto (etiqueta `<properties>`) por:

### Propiedades del proyecto

```
<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

- Indicamos las **dependencias** del proyecto (etiqueta `<dependencies>`, en donde se incluyen las librerías necesarias para la construcción del proyecto). En nuestro caso, necesitamos incluir la librería `javax:javaee-web-api:7.0` que contiene el api estándar de javaee 7. Marcamos el ámbito de la librería (etiqueta `<scope>`) como `provided`. Con esto estamos indicando que sólo necesitaremos el jar correspondiente para "compilar" el proyecto, y por lo tanto **no incluiremos** dicho *jar*, en el fichero **war** generado para nuestra aplicación, ya que dicha librería ya estará disponible desde el servidor de aplicaciones en el que residirá nuestra aplicación.

### Librerías utilizadas para construir el proyecto (dependencias)

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

- A continuación configuramos la **construcción** del proyecto (etiqueta `<build>`), de la siguiente forma (cambiamos la configuración original por la que mostramos a continuación):

### Configuración de la construcción del proyecto

```
<build>
  <!-- Especificamos el nombre del war que será usado como context root
       cuando despleguemos la aplicación -->
  <finalName>${project.artifactId}</finalName>

  <plugins>
    <!-- Compilador de java. Utilizaremos la versión 1.7 -->
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.1</version>
      <configuration>
          <source>1.7</source>
          <target>1.7</target>
      </configuration>
    </plugin>

    <!-- Servidor de aplicaciones wildfly -->
    <plugin>
      <groupId>org.wildfly.plugins</groupId>
      <artifactId>wildfly-maven-plugin</artifactId>
      <version>1.0.2.Final</version>
      <configuration>
          <hostname>localhost</hostname>
```

```
        <port>9990</port>
      </configuration>
    </plugin>

    <!-- Cuando generamos el war no es necesario
         que el fichero web.xml esté presente -->
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-war-plugin</artifactId>
        <version>2.3</version>
        <configuration>
            <failOnMissingWebXml>false</failOnMissingWebXml>
        </configuration>
    </plugin>
  </plugins>
</build>
```

## Implementación del servicio: Recursos JAX-RS

Una vez que tenemos la estructura del proyecto, implementaremos los **recursos** de nuestra aplicación, que serán clases Java que utilizarán anotaciones JAX-RS para enlazar y mapear peticiones HTTP específicas a métodos java, los cuales servirán dichas peticiones. En este caso, vamos a ilustrar con un ejemplo, una posible implementación para el recurso *Cliente*. Tenemos que diferenciar entre las clases java que representarán **entidades de nuestro dominio** (objetos java que representan elementos de nuestro negocio, y que serán almacenados típicamente en una base de datos), de nuestros **recursos JAX-RS**, que también serán clases java anotadas, y que utilizarán objetos de nuestro dominio para llevar a cabo las operaciones expuestas en el API RESTful que hemos diseñado.

Así, por ejemplo, implementaremos las clases:

- **Cliente.java**: representa una entidad del dominio. Contiene atributos, y sus correspondientes *getters* y *setters*
- **ClienteResource.java**: representa las operaciones RESTful sobre nuestro recurso *Cliente* que hemos definido en esta sesión. Es una clase java con anotaciones JAX-RS que nos permitirá insertar, modificar, borrar, consultar un cliente, así como consultar la lista de clientes de nuestro sistema.

### Clases de nuestro dominio (entidades): Cliente.java

La clase que representa nuestra entidad del dominio **Cliente** es una clase java plana, con sus correspondientes atributos, y *getters* y *setters*.

#### Implementación del dominio: cliente.java

```
package org.expertojava.domain;

@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    private int id;
    private String nombre;
    private String apellidos;
    private String direccion;
```

```
    private String codPostal;
    private String ciudad;

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getNombre() { return nombre; }
    public void setNombre(String nom) { this.nombre = nom; }

    public String getApellidos() { return apellidos; }
    public void setApellidos(String apellidos) {
        this.apellidos = apellidos; }

    public String getDireccion() { return direccion; }
    public void setDireccion(String dir) { this.direccion = dir; }

    public String getCodPostal() { return codPostal; }
    public void setCodPostal(String cp) { this.codPostal = cp; }

    public String getCiudad() { return ciudad; }
    public void setCiudad(String ciudad) { this.ciudad = ciudad; }
}
```

Hemos anotado la clase *Cliente* con `@XmlRootElement` y `@XmlAccessorType`. Hablaremos de estas anotaciones en sesiones posteriores, las cuales se encargan del serializado/deserializado del cuerpo del mensaje (en formato xml o json) a nuestra clase java *Cliente*.

## Clases de nuestro servicio RESTful: ClienteResource.java

Una vez definido el objeto de nuestro dominio que representará un objeto *Cliente*, vamos a ver cómo implementar nuestros servicio JAX-RS para que diferentes usuarios, de forma remota, puedan interactuar con nuestra base de datos de clientes.

La implementación del servicio es lo que se denomina una **resource class**, que no es más que una clase java que utiliza anotaciones JAX-RS.

Por defecto, una nueva instancia de nuestra clase de recursos se crea para **cada petición** a ese recurso. Es lo que se conoce como un objeto **per-request**. Esto implica que se crea un objeto Java para procesar **cada** petición de entrada, y se "destruye" automáticamente cuando la petición se ha servido. *Per-request* también implica "sin estado", ya que no se guarda el estado del servicio entre peticiones.

Comencemos con la implementación del servicio:

```
package org.expertojava.services;

import ...;

@Path("/clientes")
public class ClienteResource {

    private static Map<Integer, Cliente> clienteDB =
            new ConcurrentHashMap<Integer, Cliente>();
```

```
    private static AtomicInteger idContador = new AtomicInteger();
```

Podemos observar que *ClientResource* es una clase java plana,y que no implementa ninguna interfaz JAX-RS particular. La anotación `javax.ws.rs.Path` indica que la clase *ClienteResource* es un servicio JAX-RS. Todas las clases que queramos que sean "reconocidas" como servicios JAX-RS tienen que tener esta anotación. Fíjate que esta anotación tiene el valor `/clientes`. Este valor representa la raíz relativa de la URI de nuestro servicio RESTful. Si la URI absoluta de nuestro servidor es, por ejemplo: *http://expertojava.org*, los métodos expuestos por nuestra clase *ClienteResource* estarían disponibles bajo la URI *http://expertojava.org/clientes*.

En nuestra clase, definimos un Mapa para el campo `ClienteDB`, que almacenará en memoria a los objetos *Cliente* de nuestro sistema. Utilizamos un *java.util.concurrent.ConcurrentHashMap* como tipo de *clienteDB* ya que nuestro recurso será accedido **concurrentemente** por los usuarios de nuestro servicio rest. El campo `idContador` lo utilizaremos para generar nuevos identificadores de nuestros objetos *Cliente* creados. El tipo de este campo es *java.util.concurrent.atomic.AtomicInteger* para garantizar que siempre generaremos un identificador único aunque tengamos peticiones concurrentes.

### Justificación del caracter static de los atributos

Como nuestros objetos serán de tipo **per-request**, el runtime de JAX-RS creará una instancia de *ClienteResource* **para cada peción** que se realice sobre nuestro servicio. La máquina virtual de java ejecutará cada petición a nuestro servicio en un hilo (*thread*) diferente, permitiendo así el acceso concurrente a nuestro recurso. Puesto que hemos decidido almacenar en memoria la información de los clientes, necesitamos que los atributos **clienteDB** y **idContador** sean **static**, para que todas las instancias de *ClienteResource* tengan acceso a la lista de clientes en memoria y no haya problemas de concurrencia. En realidad, lo que estamos haciendo con ésto es permitir que el servicio guarde el estado entre peticiones. En un sistema real, *ClienteResource* probablemente interactúe con una base de datos para almacenar y recuperar la información de los clientes, y por lo tanto, no necesitaremos guardar el estado entre peticiones.

Una mejor solución sería no utilizar variables estáticas, y definir nuestro servicio como **singleton**. Si hacemos ésto, solamente se crearía una instancia de *clienteResource* y estaríamos manteniendo el estado de las peticiones. En la siguiente sesión explicaremos cómo configurar un servicio como **singleton**. Por simplicidad, de momento optaremos por la opción de que los objetos RESTful sean **per-request**.

## Creación de clientes

Para implementar la creación de un nuevo cliente utilizamos el mismo modelo que hemos diseñado previamente. Una petición HTTP POST envía un documento XML que representa al cliente que queremos crear.

El código para crear nuevos clientes en nuestro sistema podría ser éste:

```
@POST ❶
@Consumes("application/xml") ❷
public Response crearCliente(Cliente cli) { ❸
```

```
  //el parámetro cli se instancia con los datos del cliente del body del
mensaje HTTP
  idContador++;
  cli.setId(idContador.incrementAndGet());
  clienteDB.put(cli.getId(), cli);   ❹
  System.out.println("Cliente creado " + cli.getId()); ❺
  return Response.created(URI.create("/clientes/"
                + cli.getId())).build();  ❻
    }
```

❶ se recibe una petición POST
❷ el cuerpo de la petición debe tener formato xml
❸ contiene la información del documento xml del cuerpo de la petición de entrada
❹ se añade el nuevo objeto *Cliente* a nuestro "mapa" de clientes (**clienteDB**)
❺ este método se ejectua en el servidor, por lo que el mensaje sólo será visible, por ejemplo, si consultamos los mensajes generados por el servidor durante la ejecución el método devuelve un código de respuesta `201 Created`, junto con una cabecera **Location** apuntando a la URI absoluta del cliente que acabamos de crear

Vamos a explicar la implementación con más detalle.

Para enlazar peticiones HTTP POST con el método *crearCliente()*, lo anotamos con la anotación `@javax.ws.rs.POST`. La anotación *@Path*, combinada con la anotación *@POST*, enlaza todas las peticiones POST dirigidas a la URI relativa */clientes* al método Java *crearCliente()*.

La anotación `javax.ws.rs.Consumes` aplicada a *crearCliente()* especifica qué *media type* espera el método en el cuerpo del mensaje HTTP de entrada. Si el cliente incluye en su petición POST un *media type* diferente de XML, se envía un código de error al cliente.

El método *crearCliente()* tiene un parámetro de tipo *Cliente*. En JAX-RS, **cualquier parámetro no anotado** con anotaciones JAX-RS se considera que es una **representación del cuerpo del mensaje** de la petición de entrada HTTP. Las anotaciones que hemos introducido en la clase *Cliente* de nuestro dominio, realizan el trabajo de "convertir" el documento xml contenido en el cuerpo de la petición htpp de entrada en una instancia de nuestra clase Cliente.

Solamente UNO de los parámetros del método Java puede representar el cuerpo del mensaje de la petición HTTP. Esto significa **que el resto de parámetros** deben anotarse con alguna anotación JAX-RS, que veremos más adelante.

El método *crearCliente()* devuelve una respuesta de tipo `javax.ws.rs.core.Response`. El método estático `Response.created()` crea un objeto *Response* que contiene un código de estado `201 Created`. También añade una cabecera **Location** con un valor similar a: **http://expertojava.org/clientes/123**, dependiendo del valor del valor de base de la raíz de la URI del servidor y el identificador generado para el objeto *Cliente* (en este caso se habría generado el identificador 123). Más adelante explicaremos con detalle el uso de la clase *Response*.

## Consulta de clientes

A continuación mostramos un posible código para **consultar** la información de **un cliente**:

```
@GET
```

```
@Path("{id}")
@Produces("application/xml")
public Cliente recuperarClienteId(@PathParam("id") int id) {
  final Cliente cli = clienteDB.get(id);
  if (cli == null) {
      throw new WebApplicationException(Response.Status.NOT_FOUND);
  }
  return new Cliente(cli.getId(), cli.getNombre(), cli.getApellidos(),
               cli.getDireccion(), cli.getCodPostal(), cli.getCiudad());
}
```

En este caso, anotamos el método `recuperarClienteId()` con la anotación `@javax.ws.rs.GET` para enlazar las operaciones HTTP GET con este método Java.

También anotamos *recuperarClienteId()* con la anotación `@javax.ws.rs.PRODUCES`. Esta anotación indica a JAX-RS que valor tiene la cabecera HTTP **Content-Type** en la respuesta proporcionada por la operación GET. En este caso, estamos indicando que será de tipo **application/xml**.

En la implementación del método utilizamos el parámetro `id` para consultar si existe un objeto *Cliente* en nuestro mapa **clienteDB**. Si dicho cliente no existe, lanzaremos la excepción `javax.ws.rs.WebApplictionException`. Esta excepción provocará que el código de respuesta HTTP sea `404 Not Found`, y significa que el recurso cliente requerido no existe. Discutiremos más adelante el tema del manejo de excepciones.

## Modificación de clientes

Vamos a mostrar cómo sería el código para modificar un cliente:

```
@PUT
@Path("{id}")
@Consumes("application/xml")
public void modificarCliente(@PathParam("id") int id,
                            Cliente nuevoCli) {

  Cliente actual = clienteDB.get(id);
  if (actual == null)
      throw new WebApplicationException(Response.Status.NOT_FOUND);

  actual.setNombre(nuevoCli.getNombre());
 actual.setApellidos(nuevoCli.getApellidos());
  actual.setDireccion(nuevoCli.getDireccion());
  actual.setCodPostal(nuevoCli.getCodPostal());
  actual.setCiudad(nuevoCli.getCiudad());
}
```

Anotamos el método **modificarCliente()** con `@javax.ws.rs.PUT` para enlazar las peticiones HTTP PUT a este método. Al igual que hemos hecho con **recuperarClienteId()**, el método **modificarCliente()** está anotado adicionalmente con `@Path`, de forma que podamos atender peticiones a través de las URIs **/clientes/{id}**.

El método **modificarCliente()** tiene dos parámetros. El primero es un parámetro `id` que representa el objeto *Cliente* que estamos modificando. Al igual que ocurría con el método **recuperarClienteId()**, utilizamos la anotación `@PathParam` para extraer el identificador a

partir de la URI de la petición de entrada. El segundo parámetro es un objeto `Cliente`, que representa el cuerpo del mensaje de entrada, ya que no tiene ninguna anotación JAX-RS.

El método intenta encontrar un objeto *Cliente* en nuestro mapa **clienteDB**. Si no existe, provocamos una **WebApplicationException** que enviará una respuesta al usuario con el código `404 Not Found`. Si el objeto *Cliente* existe, modificamos nuestro objeto *Cliente* existente con los nuevos valores que obtenemos de la petición de entrada.

## Construcción y despliegue del servicio

Una vez implementado nuestro servicio RESTful, necesitamos poner en marcha el proceso de construcción. El proceso de construcción compilará,…, empaquetará, …, y finalmente nos permitirá desplegar nuestro servicio en el servidor de aplicaciones.

Para poder, empaquetar nuestro servicio RESTful como un *war*, que se desplegará en el servidor de aplicaciones, vamos a incluir un "proveedor" de servicios JAX-RS, en el descriptor de despliegue de nuestra aplicación (fichero **web.xml**). En la siguiente sesión justificaremos la existencia de dicho "proveedor" (que será un *servlet*) y explicaremos el modelo de despliegue de los servicios JAX-RS. Los pasos a seguir desde IntelliJ para **configurar el despliegue** de nuestro servicio son:

- Añadimos el directorio WEB-INF como subdirectorio de webapp
- Nos vamos a File#Project Structure…#Facets#Web, y añadimos el fichero *web.xml* (en el panel *Deployment descriptors*, pulsamos sobre +, y seleccionamos *web.xml*). Editamos este fichero para añadir el servlet que servirá las peticiones de nuestros servicios REST, indicando cuál será la ruta en la que estarán disponibles dichos servicios (en nuestro ejemplo indicaremos la ruta "/rest/"). Dicha ruta es relativa a la ruta del contexto de nuestra aplicación, y que por defecto, es el nombre del artefacto ".war" desplegado, que hemos indicado en la etiqueta <finalName> dentro del <build> del fichero de configuración de Maven (pom.xml).

**Contenido del fichero *web.xml*:**

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <!-- One of the way of activating REST Services is adding these lines,
    the server is responsible for adding the corresponding servlet
 automatically,
    if the src folder has the Annotations to receive REST invocation-->
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

A continuación ya estamos en disposición de iniciar la construcción del proyecto con Maven para compilar, empaquetar y desplegar nuestro servicio en Wildfly.

**Si utilizamos el terminal**, la secuencia de pasos para empaquetar y desplegar nuestro proyecto serían:

```
cd ejemplo-rest ❶
```

```
mvn package ❷
./usr/local/wildfly-8.2.1.Final/bin/standalone.sh ❸
mvn wildfly:deploy ❹
```

❶ Nos situamos en el directorio que contiene el pom.xml de nuestro proyecto
❷ Empaquetamos el proyecto (obtendremos el .war)
❸ Arrancamos el servidor wildfly
❹ Desplegamos el war generado en el servidor wildfly

## Secuencia correcta de acciones

En ejecuciones posteriores, después de realizar modificaciones en nuestro código, es recomendable ejecutar "mvn clean" previamente al empaquetado del proyecto.

Por lo tanto, y suponiendo que el servidor de aplicaciones **ya está en marcha**, la secuencia de acciones (comandos maven) que deberíamos realizar para asegurarnos de que vamos a ejecutar exactamente la aplicación con los últimos cambios que hayamos introducido son:

- mvn wildfly:undeploy
- mvn clean
- mvn package
- mvn wildfly:deploy

También podríamos realizar todas estas acciones con un único comando maven:

- mvn wildfly:undeploy clean package wildfly:deploy

**Si utilizamos IntelliJ**, añadiremos un nuevo elemento de configuración de ejecución desde Run#Edit Configurations. Pulsamos el icono `+` y añadimos la configuración de tipo JBosss Server#Local. Podemos ponerle por ejemplo como nombre "Wilfdly start". A continuación configuramos la ruta del servidor wildfly como: /usr/local/wildfly-8.2.1.Final.

Cuando lancemos este elemento de ejecución desde IntelliJ, automáticamente se construirá el proyecto (obtendremos el war), y arrancaremos wildfly. Para desplegar el war, utlizaremos la ventana *Maven Projects* y haremos doble click sobre ejemplo-rest#Plugins#wildfly#wildfly:deploy

## Probando nuestro servicio

Podemos probar nuestro servicio de varias formas. Vamos a mostrar como hacerlo directamente desde línea de comandos, utilizando IntelliJ, o bien utilizando la herramienta Postman (que tenéis disponible desde el navegador Chrome).

**Invocación del servicio desde línea de comandos**

Utilizaremos la herramienta *curl*. Por ejemplo, para realizar una inserción de un cliente, el comando sería:

```
curl -i -H "Accept: application/xml" -H "Content-Type: application/
xml" -X POST -d @cliente.xml http://localhost:8080/ejemplo-rest/rest/
clientes/
```

En donde:

**-i**
: También se puede utilizar la opción equivalente **--include**. Indica que se debe incluir las cabeceras HTTP en la respuesta recibida. Recuerda que la petición POST devuelve en la cabecera **Location** el enlace del nuevo recurso creado (puede hacerlo en una cabedera **Location**, o como un campo <*link*> del elemento creado en el cuerpo del mensaje, lo veremos más adelante). Esta información será necesaria para poder consultar la información del nuevo cliente creado.

**-H**
: Indica un par *cabecera*:_valor_. En nuestro caso lo utilizamos para especificar los valores de las cabeceras HTTP **Accept** y **Content-Type**

**-X**
: Indica el método a invocar (GET, POST, PUT,…)

**-d**
: También se puede utilizar **--data**. Indica cuáles son los datos enviados en el mensaje de entrada en una petición POST. Si los datos especificados van precedidos por `@`, estamos indicando que dichos datos están en un fichero. Por ejemplo, en la orden anterior, escribimos en el fichero **cliente.xml** los datos del cliente que queremos añadir en nuestro sistema.

El contenido del fichero cliente.xml podría ser éste:

```
<?xml version="1.0" encoding="UTF-8"?>
<clientes>
  <cliente>
    <nombre>Pepe </nombre>
    <apellidos>Garcia Lopez</apellido1>
    <direccion>Calle del pino, 3</apellido2>
    <codPostal>0001</codPostal>
    <ciudad>Alicante</ciudad>
  </cliente>
</clientes>
```

Finalmente, en la orden indicamos la URI a la que queremos acceder, en este caso:

```
http://localhost:8080/ejemplo-rest/rest/clientes/
```

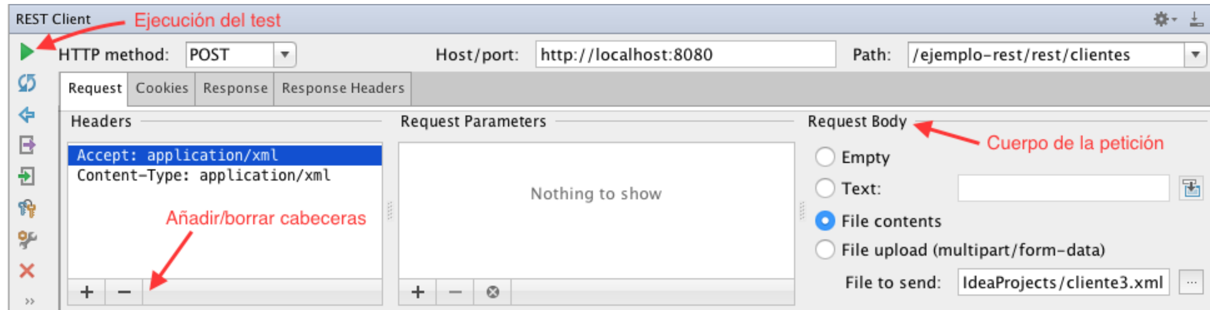
Una vez insertado el cliente, podemos recuperar el cliente, utilizando el enlace que se incluye en la cabecera de respuesta **Location**

```
curl -i -H "Accept: application/xml" -H "Content-Type: application/xml" -X
 GET http://localhost:8080/ejemplo-rest/rest/clientes/1
```

**Invocación del servicio desde IntelliJ**

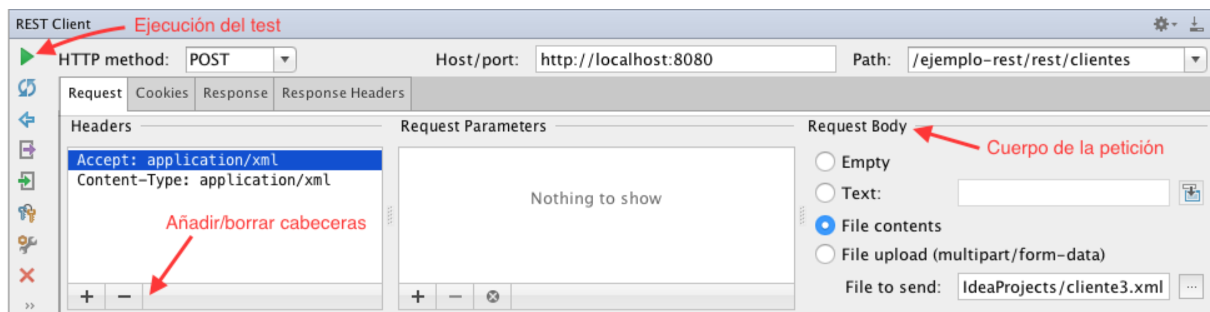
IntelliJ nos proporciona una herramienta para probar servicios REST, desde Tools#Test RESTful Web Service Desde esta nueva ventana podremos invocar al servicio REST indicando el tipo de petición HTTP, así como las cabeceras y cuerpo de la petición.

La siguiente figura muestra la elaboración de una petición POST a nuestro servicio REST:




A continuación mostramos la ejecución de una petición GET:

Cuando realizamos una petición POST debemos indicar el contenido del cuerpo del mensaje. En la siguiente figura observamos que tenemos varias opciones disponibles, como por ejemplo "teclear" directamente dicho contenido (opción *Text*), o bien "subir" dicha información desde un fichero en nuestro disco duro (opción *File Contents*). Podemos ver que hemos elegido esta última opción para probar nuestro servicio.

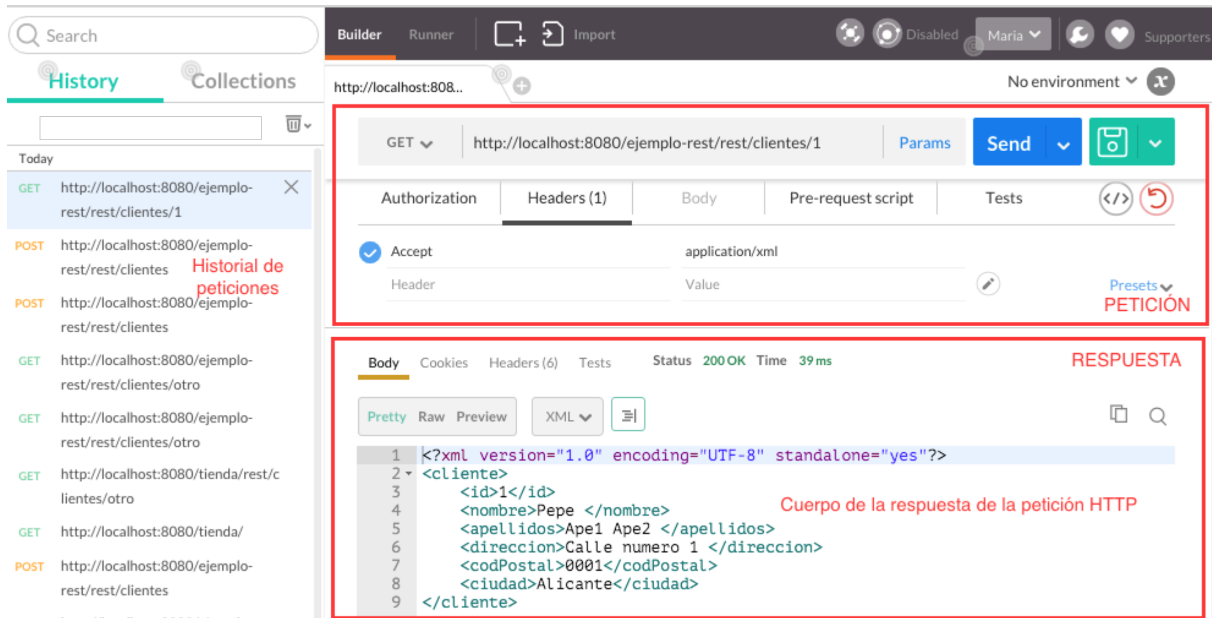



**Invocación del servicio desde Postman**

Otra alternativa sencilla para probar nuestro servicio REST es la herramienta postman, que podemos lanzar desde el navegador, en nuestro caso: Chrome.

Accederemos a la aplicación desde la barra de marcadores, seleccionando "Aplicaciones", y y a continuación, pulsaremos sobre el icono "Postman".

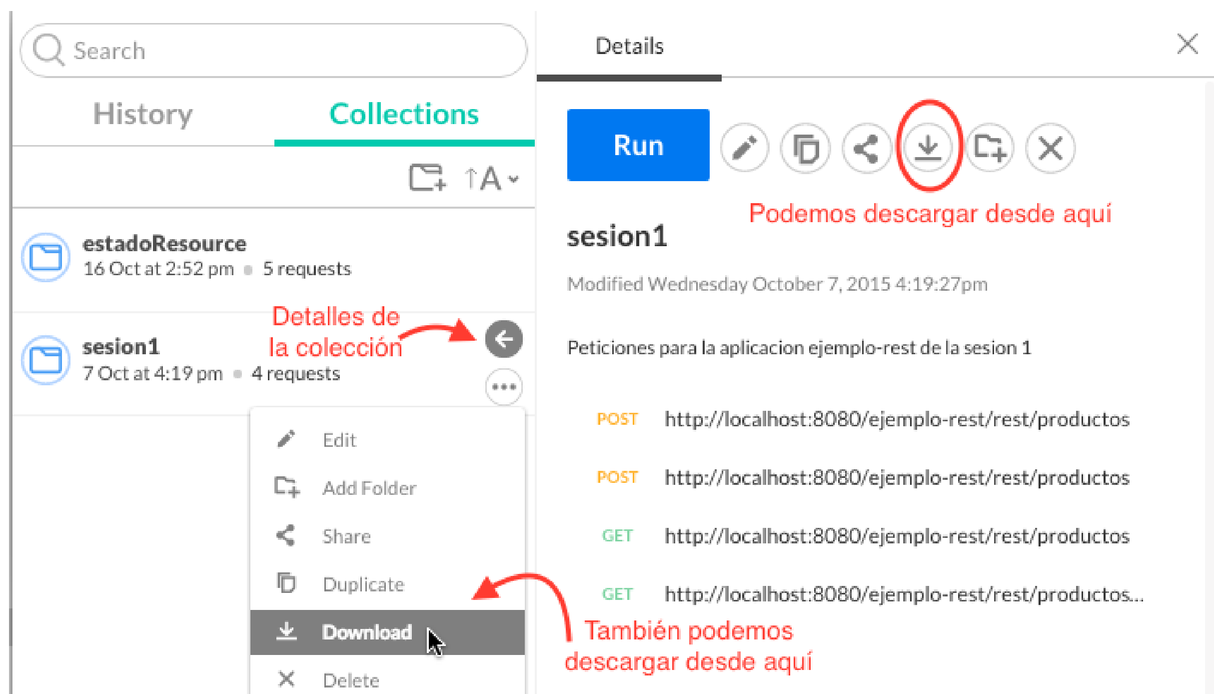
El aspecto de la herramienta es el que mostramos a continuación:




Postman, a diferencia de las alternativas anteriores, nos permitirá guardar un "historial" de peticiones, de forma que podamos repetir la ejecución de nuestros tests, exactamente de la misma forma, aunque no de forma automática, sino que tenemos que lanzar "manualmente" cada test que queramos volver a ejecutar.

También podemos crear "colecciones", que no son más que carpetas que contienen un conjunto de peticiones de nuestro "historial". Por ejemplo, podemos crear la **colección** *s1-rest-ejercicio1* en donde guardaremos todas las peticiones que hayamos hecho sobre el ejercicio 1 de la primera sesión de rest.

Podéis crearos una cuenta gratuita para almacener y gestionar vuestras peticiones rest. Una vez que tengáis creadas varias colecciones, Postman nos permite "guardarlas" en nuestro disco duro en formato json.



## 1.5. Ejercicios

Antes de empezar a crear los proyectos, debes descargarte el repositorio *git* `java_ua/ejercicios-rest-expertojava` en el que vas a implementar los ejercicios relativos a la asignatura de *Servicios REST*. El proceso es el mismo que el seguido en sesiones anteriores:

1. Accedemos al repositorio y realizamos un *Fork* en nuestra cuenta personal (así podremos tener una copia con permisos de escritura)
2. Realizamos un *Clone* en nuestra máquina:

```
$ git clone https://bitbucket.org/<alumno>/ejercicios-rest-expertojava
```

De esta forma se crea en nuestro ordenador el directorio `ejercicios-rest-expertojava` y se descarga en él un proyecto IntelliJ en el que iremos añadiendo **MÓDULOS** para cada uno de los ejercicios. Contiene también el fichero `gitignore`, así como diferentes módulos con las plantillas que vayamos a necesitar para realizar los ejercicios.

A partir de este momento se puede trabajar con dicho proyecto y realizar *Commit* y *Push* cuando sea oportuno:

```
$ cd ejercicios-rest-expertojava
$ git add .
$ git commit -a -m "Mensaje de commit"
$ git push origin master
```

Los **MÓDULOS** IntelliJ que iremos añadiendo, tendrán todos el prefijo `sx-`, siendo *x* el número de la sesión correspondiente (por ejemplo *s1-ejercicio*, *s2-otroEjercicio*,…).

### Servicio REST ejemplo (0 puntos)

Para familiarizarnos con las peticiones http POST, PUT, GET, DELETE se proporciona el **MÓDULO s1-ejemplo-rest**, con la implementación de un servicio rest que podéis probar, bien desde línea de comandos con la utilidad **curl**, desde el navegador con la herramienta **postman** , o bien desde IntelliJ con el **cliente REST** incluido en el IDE.

En el directorio `src/main/resources` de dicho módulo tenéis un fichero de texto (*instrucciones.txt*) con las instrucciones para construir, desplegar y probar la aplicación de ejemplo.

### Servicio REST saludo (1 punto)

Vamos a implementar un primer servicio RESTful muy sencillo. Para ello seguiremos las siguientes indicaciones:

- Creamos un **MÓDULO** Maven con IntelliJ (desde el directorio `ejercicios-rest-expertojava`) con el arquetipo `webapp-javaee7`, tal y como hemos visto en los apuntes de la sesión. Las coordenadas del **artefacto Maven** serán:

  # *GroupId*: org.expertojava

  # *ArtifactId*: s1-saludo-rest

# *version*: 1.0-SNAPSHOT

- Configuramos el pom.mxl del proyecto para poder compilar, empaquetar y desplegar nuestro servicio. Consulta los apuntes para ver cuál debe ser el contenido de las etiquetas *<properties>*, *<dependencies>* y *<build>*.
- Creamos la carpeta WEB-INF y añadimos el fichero de configuración *web.xml* tal y como hemos visto en los apuntes (esto será necesario para configurar el despliegue). En este caso queremos *mapear* los servicios REST, contenidos en el paquete *org.expertojava*, al directorio */recursos* dentro de nuestro contexto (recuerda que el contexto de nuestra aplicación web vendrá dado por el valor de la etiqueta <finalName>, anidada dentro de <build>).
- Creamos un recurso de nombre `HolaMundoResource`, que se mapee a la dirección **/holamundo**. Implementar un método, de forma que al acceder a él por **GET** nos devuelva en texto plano (text/plain) el mensaje "Hola mundo!". Una vez desplegada la aplicación en el servidor WildFly, prueba el servicio mediante la utilidad Postman desde Chrome. Comprobar que la invocación:

```
GET http://localhost:8080/saludo-rest/holamundo
```

Devuelve como cuerpo del mensaje: "Hola mundo!"

- Vamos a añadir un segmento variable a la ruta. Implementa un método GET nuevo, de forma que si accedemos a /holamundo/nombre, añade el nombre indicado al saludo (separado por un espacio en blanco y seguido por "!!").

Una vez desplegada la aplicación en el servidor WildFly, prueba el servicio mediante la utilidad "Test RESTFul Web Service" de IntelliJ, o con Postman. Comprobar que la invocación:

```
GET http://localhost:8080/saludo-rest/holamundo/pepe
```

Devuelve como cuerpo del mensaje: "Hola mundo! pepe!!"

- Hacer que se pueda cambiar el saludo mediante un método PUT. El nuevo saludo llegará también como texto plano en el cuerpo de la petición, y posteriores invocaciones a los métodos GET utilizarán el nuevo saludo. Almacenaremos el nuevo saludo en una variable estática de nuestro recurso. ¿Qué pasa si no lo es? (lo hemos explicado en los apuntes, puedes hacer la prueba para ver qué ocurre si la variable no es estática).

Una vez desplegada la aplicación en el servidor WildFly, prueba el servicio con Postman, o bien mediante la utilidad "Test RESTFul Web Service" de IntelliJ. Realizar las siguientes invocaciones (en este orden):

```
PUT http://localhost:8080/saludo-rest/holamundo/
y en el cuerpo del mensaje: "Buenos días"
```

```
GET http://localhost:8080/saludo-rest/holamundo
GET http://localhost:8080/saludo-rest/holamundo/pepe
```

La segunda invocación debe devolver como cuerpo del mensaje: `"Buenos dias"` Al ejecutar la tercera invocación el cuerpo del mensaje de respuesta debería ser: `"Buenos dias Pepe!!"`

## Servicio REST foro (1 punto)

Vamos a implementar un servicio RESTful que contemple las cuatro operaciones básicas (GET, PUT, POST y DELETE). Se trata de un foro con en el que los usuarios pueden intervenir, de forma anónima, en diferentes conversaciones.

Primero debes crear un nuevo módulo Maven, configurar el pom.xml, así como el fichero `web.xml`, de la misma forma que hemos hecho en el ejercicio anterior, pero para este ejercicio:

- Las coordenadas del módulo Maven serán:
  - *GroupId*: org.expertojava
  - *ArtifactId*: s1-foro-rest
  - *version*: 1.0-SNAPSHOT
- Nuestros servicios REST estarán disponibles en la URI `http://localhost:8080/s1-foro-rest/`

El foro estará formado por diferentes mensajes. Por lo tanto el modelo del dominio de nuestra aplicación estará formado por la clase `Mensaje`, que contendrá un identificador, y una cadena de caracteres que representará el contenido del mensaje (recuerda que debes implementar los correspondientes *getters* y *setters*).

Por simplicidad, vamos a almacenar los mensajes de nuestro foro en memoria. Estos estarán disponibles desde la clase `DatosEnMemoria`, que contendrá la variable estática:

```
static Map<Integer, Mensaje> datos = new HashMap<Integer, Mensaje>();
```

Los servicios que proporcionará el foro estarán implementados en la clase `MensajeResource`. Se accederá a ellos través de la ruta relativa a la raíz de nuestros servicios: "/mensajes". Concretamente podremos realizar las siguientes operaciones:

- Añadir un nuevo mensaje al foro con la URI relativa a la raíz de nuestros servicios: `"/mensajes"`. El texto del mensaje estará en el cuerpo de la petición y el tipo MIME asociado será `text/plain` (contenido de la cabecera `Content-type` de la petición HTTP). Nuestra respuesta debe incluir en la cabecera `Location` de la respuesta HTTP, la URI del nuevo recurso creado. Utiliza para ello la clase `Response` tal y como hemos mostrado en el código de ejemplo proporcionado para el ejercicio anterior. Hablaremos con detalle sobre esta clase en sesiones posteriores.

  Recuerda que para acceder al cuerpo de la petición basta con definir un parámetro de tipo *String*. JAX-RS automáticamente lo instanciará a partir del cuerpo de la petición y lo convertirá en un objeto de tipo *String*.

- Modificar un mensaje determinado con un identificador con valor `id`, a través de la URI relativa a la raíz de nuestros servicios: `/mensajes/id` (`id` debe ser, por tanto, un segmento de ruta variable). Si no existe ningún mensaje con el identificador `id`, se lanzará la excepción: `WebApplicationException(Response.Status.NOT_FOUND)`

- Borrar un mensaje determinado con un identificador con valor `id`, a través de la URI relativa a la raíz de nuestros servicios: `/mensajes/id`. Igual que en el caso anterior, si el identificador proporcionado no se corresponde con el de ningún mensaje del foro, se lanzará la excepción: `WebApplicationException(Response.Status.NOT_FOUND)`
- Consultar todos los mensajes del foro (la URI relativa será: `/mensajes`). El resultado se mostrará en tantas líneas como mensajes. Cada mensaje irá precedido de su identificador. También se informará del número total de mensajes en el foro. (La respuesta será una cadena de caracteres. Al final del ejercicio mostramos un ejemplo de mensaje de respuesta para esta operación)
- Consultar un mensaje determinado con un identificador con valor `id`, a través de la URI relativa a la raíz de nuestros servicios: `/mensajes/id`. Si el identificador proporcionado no se corresponde con el de ningún mensaje del foro, se lanzará la excepción: `WebApplicationException(Response.Status.NOT_FOUND)`

Prueba el servicio utilizando Postman, o el cliente de IntelliJ para servicios REST, con las siguientes entradas:

- Crea los mensajes "Mensaje numero 1", "Mensaje numero 2", Mensaje numero 3", en este orden
- Consulta los mensajes del foro. El resultado debe ser:

  "1: Mensaje numero 1

  2: Mensaje numero 2

  3: Mensaje numero 3

  Numero total de mensajes = 3"
- Cambia el mensaje con identificador 2 por: "Nuevo mensaje numero 2"
- Consulta los mensajes del foro. El resultado debe ser:

  "1: Mensaje numero 1

  2: Nuevo Mensaje numero 2

  3: Mensaje numero 3

  Numero total de mensajes = 3"
- Borra el mensaje con identificador 3
- Consulta el mensaje con el identificador 3. Se debe obtener una respuesta `404 Not Found`
- Consulta los mensajes del foro. El resultado debe ser:

  "1: Mensaje numero 1

  2: Nuevo Mensaje numero 2

  Numero total de mensajes = 2"
- Añade el mensaje "Mensaje final". Vuelve a consultar los mensajes, el resultado debe ser:

  "1: Mensaje numero 1

2: Nuevo Mensaje numero 2

4: Mensaje final

Numero total de mensajes = 3"

Para evitar problemas con el *id* generado si hemos borrado mensajes, lo más sencillo es que el identificador vaya incrementándose siempre con cada nuevo mensaje. Esto puede hacer que "queden huecos" en la numeración, como en el ejemplo anterior.

# 2. Anotaciones básicas JAX-RS. El modelo de despliegue.

Ya hemos visto como crear un servicio REST básico. Ahora se trata de analizar con más detalle aspectos fundamentales sobre la implementación de los servicios. Comenzaremos por detallar los usos de la anotación @Path, que es la que nos permite "etiquetar" una clase Java como un recurso REST sobre el que podremos realizar las operaciones que hemos identificado en la sesión anterior. También hablaremos algo más sobre las anotaciones @Produces y @Consumes que ya hemos utilizado para implementar nuestro primer servicio.

En segundo lugar hablaremos sobre la extracción de información de las peticiones HTTP, y cómo podemos inyectar esa información en nuestro código java. Esto nos permitirá servir las peticiones sin tener que escribir demasiado código adicional.

Finalmente, explicaremos más detenidamente cómo configurar el despliegue de nuestra aplicación REST, de forma que sea portable.

## 2.1. ¿Cómo funciona el enlazado de métodos HTTP?

JAX-RS define cinco anotaciones que se corresponden con operaciones HTTP específicas:

- @javax.ws.rs.GET
- @javax.ws.rs.PUT
- @javax.ws.rs.POST
- @javax.ws.rs.DELETE
- @javax.ws.rs.HEAD

En la sesión anterior ya hemos utilizado estas anotaciones para hacer corresponder (enlazar) peticiones HTTP GET con un método Java concreto:

Por ejemplo:

```
@Path("/clientes")
public class ServicioCliente {

  @GET @Produces("application/xml")
  public String getTodosLosClientes() { }
}
```

En este código, la anotación @GET indica al *runtime* JAX-RS que el método java *getTodosLosClientes()* atiende peticiones HTTP GET dirigidas a la URI */clientes*

Sólamente se puede utilizar una de las anotaciones anteriores para un mismo método. Si se aplica más de uno, se produce un error durante el despliegue de la aplicación

Es interesante conocer que cada una de estas anotaciones, a su vez, está anotada con otras anotaciones (podríamos llamarlas *meta anotaciones*). Por ejemplo, la implementación de la anotación @GET tiene este aspecto:

```
package javax.ws.rs;
import ...;
```

```
@Target({ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@HttpMethod(HttpMethod.GET)
public @interface GET {
}
```

@GET, en sí mismo, no tiene ningún significado especial para el proveedor JAX-RS (*runtime* de JAX-RS). Lo que hace que la anotación @GET sea significativo para el *runtime* de JAX-RS es el valor de la *meta anotación* @javax.ws.rs.HttpMethod (en este caso `HttpMethod.GET`). Este valor es el que realmente "decide" que un determinado método Java se "enlace" con un determinado método HTTP.

¿Cuáles son las implicaciones de ésto? Pues que podemos crear nuevas anotaciones que podemos enlazar a otros métodos HTTP que no sean GET, POST, PUT, DELETE, o HEAD. De esta forma podríamos permitir que diferentes tipos de clientes que hacen uso de la operación HTTP LOCK, puedan ser "atendidos" por nuestro servicio REST (como por ejemplo un cliente WebDAV ).

## 2.2. La anotación @Path

La anotación @Path identifica la "plantilla" de *path* para la URI del recurso al que se accede y se puede especificar a nivel de clase o a nivel de método de dicho recurso.

El **valor** de una anotación @Path es una **expresión** que denota una URI relativa a la URI base del servidor en el que se despliega el recurso, a la raiz del contexto de la aplicación, y al patrón URL al que responde el runtime de JAX-RS.

Un **segmento** de la URI es cada una de las subcadenas delimitadas por `/` que aparecen en dicha URI. Por ejemplo, la URI *http://ejemplo.clientes.com/clientes/vip/recientes* contiene 4 segmentos de ruta: *ejemplo.clientes.com*, *clientes*, *vip* y *recientes*.

La anotación @Path no es necesario que contenga una ruta que empiece o termine con el carácter `/`. El *runtime* de JAX-RS analiza igualmente la expresión indicada como valor de @Path

Para que una **clase Java** sea identificada como una clase que puede atender peticiones HTTP, ésta tiene que estar anotada con al menos la expresión: `@Path("/")`. Este tipo de clases se denominan **recursos JAX-RS raíz**.

Para recibir una petición, un **método Java** debe tener al menos una anotación de método HTTP, como por ejemplo @javax.ws.rs.GET. Este método no requiere tener ninguna anotación @Path adicional. Por ejemplo:

```
@Path("/pedidos")
public class PedidoResource {
  @GET
  public String getTodosLosPedidos() {
    ...
  }
}
```

Una petición HTTP **GET /pedidos** se delegará en el método **getTodosLosPedidos()**.

Podemos aplicar también @Path a un método Java. Si hacemos esto, la expresión de la anotación @Path de la clase, se concatenará con la expresión de la anotación @Path del método. Por ejemplo:

```
@Path("/pedidos")
public class PedidoResource {

  @GET
  @Path("noPagados")
  public String getPedidosNoPagados() {
    ...
  }
}
```

De esta forma, una petición **GET /pedidos/noPagados** se delegará en el método **getPedidosNoPagados()**.

Podemos tener anotaciones @Path para cada método, que serán relativos a la ruta indicada en la anotación @Path de la definición de la clase. Por ejemplo, la siguiente clase de recurso sirve peticiones a la URI **/pedidos**:

```
@Path("/pedidos")
public class PedidoResource {

  @GET
  public String getPedidos() {
    ...
  }
}
```

Si quisiéramos proporcionar el servicio en la URI **pedidos/incidencias**, por ejemplo, no necesitamos una nueva definición de clase, y podríamos anotar un nuevo método *getIncidenciasPedidos()* de la siguiente forma:

```
@Path("/pedidos")
public class PedidoResource {

  @GET
  public String getPedidos() {...}

  @GET
  @Path("/incidencias")
  public String getIncidenciasPedidos() {...}
}
```

Ahora tenemos una clase de recurso que gestiona peticiones para **/pedidos**, y para **/pedidos/incidencias/**.

## Expresiones @Path

El valor de una anotación @Path puede ser una cadena de caracteres, o también puede contener expresiones más complejas si es necesario, nos referiremos a ellas como **expresiones @Path**

Una expresión @Path puede incluir variables, que se indican entre llaves, que serán sustituidas en tiempo de ejecución dependiendo del valor que se indique en la llamada al recurso. Así, por ejemplo, si tenemos la siguiente anotación:

```
@GET
@Path("/clientes/{id}")
```

y el usuario realiza la llamada:

```
GET http://org.expertojava/contexto/rest/clientes/Pedro
```

la petición se delegará en el método que esté anotado con las anotaciones anteriores y el valor de {id} será instanciado en tiempo de ejecución a "Pedro".

Para obtener el valor del nombre del cliente, utilizaremos la anotación `@PathParam` en los parámetros del método, de la siguiente forma:

```
@GET
@Path("/clientes/{nombre}")
public String getClientePorNombre(@PathParam("nombre") String nombre) {
    ...
}
```

Una expresión @Path puede tener más de una variable, cada una figurará entre llaves. Por ejemplo, si utilizamos la siguiente expresión @Path:

```
@Path("/{nombre1}/{nombre2}/")
public class MiResource {
  ...
}
```

podremos atender peticiones dirigidas a URIs que respondan a la plantilla:

```
http://org.expertojava/contexto/recursos/{nombre1}/{nombre2}
```

como por ejemplo:

```
http://org.expertojava/contexto/recursos/Pedro/Lopez
```

Las expresiones @Path pueden incluir más de una variable para referenciar un segmento de ruta. Por ejemplo:

```
@Path("/")
public class ClienteResource {
  @GET
  @Path("clientes/{apellido1}-{apellido2}")
  public String getCliente(@PathParam("apellido1") String ape1,
                           @PathParam("apellido2") String ape2) {
```

```
    ...
  }
}
```

Una petición del tipo:

```
GET http://org.expertojava/contexto/clientes/Pedro-Lopez
```

será procesada por el método **getCliente()**

## Expresiones regulares

Las anotaciones @Path pueden contener expresiones regulares (asociadas a las variables). Por ejemplo, si nuestro método **getClienteId()** tiene un parámetro de tipo entero, podemos restringir las peticiones para tratar solamente aquellas URIs que contengan dígitos en el segmento de ruta que nos interese:

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Path("{id : \\d+}")  //solo soporta dígitos
  public String getClienteId(@PathParam("id") int id) {
   ...
  }
}
```

Si la URI de la petición de entrada no satisface ninguna expresión regular de ninguno de los metodos del recurso, entonces se devolverá el código de error: **404 Not Found**

El formato para especificar expresiones regulares para las variables del *path* es:

```
{" nombre-variable [ ":" expresion-regular ] "}
```

El uso de expresiones regulares es opcional. Si no se proporciona una expresión regular, por defecto se admite cualquier carácter. En términos de una expresión regular, la expresión regular por defecto sería:

```
"[^/]+?"
```

Por ejemplo, si queremos aceptar solamente nombres que comiencen por una letra, y a continuación puedan contener una letra o un dígito, lo expresaríamos como:

```
@Path("/clientes")
public class ClienteResource {
   @GET
   @Path("{nombre : [a-zA-Z][a-zA-Z_0-9]}")
   public String getClienteNombre(@PathParam("nombre") string nom) {
    ...
   }
}
```

De esta forma, la URI `/clientes/aaa` no sería válida, la URI `/clientes/a9` activaría el método *getClienteNombre()*, y la URI `/clientes/89` activaría el método *getClienteId()*.

Las expresiones regulares no se limitan a un sólo segmento de la URI. Por ejemplo:

```
@Path("/clientes")
public class ClienteResource {
   @GET
   @Path("{id : .+}")
   public String getCliente(@PathParam("id") String id) {
   ...
   }

   @GET
   @Path("{id : .+}/direccion")
   public String getDireccion(@PathParam("id") String id) {
   ...
   }
}
```

La expresión regular `.+` indica que están permitidos cualquier número de caracteres. Así, por ejemplo, la petición `GET /clientes/pedro/lopez` podría delegarse en el método *getClientes()*

El método *getDireccion()* tiene asociada una expresión más específica, la cual puede mapearse con cualquier cadena de caracteres que termine con */direccion*. Según ésto, la petición `GET /clientes/pedro/lopez/direccion` podría delegarse en el método *getDireccion()*.

### Reglas de precedencia

En el ejemplo anterior, acabamos de ver que las expresiones @Path para *getCliente()* y *getDireccion()* son ambiguas. Una petición `GET /clientes/pedro/lopez/direccion` podría mapearse con cualquiera de los dos métodos. La especificación JAX-RS define las siguientes reglas para priorizar el mapeado de expresiones regulares:

- El primer criterio para ordenar las acciones de mapeado es el número de caracteres literales que contiene la expresión @Path, teniendo prioridad aquellas con un mayor número de caracteres literales. El patrón de la URI para el método *getCliente()* tiene 10 carácteres literales: `/clientes`. El patrón para el método *getDireccion()* tiene 19: `clientes/direccion`. Por lo tanto se elegiría primero el método *getDireccion()*
- El segundo criterio es el número de variables en expresiones @Path (por ejemplo {id}, o {id: .+}). Teniendo precedencia las patrones con un mayor número de variables
- El tercer criterio es el número de variables que tienen asociadas expresiones regulares (también en orden descendente)

A continuación mostramos una lista de expresiones @Path, ordenadas en orden descendente de prioridad:

1. /clientes/{id}/{nombre}/direccion
2. /clientes/{id : .+}/direccion
3. /clientes/{id}/direccion
4. /clientes/{id : .+}

Las expresiones 1..3 se analizarían primero ya que tienen más caracteres literales que la expresión número 4. Si bien las expresiones 1..3 tienen el mismo número de caracteres literales. La expresión 1 se analizaría antes que las otras dos debido a la segunda regla (tiene más variables). Las expresiones 2 y 3 tienen el mismo número de caracteres literales y el mismo número de variables, pero la expresión 2 tiene una variable con una expresión regular asociada.

Estas reglas de ordenación no son perfectas. Es posible que siga habiendo ambigüedades, pero cubren el 90% de los casos. Si el diseño de nuestra aplicación presenta ambigüedades aplicando estas reglas, es bastante probable que hayamos complicado dicho diseño y sería conveniente revisarlo y refactorizar nuestro esquema de URIs.

## Parámetros *matrix* (*Matrix parameters*)

Los parámetros *matrix* con pares nombre-valor incluidos como parte de la URI. Aparecen al final de un **segmento** de la URI (segmento de ruta) y están delimitados por el carácter `;`. Por ejemplo:

```
http://ejemplo.coches.com/seat/ibiza;color=black/2006
```

En la ruta anterior el parámetro *matrix* aparece después del segmento de ruta *ibiza*. Su nombre es `color` y el `valor` asociado es `black`.

Un parámetro *matrix* es diferente de lo que denominamos **parámetro de consulta** (*query parameter*), ya que los parámetros *matrix* representan atributos de ciertos **segmentos** de la URI y se utilizan para propósitos de **identificación**. Pensemos en ellos como adjetivos. Los **parámetros de consulta**, por otro lado, **siempre** aparecen al **final** de la URI, y **siempre** pertenecen al recurso "completo" que estemos referenciando.

Los parámetros *matrix* son ignorados cuando el runtime de JAX-RS realiza el *matching* de las peticiones de entrada a métodos de recursos REST. De hecho, es "ilegal" incluir parámetros *matrix* en las expresiones @Path. Por ejemplo:

```
@Path("/seat")
public class SeatService {

   @GET
   @Path("/ibiza/{anyo}")
   @Produces("image/jpeg")
   public Response getIbizaImagen(@PathParam("anyo") String anyo) {
   ... }
}
```

Si la petición de entrada es: **GET /seat/ibiza;color=black/2009**, el método *getIbizaImagen()* sería elegido por el proveedor de JAX-RS para servir la petición de entrada, y sería invocado. Los parámetros *matrix* **NO** se consideran parte del proceso de *matching* debido a que normalmente son atributos variables de la petición.

## Subrecursos (*Subresource Locators*)

Acabamos de ver la capacidad de JAX-RS para hacer corresponder, de forma **estática** a través de la anotación @Path, URIs especificadas en la entrada de la petición con métodos Java específicos. JAX-RS también nos permitirá, de forma **dinámica** servir nosotros

mismos las peticiones a través de los denominados **subresource locators** (localizadores de subrecursos).

Los **subresource locators** son métodos Java anotados con @Path, pero sin anotaciones @GET, @PUT, ... Este tipo de métodos devuelven un objeto, que es, en sí mismo, un servicio JAX-RS que "sabe" cómo servir el resto de la petición. Vamos a describir mejor este concepto con un ejemplo.

Supongamos que queremos extender nuestro servicio que proporciona información sobre los clientes. Disponemos de diferentes bases de datos de clientes según regiones geográficas. Queremos añadir esta información en nuestro esquema de URIs pero desacoplando la búsqueda del servidor de base de datos, de la consulta particular de un cliente en concreto. Añadiremos la información de la zona geográfica en la siguiente expresión @Path:

```
/clientes/{zona}-db/{clienteId}
```

A continuación definimos la clase **ZonasClienteResource**, que delegará en la clase ClienteResource, que ya teníamos definida.

```
@Path("/clientes")
public class ZonasClienteResource {

  @Path("{zona}-db")
  public ClienteResource getBaseDeDatos(@PathParam("zona") String db) {
      // devuelve una instancia dependiendo del parámetro db
      ClienteResource resource = localizaClienteResource(db);
      return resource;
  }

  protected ClienteResource localizaClienteResource(String db) {
  ...
  }

}
```

La clase **ZonasClienteResource** es nuestro recurso raíz. Dicha clase no atiende ninguna petición HTTP directamente. Nuestro recurso raíz procesa el segmento de URI que hace referencia a la base de datos en donde buscar a nuestro cliente y devuelve una instancia de dicha base de datos (o más propiamente dicho, del objeto con en que accederemos a dicha base de datos). El "proveedor" de JAX-RS utiliza dicha instancia para "servir" el resto de la petición:

```
public class ClienteResource {
  private Map<Integer, Cliente> clienteDB =
                 new ConcurrentHashMap<Integer, Cliente>();
  private AtomicInteger idContador = new AtomicInteger();

  public ClienteResource(Map<Integer, Cliente> clienteDB) {
      this.clienteDB = clienteDB;
  }

  @POST
  @Consumes("application/xml")
```

```
    public Response crearCliente(InputStream is) { ... }

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public Cliente recuperarClienteId(@PathParam("id") int id) { ... }

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void modificarCliente(@PathParam("id") int id, Cliente cli)
   { ... }
  }
```

Si un usuario envía la petición *GET /clientes/norteamerica-db/333*, el proveedor JAX-RS primero realizará un *matching* de la expresión sobre el método *ZonasClienteResource.getBaseDeDatos()*. A continuación procesará el resto de la petición ("/333") a través del método ClienteResource.recuperarClienteId().

Podemos observar que la nueva clase **ClienteResource**, además de tener un nuevo constructor, ya no está anotada con @Path. Esto implica que **ya no es un recurso** de nuestro sistema; es un **subrecurso** y no debe ser registrada en el *runtime* de JAX-RS a través de la clase *Application* (como veremos más adelante).

Veamos otro ejemplo. Supongamos que tenemos un conjunto de alumnos, del que podemos obtener el listado completo de alumnos y añadir nuevos alumnos, pero además queremos que cada alumno individual pueda consultarse, modificarse o borrarse. Una forma sencilla de tratar esto es dividir el código en un **recurso** (lista de alumnos) y un **subrecurso** (alumno individual) de la siguiente forma:

```
@Path("/alumnos")
public class AlumnosResource {

    @Context
    UriInfo uriInfo;

    @GET
    @Produces({MediaType.APPLICATION_XML,
               MediaType.APPLICATION_JSON})
    public List<AlumnoBean> getAlumnos() {
        return FactoriaDaos.getAlumnoDao().getAlumnos();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void addAlumno(AlumnoBean alumno) throws IOException {
        String dni = FactoriaDaos.getAlumnoDao().addAlumno(alumno);
        URI uri =
          uriInfo.getAbsolutePathBuilder().path("{dni}").build(dni);
        Response.created(uri).build();
    }

    @Path("{alumno}")
    public AlumnoResource getAlumno(
                             @PathParam("alumno") String dni) {
```

```
        return new AlumnoResource(uriInfo, dni);
    }

}
```

Vemos que en este recurso inyectamos información sobre la URI solicitada como variable de instancia (utilizando la anotación `@Context`, de la que hablaremos más adelante). Para el conjunto de alumnos ofrecemos dos operaciones: obtener la lista de alumnos, y añadir un nuevo alumno a la lista, la cual devuelve como respuesta la URI que nos da acceso al recurso que acabamos de añadir.

Sin embargo, lo más destacable es el último método. Éste se ejecutará cuando añadamos a la ruta el identificador de un alumno (por ejemplo `/alumnos/15`). En este caso lo que hace es devolver un subrecurso (**AlumnoResource**), para así tratar un alumno individual (destacamos que el nombre está en singular, para distinguirlo del recurso anterior que representa el conjunto).

Cuando hacemos esto estamos delegando en el nuevo *Recurso* para tratar la petición.

```
public class AlumnoResource {

    UriInfo uriInfo;

    String dni;

    public AlumnoResource(UriInfo uriInfo, String dni) {
        this.uriInfo = uriInfo;
        this.dni = dni;
    }

    @GET
    @Produces({MediaType.APPLICATION_XML,MediaType.APPLICATION_JSON})
    public AlumnoBean getAlumno() {
        AlumnoBean alumno =
            FactoriaDaos.getAlumnoDao().getAlumno(dni);
        if(alumno==null)
            throw new WebApplicationException(Status.NOT_FOUND);
        return alumno;
    }

    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public Response setAlumno(AlumnoBean alumno) {
        // El DNI del alumno debe coincidir con el de la URI
        alumno.setDni(dni);

        if(FactoriaDaos.getAlumnoDao().getAlumno(dni) != null) {
            FactoriaDaos.getAlumnoDao().updateAlumno(alumno);
            return Response.noContent().build();
        } else {
            FactoriaDaos.getAlumnoDao().addAlumno(alumno);
            return
                Response.created(uriInfo.getAbsolutePath()).build();
        }
    }
```

```
    @DELETE
    public void deleteAlumno() {
        FactoriaDaos.getAlumnoDao().deleteAlumno(dni);
    }
}
```

Este recurso ya no es un recurso raíz mapeado a una ruta determinada (podemos ver que la clase no lleva la anotación @Path), sino que es creado desde otro recurso. Es, por lo tanto, un subrecurso.

Como ya hemos visto, los subrecursos nos permiten simplificar la forma de trabajar con conjuntos de recursos, definiendo en un único método la ruta de acceso a un recurso individual, en lugar de tenerlo que hacer de forma independiente para cada operación.

Además, este diseño modular de los recursos nos va a permitir reutilizar determinados recursos dentro de otros. Por ejemplo, dentro del recurso de un alumno podríamos ver la lista de asignaturas en las que se ha matriculado, y reutilizar el subrecurso encargado de acceder a las asignaturas para poder acceder a sus datos a partir del recurso del alumno. No deberemos abusar de esta característica, ya que si creamos relaciones cíclicas perdemos la característica deseable de los servicios REST de que cada recurso está asignado a una única URI.

En un subrecurso **NO** podemos inyectar objetos de contexto mediante la anotación `@Context`, ya que no estamos en un recurso raíz. Por ese motivo en el ejemplo hemos proporcionado el objeto **UriInfo** en el constructor del subrecurso. De forma alternativa, también podríamos haber inyectado este objeto como parámetro de sus métodos, en ese caso si que habría sido posible la inyección.

## Carácter dinámico del "dispatching" de peticiones

En los ejemplos anteriores hemos ilustrado el concepto de *subresource locator*, aunque no hemos mostrado completamente su carácter dinámico. Así, si volvemos al primero de ellos, el método *ZonasClienteResource.getBaseDeDatos()* puede devolver **cualquier** instancia de **cualquier clase**. En tiempo de ejecución, el proveedor JAX-RS "buscará el interior" de esta instancia métodos de recurso que puedan gestionar la petición.

Supongamos que tenemos dos bases de datos de clientes con diferentes tipos de identificadores. Una de ellas utiliza una clave numérica. La otra utiliza una clave formada por el nombre y apellidos. Necesitamos tener dos clases diferentes para extraer la información adecuada de la URI de la petición. Cambiaremos la implementación de la siguiente forma:

```
@Path("/clientes")
public class ZonasClienteResourceResource {
   protected ClienteResource europa = new ClienteResource();
   protected OtraClaveClienteResource norteamerica =
                          new OtraClaveClienteResource();

   @Path("{zona}-db")
   public Object getBaseDeDatos(@PathParam("zona") String db) {
      if (db.equals("europa")) {
         return europa;
      }
      else if (db.equals("norteamerica")) {
               return northamerica; }
```

```
        else return null; }
}
```

En lugar de devolver una instancia de ClienteResource, el método getBaseDeDatos() devuelve una instancia de `java.lang.Object`. JAX-RS analizará la instancia devuelta para ver cómo procesar el resto de la petición.

Ahora, si un usuario envía la petición *GET /clientes/europa-db/333*, se utilizará la clase ClienteResource para servir el resto de la petición. Si la petición es *GET /clientes/norteamerica-db/john-smith* utilizaremos el nuevo subrecurso OtraClaveClienteResource:

```
public class OtraClaveClienteResource {
   private Map<String, Cliente> clienteDB =
               new ConcurrentHashMap<String, Cliente>();
   @GET
   @Path("{nombre}-{apellidos}")
   @Produces("application/xml")
   public Cliente getCliente(@PathParam("nombre") String nombre,
                             @PathParam("apellidos") String apelllidos) {
     ...
   }

   @PUT
   @Path("{nombre}-{apellidos}")
   @Consumes("application/xml")
   public void actualizaCliente()@PathParam("nombre") String nombre,
                             @PathParam("apellidos") String apelllidos,
                              Cliente cli) {
     ...
   }
}
```

## 2.3. Usos de las anotaciones @Produces y @Consumes

La información enviada a un recurso y posteriormente devuelta al cliente que realizó la petición se especifica con la cabecera HTTP `Media-Type`, tanto en la petición como en la respuesta. Como ya hemos visto, podemos especificar que representaciones de los recursos (valor de **Media_Type**) son capaces de aceptar y/o producir nuestros servicios mediante las siguientes anotaciones:

- javax.ws.rs.Consumes
- javax.ws.rs.Produces

La ausencia de dichas anotaciones es equivalente a incluirlas con el valor de *media type* `/`, es decir, su ausencia implica que se soporta (acepta) cualquier tipo de representación.

### Anotación @Consumes

Esta anotación funciona conjuntamente con @POST y @PUT. Le indica al framework (librerías JAX-RS) a qué método se debe delegar la petición de entrada. Específicamente, el cliente fija la cabecera HTTP `Content-Type` y el framework delega la petición al correspondiente método capaz de manejar dicho contenido. Un ejemplo de anotación con @PUT es la siguiente:

```
@Path("/pedidos")
public class PedidoResource {
   @PUT
   @Consumes("application/xml")
   public void modificarPedido(Pedido representation) { }
}
```

Si @Consumes se aplica a la clase, por defecto los métodos correspondientes aceptan los tipos especificados de tipo MIME. Si se aplica a nivel de método, se ignora cualquier anotación @Consumes a nivel de clase para dicho método.

En este ejemplo, le estamos indicando al framework que el método *modificarPedido()* acepta un recurso cuya representación (tipo MIME) es "application/xml" (y que se almacenará en la variable *representation*, hablaremos de ello en la siguiente sesión). Por lo tanto, un cliente que se conecte al servicio web a través de la URI */pedidos* debe enviar una petición HTTP PUT conteniendo el valor de `application/xml` como tipo MIME de la cabecera `HTTP Content-Type`, y el cuerpo (*body*) del mensaje HTTP debe ser, por tanto, un documento xml válido.

Si no hay métodos de recurso que puedan responder al tipo MIME solicitado (tipo MIME especificado en la anotación @Consumes del servicio), se le devolverá al cliente un código **HTTP 415 ("Unsupported Media Type")**. Si el método que consume la representación indicada como tipo MIME no devuelve ninguna representación, se enviará un el código **HTTP 204 ("No content")**. A continuación mostramos un ejemplo en el que sucede ésto:

```
@POST
@Consumes("application/xml")
public void creaPedido(Pedido pedido) {
    // Crea y almacena un nuevo _Pedido_
}
```

Podemos ver que el método "consume" una representación en texto plano, pero devuelve **void**, es decir, no devuelve ninguna representación. En este caso, se envía el código de estado **HTTP 204 No content** en la respuesta.

Un recurso puede aceptar diferentes tipos de "entradas". Así, podemos utilizar la anotación @PUT con más de un método para gestionar las repuestas con tipos MIME diferentes. Por ejemplo, podríamos tener un método para aceptar estructuras XML, y otro para aceptar estructuras JSON.

```
@Path("/pedidos")
public class PedidoResource {

   @PUT
   @Consumes("application/xml")
   public void modificarPedidoXML(InputStream pedido) { }

   @PUT
   @Consumes("application/json")
   public void modificarPedidoJson(InputStream pedido) { }

}
```

### Anotación @Produces

Esta anotación funciona conjuntamente con @GET, @POST y @PUT. Indica al *framework* qué tipo de representación se envía de vuelta al cliente.

De forma más específica, el cliente envía una petición HTTP junto con una cabecera HTTP **Accept** que se mapea directamente con el **Content-Type** que el método produce. Por lo tanto, si el valor de la cabecera Accept HTTP es **application/xml**, el método que gestiona la petición devuelve un stream de tipo MIME **application/xml**. Esta anotación también puede utilizarse en más de un método en la misma clase de recurso. Un ejemplo que devuelve representaciones XML y JSON sería el siguiente:

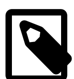
```
@Path("/pedidos")
public class PedidoResource {

   @GET
   @Produces("application/xml")
   public String getPedidoXml() { }

   @GET
   @Produces("application/json")
   public String getPedidoJson() { }
}
```

Si un cliente solicita una petición a una URI con un tipo MIME no soportado por el recurso, el *framework* JAX-RS lanza la excepción adecuada, concretamente el runtime de JAX-RS envía de vuelta un error **HTTP 406 Not acceptable**

Se puede declarar más de un tipo en la misma declaración @Produces, como por ejemplo:

```
@Produces({"application/xml", "application/json"})
public String getPedidosXmlOJson() {
    ...
}
```

El método *getPedidosXmlOJson()* será invocado si cualquiera de los dos tipos MIME especificados en la anotación @Produces son aceptables (la cabecera **Accept** de la petición HTTP indica qué representación es aceptable). Si ambas representaciones son igualmente aceptables, se elegirá la primera.



En lugar de especificar los tipos MIME como cadenas de texto en @Consumes y @Produces, podemos utilizar las constantes definidas en la clase **javax.ws.rs.core.MediaType**, como por ejemplo `MediaType.APPLICATION_XML` o `MediaType.APPLICATION_JSON`, en lugar de `application/xml` y `application/json`.

## 2.4. Inyección de parámetros JAX-RS

Buena parte del "trabajo" de JAX-RS es el "extraer" información de una petición HTTP e inyectarla en un método Java. Podemos estar interesados en un fragmento de la URI de

entrada, en los parámetros de petición,… El cliente también podría enviar información en las cabeceras de la petición. A continuación indicamos una lista con algunas de las anotaciones que podemos utilizar para inyectar información de las peticiones HTTP.

- @javax.ws.rs.PathParam
- @javax.ws.rs.MatrixParam
- @javax.ws.rs.QueryParam
- @javax.ws.rs.FormParam
- @javax.ws.rs.HeaderParam
- @javax.ws.rs.Context
- @javax.ws.rs.BeanParam

Habitualmente, estas anotaciones se utilizan en los parámetros de un método de recurso JAX-RX. Cuando el proveedor de JAX-RS recibe una petición HTTP, busca un método Java que pueda servir dicha petición. Si el método Java tiene parámetros anotados con alguna de estas anotaciones, extraerá la información de la petición HTTP y la "pasará" como un **parámetro** cuando se invoque el método.

## @javax.ws.rs.PathParam

Ya la hemos utilizado en la sesión anterior. @PathParam nos permite inyectar el valor de los parámetros de la URI definidos en expresiones @Path. Recordemos el ejemplo:

```
@Path("/clientes")
public class ClienteResource {
  ...

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Cliente recuperarClienteId(@PathParam("id") int id) {
    ...
  }
}
```

Podemos referenciar más de un parámetro en el path de la URI en nuestros método java. Por ejemplo, supongamos que estamos utilizando el nombre y apellidos para identificar a un cliente en nuestra clase de recurso:

```
@Path("/clientes")
public class ClienteResource {
  ...

  @GET
  @Path("{nombre}-{apellidos}")
  @Produces("application/xml")
  public Cliente recuperarClienteId(@PathParam("nombre") String nom,
                                    @PathParam("apellidos") String ape) {
    ...
  }
}
```

En ocasiones, un paramétro de path de la URI puede repetirse en diferentes expresiones @Path que conforman el patrón de *matching* completo para un método de un recurso (por ejemplo puede repetirse en la expresión @Path de la clase y de un método). En estos casos, la anotación @PathParam siempre referencia el parámetro path final. Así, en el siguiente código:

```
@Path("/clientes/{id}")
public class ClienteResource {
  ...

  @GET
  @Path("/direccion/{id}")
  @Produces("text/plain")
  public String getDireccion(@PathParam("id") String direccionId) {
    ...
  }
}
```

Si nuestra petición HTTP es: **GET /clientes/123/direccion/456**, el parámetro `direccionId` del método *getDireccion()* tendría el valor inyectado de **"456"**.

## Interfaz UriInfo

Podemos disponer, además, de un API más general para consultar y extraer información sobre las peticiones URI de entrada. Se trata de la interfaz `javax.ws.rs.core.UriInfo`:

```
public interface UriInfo {
  public java.net.URI getAbsolutePath();
  public UriBuilder getAbsolutePathBuilder();

  public java.net.URI getBaseUri();
  public UriBuilder getBaseUriBuilder();

  public String getPath();
  public List<PathSegment> getPathSegments();
  public MultivaluedMap<String, String> getPathParameters();
  ...
}
```

Los métodos **getAbsolutePathBuilder()** y **getAbsolutePath()** devuelven la ruta absoluta de la **petición** HTTP en forma de *UriBuilder* y *URI* respectivamente.

Los métodos **getBaseUri()** y **getBaseUriBuilder()** devuelven la ruta "base" de la aplicación (ruta raiz de nuestros servicios rest) en forma de *UriBuilder* y *URI* respectivamente.

El método **UriInfo.getPath()** permite obtener la ruta relativa de nuestros servicios REST utilizada para realizar el *matching* con nuestra petición de entrada (es la ruta de la petición actual relativa a la ruta base de la petición rest)

El método **UriInfo.getPathSegments()** "divide" la ruta relativa de nuestro servicio REST en una serie de objetos *PathSegment* (segmentos de ruta, delimitados por `/`).

El método **UriInfo.getPathParameters()** devuelve un objeto de tipo *MultivaluedMap* con todos los parámetros del path definidos en todas las expresiones @Path de nuestra petición rest.

Por ejemplo, si la ruta de nuestra petción http es: *http://localhost:8080/contexto/rest/clientes/2* (siendo "contexto" la ruta raíz del *war* desplegado, y "rest" la ruta de servicio de jax-rs):

- la ruta absoluta (método *getAbsolutePath()*) sería *http://localhost:8080/contexto/rest/clientes/2*
- la ruta base (método *getBaseUri*) sería *http://localhost:8080/contexto/rest/*
- la ruta relativa a la ruta base (método *getPath()*) sería */clientes/2*
- el número de segmentos de la petición rest (método *getPathSegments()*) serían 2: "clientes" y "2"

Podemos inyectar una instancia de la interfaz `UriInfo` utilizando la anotación `@javax.ws.rs.core.Context`. A continuación mostramos un ejemplo:

```
@Path("/coches/{marca}")
public class CarResource {

  @GET
  @Path("/{modelo}/{anyo}")
  @Produces("image/jpeg")
  public Response getImagen(@Context UriInfo info) {
          String fabricado = info.getPathParameters().getFirst("marca");
          PathSegment modelo = info.getPathSegments().get(2);
          String color = modelo.getMatrixParameteres().getFirst("color");
          ...
  }
}
```

En este ejemplo, inyectamos una instancia de *UriInfo* como parámetro del método *getImagen()*. A continuación hacemos uso de dicha instancia para extraer información de la URI.

Recuerda que también podríamos inyectar una instancia de *UriInfo* en una variable de instancia de la clase raíz de nuestro recurso.

El método **CarResource.getImagen()** utiliza la interfaz `javax.ws.rs.core.PathSegment` que, como ya hemos indicado, representa un segmento de ruta.

```
package javax.ws.rs.core;
public interface PathSegment {
       String getPath();
       MultivaluedMap<String, String> getMatrixParameters();
}
```

El método **PathSegment.getPath()** devuelve el valor de la cadena de caracteres del segmento de ruta actual, sin considerar nigún parámetro matrix que pudiese contener.

El método **PathSegment.getMatrixParameters()** devuelve un "mapa" con todos los parámetros matrix aplicados a un segmento de ruta.

Supongamos que realizamos la siguiente petición http para el código anterior (clase CarResource):

```
GET /coches/seat/leon;color=rojo/2015
```

Esta petición es delegada en el método *ClarResource.getImagen()*. La ruta contiene 4 segmentos: *coches*, *seat*, *leon* y *2015. La variable _modelo* tomará el valor *leon* y la variable *color* se instanciará con el valor *rojo*.

## @javax.ws.rs.MatrixParam

La especificación JAX-RS nos permite inyectar una matriz de valores de parámetros a través de la anotación `javax.ws.rs.MatrixParam`:

```
@Path("/coches/{marca}")
public class CarResource {

  @GET
  @Path("/{modelo}/{anyo}")
  @Produces("image/jpeg")
  public Response getImagen(@PathParam("marca") String marca
                            @PathParam("modelo") String modelo
                            @MatrixParam("color") String color) {
  ... }
}
```

El uso de la anotación `@MatrixParam` simplifica nuestro código y lo hace algo más legible. Si, por ejemplo, la petición de entrada es:

```
GET /coches/seat/ibiza;color=black/2009
```

entonces el parámetro *color* del método *CarResource.getImagen()* tomaría el valor *black*.

## @javax.ws.rs.QueryParam

La anotación `@javax.ws.rs.QueryParam` nos permite inyectar parámetros de consulta (**query parameters**) de la URI en los valores de los parámetros de los métodos java de nuestros recursos. Por ejemplo, supongamos que queremos consultar información de nuestros clientes y queremos recuperar un subconjunto de clientes de nuestra base de datos. Nuestra URI de petición podría ser algo así:

```
GET /clientes?inicio=0&total=10
```

El parámetro de consulta `inicio` representa el índice (o posición) del primer cliente que queremos consultar, y el parámetro `total` representa cuántos clientes en total queremos obtener como respuesta. Una implementación del servicio RESTful podría contener el siguiente código:

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Produces("application/xml")
  public String getClientes(@QueryParam("inicio") int inicio,
```

```
                                @QueryParam("total") int total)
  ... }
}
```

En este ejemplo, el parámetro `inicio` tomaría el valor `0`, y el parámetro `total` tomaría el valor `10` (JAX-RS convierte automáticamente las cadenas de caracteres de los parámetros de consulta en enteros).

## @javax.ws.rs.FormParam

La anotación `@javax.ws.rs.FormParam` se utiliza para acceder al cuerpo del mensaje de la petición HTTP de entrada, cuyo valor de *Content-Type* es **application/x-www-form-urlencoded**. Es decir, se utiliza para acceder a entradas individuales de un formulario HTML. Por ejemplo, supongamos que para registrar a nuevos clientes en el sistema tenemos que rellenar el siguiente formulario:

```
<FORM action="http://ejemplo.com/clientes" method="post">
  <P>
    Nombre: <INPUT type="text" name="nombre"><BR>
    Apellido: <INPUT type="text" name="apellido"><BR>
    <INPUT type="submit" value="Send">
  </P>
</FORM>
```

La ejecución de este código, inyectará los valores del formulario como parámetros de nuestro método Java que representa el servicio, de la siguiente forma:

```
@Path("/clientes")
public class ClienteResource {
   @POST
   public void crearCliente(@FormParam("nombre") String nom,
                            @FormParam("apellido") String ape) {
   ... }
}
```

Aquí estamos inyectando los valores de `nombre` y `apellidos` del formulario HTML en los parámetors `nom` y `ape` del método java *crearCliente()*. Los datos del formulario "viajan" a través de la red codificados como *URL-encoded*. Cuando se utiliza la anotación @FormParam, JAX-RS decodifica de forma automática las entradas del fomulario antes de inyectar sus valores.

Así, por ejemplo, si tecleamos los valores *Maria Luisa* y_Perlado_, como valores en los campos de texto *nombre* y *apellido* del formulario, el cuerpo de nuestro mensaje HTTP será **nombre=Maria%20Luisa;apellido=Perlado**. Este mensaje será recibido por nuestro método, que extraerá los valores correspondientes y los instanciará en los parámetros *nom*, y *ape* del método _ClienteResource.crearCliente().

## @javax.ws.rs.HeaderParam

La anotación `@javax.ws.rs.HeaderParam` se utiliza para inyectar valores de las cabeceras de las peticiones HTTP. Por ejemplo, si estamos interesados en la página web que nos ha referenciado o enlazado con nuestro servicio web, podríamos acceder a la cabecera HTTP **Referer** utilizando la anotación @HeaderParam, de la siguiente forma:

```
@Path("/miservicio")
public class MiServicio {
  @GET
  @Produces("text/html")
  public String get(@HeaderParam("Referer") String referer) {
  ... }
}
```

De forma alternativa, podemos acceder de forma programativa a todas las cabeceras de la petición de entrada, utilizando la interfaz `javax.ws.rs.core.HttpHeaders`.

```
public interface HttpHeaders {
  public List<String> getRequestHeader(String name);
  public MultivaluedMap<String, String> getRequestHeaders();
  ...
}
```

El método `getRequestHeader()` permite acceder a una cabecera en concreto, y el método `getRequestHeaders()` nos proporciona un objeto de tipo *Map* que representa **todas** las cabeceras. A continuación mostramos un ejemplo que accede a todas las cabeceras de la petición HTTP de entrada.

```
@Path("/miservicio")
public class MiServicio {
  @GET
  @Produces("text/html")
  public String get(@Context HttpHeaders cabeceras) {
     String referer = headers.getRequestHeader("Referer").get(0);
     for (String header : headers.getRequestHeaders().keySet()) {
        System.out.println("Se ha utilizado esta cabecera : " + header);
     }
     ...
  }
}
```

## @javax.ws.rs.core.Context

Dentro de nuestros recursos JAX-RS podemos inyectar determinados objetos con información sobre el contexto de JAX-RS, sobre el contexto de servlets, o sobre elementos de la petición recibida desde el cliente. Para ello utilizaremos la anotación `@javax.ws.rs.core.Context`.

En los ejemplos de esta sesión, ya hemos visto como utilizarla para inyectar objetos de tipo *UriInfo* y *HttpHeaders*.

A continuación mostramos un ejemplo en el que podos obtener detalles sobre el contexto del despliegue de la aplicacion, asi como del contexto de peticiones individuales utilizando la anotacion @Context:

**Implementación de un servicio que muestra información sobre el contexto de la petición**

```
@Path("orders")
```

```
public class PedidoResource {
    @Context Application app; ❶
    @Context UriInfo uri; ❷
    @Context HttpHeaders headers; ❸
    @Context Request request; ❹
    @Context SecurityContext security; ❺
    @Context Providers providers; ❻

    @GET
    @Produces("application/xml")
    public List<Order> getAll(@QueryParam("start")int from,
    @QueryParam("end")int to) { //. . .(app.getClasses());
         //. . .(uri.getPath());
         //. . .(headers.getRequestHeader(HttpHeaders.ACCEPT));
         //. . .(headers.getCookies());
         //. . .(request.getMethod());
         //. . .(security.isSecure());
         //. . .
    }
}
```

❶ **Application** proporciona acceso a la información de la configuración de la aplicación (clase Application)

❷ **UriInfo** proporciona acceso a la URI de la petición

❸ **HttpHeaders** proporciona acceso a las cabeceras de la petición HTTP La anotación @HeaderParam puede también utilizarse para enlazar una cabecera HTTP a un parámetro de un método de nuestro recurso, a un campo del mismo, o a una propiedad de un *bean*

❹ **Request** se utiliza para procesar la respuestas, típicamente se usa juntamente con la clase Response para construir la respuesta de forma dinámica

❺ **SecurityContext** proporciona acceso a la información de la petición actual relacionada con la seguridad

❻ **Providers** proporciona información sobre la búsqueda del runtime de las instancias de proveedores utilizando un conjunto de criterios de búsqueda

Con respecto a contexto de servlets, podremos inyectar información de *ServletContext*, *ServletConfig*, *HttpServletRequest*, y *HttpServletResponse*. Debemos recordar que los recursos JAX-RS son invocados por un servlet dentro de una aplicación web, por lo que podemos necesitar tener acceso a la información del contexto de servlets. Por ejemplo, si necesitamos acceder a la ruta en disco donde tenemos los datos de nuestra aplicación web tendremos que inyectar el objeto @ServletContext:

```
@GET
@Produces("image/jpeg")
public InputStream getImagen(@Context ServletContext sc) {
    return sc.getResourceAsStream("/fotos/" + nif + ".jpg");
}
```

## @javax.ws.rs.BeanParam

La anotación `@javax.ws.rs.BeanParam` nos permite inyectar una clase específica cuyos métodos o atributos estén anotados con alguna de las anotaciones de inyección de parámetros *@xxxParam* que hemos visto en esta sesión. Por ejemplo, supongamos esta clase:

```
public class ClienteInput {
   @FormParam("nombre")
   String nombre;

   @FormParam("apellido")
   String apellido;

   @HeaderParam("Content-Type")
   String contentType;

   public String getFirstName() {...}
   ...
}
```

La clase `ClienteInput` es un simple POJO (Plain Old Java Object) que contiene el nombre y apellidos de un cliente, así como el tipo de contenido del mismo. Podemos dejar que JAX-RS cree, inicialice, e inyecte esta clase usando la anotación @BeanParam de la siguiente forma:

```
@Path("/clientes")
public class ClienteResource {
  @POST
  public void crearCliente(@BeanParam ClienteInput newCust) {
  ...}
}
```

El runtime de JAX-RS "analizará" los parámetros anotados con @BeanParam para inyectar las anotaciones correspondientes y asignar el valor que corresponda. En este ejemplo, la clase *ClienteInput* contendrá dos valores de un formulario de entrada, y uno de los valores de la cabecera de la petición. De esta forma, nos podemos evitar una larga lista de parámetros en el método *crearCliente()* (en este caso son sólo tres pero podrían ser muchos más).

## Conversión automática de tipos

Todas las anotaciones que hemos visto referencian varias partes de la petición HTTP. Todas ellas se representan como una cadena de caracteres en dicha petición HTTP. JAX-RS puede convertir esta cadena de caracteres en cualquier tipo Java, siempre y cuando se cumpla **al menos uno** de estos casos:

1. Se trata de un tipo primitivo. Los tipos *int*, *short*, *float*, *double*, *byte*, *char*, y *boolean*, pertenecen a esta categoría.
2. Se trata de una clase Java que tiene un constructor con un único parámetro de tipo *String*
3. Se trata de una clase Java que tiene un método estático denominado *valueOf()*, que toma un único *String* como argumento, y devuelve una instancia de la clase.
4. Es una clase de tipo java.util.List<T>, java.util.Set<T>, o java.util.SortedSet<T>, en donde T es un tipo que satisface los criterios 2 ó 3, o es un *String*. Por ejemplo, List<Double>, Set<String>, o SortedSet<Integer>.

Si el runtime JAX-RS falla al convertir una cadena de caracteres en el tipo Java especificado, se considera un error del cliente. Si se produce este fallo durante el procesamiento de una inyección de tipo @MatrixParam, @QueryParam, o @PathParam, se devuelve al cliente un error "404 Not found". Si el fallo tiene lugar con el procesamiento de las inyecciones

@HeaderParam o @CookieParam (esta última no la hemos visto), entonces se envía al cliente el eror "400 Bad Request".

### Valores por defecto (@DefaultValue)

Suele ser habitual que algunos de los parámetros proporcionados en las peticiones a servicios RESTful sean opcionales. Cuando un cliente no proporciona esta información opcional en la petición, JAX-RS inyectará por defecto un valor null si se trata de un objeto, o un valor cero en el caso de tipos primitivos.

Estos valores por defecto no siempre son los que necesitamos para nuestro servicio. Para solucionar este problema, podemos definir nuestro propio valor por defecto para los parámetros que sean opcionales, utilizando la anotación `@javax.ws.rs.DefaultValue`.

Consideremos el ejemplo anterior relativo a la recuperación de la información de un subconjunto de clientes de nuestra base de datos. Para ello utilizábamos dos parámetros de consulta para indicar el índice del primer elemento, así como el número total de elementos que estamos interesados en recuperar. En este caso, no queremos que el cliente tenga que especificar siempre estos parámetros al realizar la peticion. Usaremos la anotación @DefaultValue para indicar los valores por defecto que nos interese.

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Produces("application/xml")
  public String getClientes(
         @DefaultValue("0") @QueryParam("inicio") int inicio,
        @DefaultValue("10") @QueryParam("total") int total)
  ... }
}
```

Hemos usado @DefaultValue para especificar un índice de comienzo con valor cero, y un tamaño del subconjunto de los datos de la respuesta. JAX-RS utilizará las reglas de conversión de cadenas de caracteres que acabamos de indicar para convertir el valor del parámetro en el tipo Java que especifiquemos.

## 2.5. Configuración y despliegue de aplicaciones JAX-RS

Como ya hemos visto en la sesión anterior, implementamos nuestros servicios REST utilizando el API de Java JAX-RS (especificación JSR-339). Una aplicación JAX-RS consiste en uno o más **recursos** y cero o más **proveedores**. En este apartado vamos a describir ciertos aspectos aplicados a las aplicaciones JAX-RS como un todo, concretamente a la configuración y también a la publicación de las mismas cuando utilizamos un servidor de aplicaciones JavaEE 7 o bien un contenedor de servlets 3.0, que incluyan una implementación del API JAX-RS. También indicaremos cómo configurar el despliegue en el caso de no disponer como mínimo de un contenedor de *servlets 3.0*.

### Configuración mediante la clase *Application*

Tanto los recursos (clases anotadas con @Path) como los proveedores que conforman nuestra aplicación JAX-RS pueden configurarse utilizando una subclase de **Application**. Cuando hablamos de configuración nos estamos refiriendo, en este caso, a definir los mecanismos para localizar las clases que representan los recursos, así como a los proveedores.

Un **proveedor** es una clase que implementa una o alguna de las siguientes interfaces JAX-RS: MesssageBodyReader, MessageBodyWriter, ContextResolver<T>, y ExceptionMapper<T>. Las dos primeras permiten crear *proveedores de entidades* (*entity providers*), la tercera es un *proveedor de contexto* (*context provider*), y la última un proveedor de mapeado de excepciones (*exception mapping provider*). Las clases que actúan como "proveedores" están anotadas con @Provider, para que puedan ser identificadas automáticamente por el *runtime* JAX-RS.

El uso de una subclase de **Application** para configurar nuestros servicios REST constituye la forma más sencilla de desplegar los servicios JAX-RS en un servidor de aplicaciones certificado como Java EE (en este caso, Wildfly cumple con este requisito), o un contenedor *standalone* de *Servlet 3* (como por ejemplo Tomcat).

Pasemos a conocer la clase `javax.ws.rs.core.Application`. El uso de la clase *Application* es la única forma **portable** de "decirle" a JAX-RS qué servicios web (clases anotadas con @Path), así como qué otros elementos, como filtros, interceptores,…, queremos publicar (desplegar).

La clase **Application** se define como:

```
package javax.ws.rs.core;

import java.util.Collections;
import java.util.Set;

public abstract class Application {
   private static final Set<Object> emptySet =
                           Collections.emptySet();

   public abstract Set<Class<?>> getClasses();

   public Set<Object> getSingletons() {
      return emptySet;
   }
}
```

La clase *Application* es muy simple. Como ya hemos indicado, su propósito es proporcionar una lista de clases y objetos que "queremos" desplegar.

El método `getClasses()` devuelve una lista de **clases** de servicios web y proveedores JAX-RS. Cualquier **servicio** JAX-RS devuelto por este método sigue el modelo *per-request*, que ya hemos introducido en la sesión anterior. Cuando la implementación de JAX-RS determina que una petición HTTP necesita ser procesada por un método de una de estas clases, se creará una instancia de dicha clase durante la petición, y se "destruirá" al finalizar la misma. En este caso estamos **delegando** en el *runtime* JAX-RS la **creación de los objetos**. Las clases "proveedoras" son instanciadas por el contenedor JAX-RS y registradas una única vez por aplicación.

El método `getSingletons()` devuelve una lista de servicios y proveedores web JAX-RS "ya instanciados". Nosotros, como programadores de las aplicaciones, somos responsables de crear estos objetos. El *runtime* JAX-RS iterará a través de la lista de **objetos** y los registrará internamente.

Un ejemplo de uso de una subclase de *Application* podría ser éste:

```
package org.expertojava;

import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;

@ApplicationPath("/rest")
public class ComercioApplication extends Application {

  public Set<Class<?>> getClasses() {
      HashSet<Class<?>> set = new HashSet<Class<?>>();
      set.add(ClienteResource.class);
      set.add(PedidoResource.class);
      return set;
  }

  public Set<Object> getSingletons() {
     JsonWriter json = new JsonWriter();
     TarjetaCreditoResource servicio = new TarjetaCreditoResource();

     HashSet<Object> set = new HashSet();
     set.add(json);
     set.add(servicio);
     return set;
   }
}
```

La anotación `@ApplicationPath` define la base URL de la ruta para todos nuestros servicios JAX-RS desplegados. Así, por ejemplo, accederemos a todos nuestros servicios JAX-RS serán desde la ruta "/rest" cuando los ejecutemos. En el ejemplo anterior estamos indicando que *ClienteResource* y *PedidoResource* son servicios *per-request*. El método *getSingletons()* devuelve el servicio de tipo *TarjetaCreditoResource*, así como el proveedor *JsonWriter* (que implementa la interfaz *MessageBodyWriter*).

Si tenemos al menos una implementación de la clase *Application* anotada con *@ApplicationPath*, esta será "detectada" y desplegada automáticamente por el servidor de aplicaciones.

Podemos aprovechar completamente esta capacidad para "escanear" y detectar automáticamente nuestros servicios si tenemos implementada una subclase de Application, pero dejamos que *getSingletons()* devuelva el conjunto vacío, y no indicamos nada en el método *getClasses()*, de esta forma:

```
package org.expertojava;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

@ApplicationPath("/rest")
public class ComercioApplication extends Application {

}
```

En este caso, el servidor de aplicaciones se encargará de buscar en el directorio *WEB-INF/classes* y en cualquier fichero *jar* dentro del directorio *WEB-INF/lib*. A continuación añadirá cualquier clase anotada con *@Path* o *@Provider* a la lista de "cosas" que necesitan ser desplegadas y registradas en el *runtime* JAX-RS.

Los servicios REST son "atendidos" por un *servlet*, que es específico de la implementación JAX-RS utilizada por el servidor de aplicaciones. El servidor wildfly utiliza la implementación de JAX-RS 2.0 denomindada *resteasy* (otra implementación muy utilizada es *jersey*, por ejemplo con el servidor de aplicaciones *Glassfish*). El *runtime* de JAX-RS contiene un *servlet* inicializado con un parámetro de inicialización de tipo *javax.ws.rs.Application*, cuyo valor será instanciado "automáticamente" por el servidor de aplicaciones con el nombre de la subclase de *Application* que sea detectada en el *war* de nuestra aplicación.

## Configuración mediante un fichero *web.xml*

En la sesión anterior, no hemos utilizado de forma explícita la clase *Application* para configurar el despliegue. En su lugar, hemos indicado esta información en el fichero "web.xml":

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
    <!-- Con estas líneas, el servidor es el responsable de
      añadir el servlet correspondiente de forma automática.
      Si en nuestro war, tenemos clases anotadas con anotaciones JAX-RS
      para recibir invocaciones REST, éstas serán detectadas y
 registradas-->
    <servlet-mapping>
        <servlet-name>javax.ws.rs.core.Application</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Esta configuración es equivalente a incluir una subclase de *Application* sin sobreescribir los métodos correspondientes. En este caso, se añade de forma dinámica el servlet que sirve las peticiones REST, con el nombre *javax.ws.rs.core.Application*, de forma que se detecten automáticamente todas las clases de recursos, y clases proveedoras empaquetadas en el *war* de la aplicación.

## Configuración en un contenedor que no disponga de una implementación JAX-RS

Si queremos hacer el despliegue sobre servidores de aplicaciones o servidores web que den soporte a una especificación de *servlets* con una versión inferior a la *3.0*, tendremos que configurar MANUALMENTE el fichero *web.xml* para que "cargue" el servlet de nuestra implementación propietaria de JAX-RS (cuyos ficheros *jar* deberemos incluir en el directorio *WEB-INF/lib* de nuestro war). Un ejemplo de configuración podría ser éste:

**Configuración del fichero web.xml (directorio de fuentes: webapp/WEB-INF/web.xml)**

```
<?xml version="1.0"?>
<web-app>
   <servlet>
```

```
        <servlet-name>JAXRS</servlet-name>
        <servlet-class>
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
        </servlet-class>
        <init-param>
           <param-name>
              javax.ws.rs.Application
           </param-name>
           <param-value>
              org.expertoJava.ComercioApplication
           </param-value>
         </init-param>
    </servlet>

    <servlet-mapping>
       <servlet-name>JAXRS</servlet-name>
       <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

En la configuración anterior estamos indicando de forma explícita el *servlet* JAX-RS que recibe las peticiones REST, que a su vez, utilizará la clase Application para detectar qué servicios y proveedores REST serán desplegados en el servidor.

También será necesario incluir la librería con la implementación JAX-RS 2.0 de forma explícita en el *war* generado (recordemos que para ello, tendremos que utilizar la etiqueta <scope>compile<scope>, para que se añadan los *jar* correspondientes).

### Librería con la implementación de JAX-RS 2.0

```
<dependency>
  <groupId>javax</groupId>
  <artifactId>javaee-web-api</artifactId>
  <version>7.0</version>
  <scope>compile</scope>
</dependency>
```

## 2.6. Ejercicios

Para esta sesión añadiremos un nuevo módulo en el que implementaremos un servicio rest incorporando los conceptos que hemos explicado durante la sesión. En concreto:

- Creamos un módulo Maven con IntelliJ (desde el directorio `ejercicios-rest-expertojava`) con el arquetipo `webapp-javaee7`, tal y como hemos visto en los apuntes de la sesión. Las coordenadas del **artefacto Maven** serán:

    # *GroupId*: org.expertojava

    # *ArtifactId*: s2-foro-nuevo

    # *version*: 1.0-SNAPSHOT

- Configuramos el pom.mxl del proyecto para poder compilar, empaquetar y desplegar nuestro servicio en el servidor de aplicaciones Wildfly. Consulta los apuntes para ver cuál debe ser el contenido de las etiquetas *<properties>*, *<dependencies>* y *<build>*
- Vamos a estructurar los fuentes (directorio **src/main/java**) de nuestro proyecto en los siguientes paquetes:

    # *org.expertojava.datos*: contendrá clases relacionadas con los datos a los que accede nuestra aplicación rest. Por simplicidad, almacenaremos en memoria los datos de nuestra aplicación.

    # *org.expertojava.modelo*: contiene las clases de nuestro modelo de objetos, que serán clases java con atributos y sus correspondientes *getters* y *setters*

    # *org.expertojava.rest*: contiene los recursos JAX-RS, que implementan nuestros servicios rest, así como las clases necesarias para automatizar el despliegue de dichos recursos

### Creación de un recurso: creación y consulta de temas en el foro (0,5 puntos)

Vamos a crear un recurso JAX-RS al que denominaremos `TemasResource` (en el paquete *org.expertojava.rest* ). En el siguiente ejercicio , al configurar la aplicación, haremos que este recurso sea un **singleton**. Nuestro recurso gestionará sus propios datos en memoria. Por ejemplo podemos utilizar un atributo *private* de tipo *HashMap* en el que almacenaremos los temas, cada uno con un identificador numérico como clave. También necesitaremos un atributo para generar las claves para cada uno de los temas. Por ejemplo:

```
private Map<Integer, Tema> temasDB = new HashMap<Integer, Tema>();
private int contadorTemas = 0;
```

Fíjate que si utilizamos los tipos *HashMap* e *int* podemos tener problemas de concurrencia si múltiples usuarios están realizando peticiones para crear y/o consultar los temas del foro. En una situación real deberíamos utilizar en su lugar los tipos *ConcurrentHasMap* y *AtomicInteger*, para evitar el que dos usuarios intentaran crear un nuevo tema con la misma clave, perdiéndose así uno de los dos temas creados. Al tratarse de un ejercicio en el que solamente tendremos un cliente, no nos planteará ningún problema el trabajar con *HashMap* e *int*, por lo que podéis elegir cualquiera de las dos opciones para realizar el ejercicio

- Nuestro recurso estará accesible en el servidor en la ruta `/temas` (relativa a la raíz del contexto de nuestra aplicación, y a la ruta de nuestro servlet JAX-RS, que determinaremos con la anotación @ApplicationPath de nuestra clase *Application*).
- En el paquete *org.expertojava.modelo* crearemos la clase `Tema`, con los atributos privados:

```
int id;
String nombre;
```

y sus correspondientes *getters* y *setters*:

```
setId(), getId()
setNombre(), getNombre()
```

- Implementamos un primer método en el recurso `TemasResource`, denominado `creaTema()`, para poder crear un nuevo tema en el foro. Dicho método atenderá peticiones POST a nuestro servicio. Los datos de entrada (cadena de caracteres que respresenta el nombre del tema) se pasan a través de un formulario html, en el que tenemos una única entrada denominada "nombre".

Puedes incluir el siguiente contenido en el fichero `index.html` para introducir los datos desde el navegador:

```
<!DOCTYPE html>
<html>
<head>
    <title>Start Page</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
<h1>Alta de temas en el foro: </h1>
<form action="/s2-foro-nuevo/rest/temas" method="post">
    Nombre del tema: <input type="text" name="nombre" /><br />

    <input type="submit" value="Enviar" />
</form>
</body>
</html>
```

Cada nuevo `Tema` creado se añadirá a nuestra base de datos en memoria `temasDB` junto con un identificador numérico (que se irá incrementando para cada nueva instancia creada).

- Implementamos un segundo método para consultar los temas creados en el foro. El método se denominará `verTemasTodos()`, y devuelve (en formato texto) todos los temas actualmente creados. Dado que puede haber un gran número de ellos, vamos a permitir que el usuario decida cuántos elementos como máximo quiere consultar a partir de una posición determinada. Por defecto, si no se indica esta información, se mostrarán como máximo los primeros 8 temas registrados en el foro. Si el identificador a partir del cual queremos iniciar la consulta es mayor que el número de temas almacenados, entonces devolveremos la cadena: "No es posible atender la consulta". Ejemplos de URIs que acepta dicho método son:

# /temas

en este caso, y suponiendo que hayamos creado solamente los tres temas del apartado anterior, el resultado sería:

```
"Listado de temas del 1 al 8:
1. animales
2. plantas
3. ab"
```

# /temas?inicio=2&total=2

el resultado sería:

```
"Listado de temas del 2 al 3:
2. plantas
3. ab"
```

# /temas?inicio=7&total=1

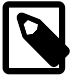
el resultado sería:

```
"No es posible atender la consulta"
```

Como ya hemos comentado, las URIs indicadas en este ejercicio son relativas a la raíz del contexto de nuestra aplicación y a la ruta especificada para nuestros servicios rest. Recuerda que si has configurado el pom.xml como en la sesión anterior, la raíz del contexto de la aplicación vendrá dada por el valor de la etiqueta <finalName>, anidada en <build>. En nuestro caso debería ser "/s2-foro-nuevo". Más adelante, fijaremos la ruta de nuestros servicios rest como `/rest`. Por ejemplo la URI completa para el último apartado sería: http://localhost:8080/s2-foro-nuevo/rest/temas?inicio=7&total=1

## Despliegue y pruebas del recurso (0,5 puntos)

Vamos a construir y desplegar nuestro servicio en el servidor de aplicaciones. Para ello vamos a utilizar una subclase de *Application* que añadiremos en el paquete *org.expertojava.rest*. La ruta en la que se van a servir nuestras peticiones rest será `rest`. Fíjate que el recurso que hemos creado es el encargado de gestionar (crear, modificar,…) sus propios datos. Por lo tanto necesitamos que nuestro recurso REST sea un *singleton*. Implementa la clase `ForoApplication` y realiza la construcción y despliegue del proyecto. A continuación prueba el servicio utilizando postman. Puedes probar la inserción de temas utilizando también el formulario a través de la URI: *http://localhost:8080/s2-foro-nuevo*. Podemos utilizar las entradas del apartado anterior, de forma que comprobemos que se crean correctamente los temas "animales", "plantas", y "ab", y que obtenemos los listados correctos tanto si no indicamos el inicio y total de elementos, como si decidimos mostrar los temas desde el 2 hasta el 3.



Cuando utilices el cliente IntelliJ para probar métodos POST, debes proporcionar un **Request Body no vacío**. En este caso, como en la propia URI incluimos el contenido del mensaje, que es el nombre del tema que queremos añadir al foro tendrás que seleccionar **Text** aunque no rellenemos el campo correspondiente. De no hacerlo así, obtendremos como respuesta un cuerpo de mensaje vacío, y la cabecera de respuesta **HTTP/1.1 415 Unsupported Media Type**

## Múltiples consultas de los temas del foro (0,5 puntos)

Implementa tres nuevas consultas de los temas del foro, de forma que:

- Se pueda realizar una consulta de un tema concreto a partir de su identificador numérico (el método solamente debe admitir identificadores formados por uno o más dígitos). Si el tema consultado no existe se debe devolver una excepción con la cabecera de respuesta **HTTP/1.1 404 Not Found**. Por ejemplo:

  # /temas/2

  Debe devolver lo siguiente:

  ```
  "Ver el tema 2:
  plantas"
  ```

  # /temas/4

  Obtenemos como respuesta un cuerpo de mensaje vacío, y la cabecera de respuesta: **HTTP/1.1 404 Not Found**

- Se pueda realizar una consulta de los temas que comiencen por uno de los siguientes caracteres: a, b, c, ó d. Por ejemplo, teniendo en cuenta que hemos introducido los temas anteriores:

  # /temas/a

  Debe devolver lo siguiente:

  ```
  "Listado de temas que comienzan por a:
  animales"
  ```

  # /temas/d

  Debe devolver: "Listado de temas que comienzan por d:"

- Se pueda realizar una consulta de los temas que contengan una subcadena de caracteres. Por ejemplo, teniendo en cuenta que hemos introducido los temas anteriores:

  # /temas/ma + Debe devolver lo siguiente:

  ```
  "Listado de temas que contienen la subcadena : ma
  animales"
  ```

## Creación de subrecursos (0,5 puntos)

Vamos a crear el subrecurso `MensajesResource` (en el paquete *org.expertojava.rest*), de forma que este recurso gestione la creación y consulta de mensajes para cada uno de los temas del foro. Este subrecurso debe atender peticiones desde rutas del tipo: `/temas/identificadorTema/mensajes`, siendo *identificadorTema* la clave numérica asociada a uno de los temas almacenados.

- En este caso, nuestro subrecurso no será un *singleton*, por lo que necesitaremos almacenar los mensajes en otra clase diferente (ya que crearemos una nueva instancia del recurso para cada petición). La clase `DatosEnMemoria` (en el paquete *org.expertojava.datos*) será la encargada de almacenar en memoria la información de los mensajes publicados para cada tema. Por ejemplo puedes utilizar los siguientes campos **estáticos** para gestionar los mensajes:

```
public static Map<Mensaje, String> mensajesDB =
                        new HashMap<Mensaje, String>();
```

  La clave será el propio mensaje (objeto *Mensaje*, que se asociará al tema correspondiente)

```
public static int contadorMen = 0;
```

Como ya hemos comentado puedes usar *ConcurrentHashMap* y *AtomicInteger* en lugar de los tipos anteriores, para evitar problemas de concurrencia.

- En el paquete *org.expertojava.datos* crearemos la clase `Mensaje`, con los atributos privados:

```
int id;
String texto;
String autor="anonimo";
```

y sus correspondientes *getters* y *setters*:

```
setId(), getId()
setTexto(), getTexto()
setAutor(), getAutor()
```

- Vamos a crear un método para poder realizar la publicación de un mensaje de texto en el foro, en uno de los temas ya creados. Independientemente del tipo de petición realizada sobre los mensajes, si el tema indicado en la URI no existe, lanzaremos la excepción *WebApplicationException(Response.Status.NOT_FOUND)*. Veamos algún ejemplo:

  # Deberemos poder realizar una petición **POST** a **/temas/1/mensajes**, con el cuerpo de mensaje = "Mensaje numero 1". El mensaje creado, por defecto tendrá asociado el autor "anonimo"

- # Si realizamos una petición para añadir un mensaje a la URI: **/temas/9/mensajes**, deberíamos obtener como cabecera de respuesta: **HTTP/1.1 404 Not Found**, independientemente del cuerpo del mensaje

- Vamos a crear un método para realizar una consulta de todos los mensajes publicados en un tema concreto. Por ejemplo:

  # Una petición **GET** a **/temas/1/mensajes** debería dar como resultado:

```
"Lista de mensajes para el tema: animales
1. Mensaje anonimo"
```

  # Si realizamos una petición **GET** a la URI: **/temas/9/mensajes**, deberíamos obtener como cabecera de respuesta: **HTTP/1.1 404 Not Found**, independientemente del cuerpo del mensaje

- Finalmente vamos a añadir dos nuevos métodos para: (a) añadir un nuevo mensaje en un tema concreto, indicando el autor del mensaje. Como restricción, el nombre del autor deberá estar formado solamente por caracteres alfabéticos, utilizando mayúsculas o minúsculas, y como mínimo tiene que tener un caracter; y (b) consultar todos los mensajes que un determinado autor ha publicado en el foro en un tema determinado

  # Una petición **POST** a la URI: /temas/1/mensajes/pepe, con el cuerpo de mensaje con valor "mensaje de pepe" debería crear un nuevo mensaje para el tema con identificador 2, y devolver como resultado el nuevo id (y/o la URI del nuevo recurso en la cabecera de respuesta *Location*, si seguimos la ortodoxia REST). En caso de que devolvamos la URI del nuevo recurso podemos utilizar la orden:

```
return Response.created(uriInfo.getAbsolutePathBuilder()  ❶
                                .segment(String.valueOf(id))  ❷
                                .build())  ❸
                        .build(); ❹
```

❶ Obtenemos el path absoluto de la uri que nos ha invocado
❷ Añadimos el identificador *id* del nuevo recurso creado
❸ Construimos la nueva URI
❹ Construimos el objeto Response.

Veremos cómo manipular objetos de tipo *Response* en sesiones posteriores.

Recuerda que para acceder al cuerpo de la petición basta con definir un parámetro de tipo *String*. JAX-RS automáticamente lo instanciará con el cuerpo de la petición como una cadena.

- Una petición **GET** a la URI: /temas/1/mensajes/anonimo, daría como resultado:

  "Lista de mensajes tema= animales ,y autor= anonimo

  1. Mensaje anonimo"

- Una petición **GET** a la URI: /temas/1/mensajes/, daría como resultado:

```
"Lista de mensajes para el tema: animales
```

```
1. Mensaje anonimo
2. mensaje de pepe"
```

- Una petición **GET** a la URI: /temas/1/mensajes/roberto, daría como resultado:

```
"Lista de mensajes tema= animales ,y autor= roberto"
```

# 3. Manejadores de contenidos. Respuestas del servidor y manejo de excepciones.

En la sesión anterior hemos hablado de cómo inyectar información contenida en las cabeceras de las peticiones HTTP, ahora nos detendremos en el cuerpo del mensaje, tanto de la petición como de la respuesta. En el caso de las peticiones, explicaremos el proceso de transformar los datos de entrada en objetos Java, para poder ser procesados por nuestros servicios. Con respecto a las respuestas proporcionadas por nuestros servicios, analizaremos tanto los códigos de respuesta por defecto, como la elaboración de respuestas complejas y manejo de excepciones.

## 3.1. Proveedores de entidades

JAX-RS define lo que se denominan **proveedores de entidades**, que son clases que proporcionan servicios de mapeado entre las representaciones del cuerpo del mensaje HTTP y los correspondientes tipos java que utilizaremos en nuestros recursos (parámetros en los métodos, o bien como tipo de la respuesta de los mismos). Las **entidades** también se conocen con el nombre de **"message payload"**, o simplemente como **payload**, y representan el **contenido del cuerpo** del mensaje HTTP.

**Providers**

> El *runtime* de JAX-RS puede "extenderse" (ampliarse) utilizando clases "proveedoras" (*providers*) suministradas por nuestra aplicación. Concretamente, JAX-RS nos proporciona un conjunto de interfaces que podemos implementar en nuestra aplicación, creando así dichas clases "proveedoras de entidades" (*entity providers*). La especificación de JAX-RS define un **proveedor** como una clase que implementa una o más interfaces JAX-RS (de entre un conjunto determinado) y que pueden anotarse con `@provider` para ser "descubiertas" de forma automática por el *runtime* de JAX-RS.

Nuestra aplicación puede proporcionar su propio mapeado entre representaciones (tipos MIME) del mensaje de entrada y tipos Java implementando las interfaces `MessageBodyWriter` y `MessageBodyReader`, convirtiéndose así en clases **proveedoras de entidades** (**entity providers**). Por ejemplo, podemos tener nuestro propio proveedor de entidades para el formato XML, o JSON, de forma que, utilizando las librerías de java para procesamiento XML o JSON (*Java API for XML Processing: JAXP*[1] y *Java API for JSON Processing: JSON-P*[2]), implementemos el serializado/deserializado del cuerpo del mensaje HTTP de entrada cuando éste presente los tipos MIME *"application/xml"* o *"application_json"*. Las clases que realizan dichos mapeados son clases *entity provider*.

### Interfaz javax.ws.rs.ext.MessageBodyReader

La interfaz `MessageBodyReader` define el contrato entre el *runtime* de JAX-RS y los componentes que proporcionan servicios de mapeado desde diferentes representaciones (indicadas como tipos *mime*) al tipo Java correspondiente. Cualquier clase que quiera proporcionar dicho servicio debe implementar la interfaz `MessageBodyReader` y debe anotarse con @Provider para poder ser "detectada" de forma automática por el *runtime* de JAX-RS.

[1] https://www.jcp.org/en/jsr/detail/summary?id=206
[2] https://jcp.org/en/jsr/detail?id=353

La secuencia lógica de pasos seguidos por una implementación de JAX-RS cuando se mapea el cuerpo de un mensaje HTTP de entrada a un parámetro de un método Java es la siguiente:

1. Se obtiene el *media type* de la petición (valor de la cabecera HTTP `Content-Type`). Si la petición no contiene una cabecera `Content-Type` se usará `application/octet-stream`
2. Se identifica el tipo java del parámetro cuyo valor será mapeado desde el cuerpo del mensaje
3. Se localiza la clase `MessageBodyReader` que soporta el *media type* de la petición y se usa su método `readFrom()` para mapear el contenido del cuerpo del mensaje HTTP en el tipo Java que corresponda
4. Si no es posible encontrar el *MessageBodyReader* adecuado se genera la excepción `NotSupportedException`, con el código 405

### Interfaz javax.ws.rs.ext.MessageBodyWriter

La interfaz `MessageBodyWriter` define el contrato entre el *runtime* de JAX-RS y los componentes que proporcionan servicios de mapeado desde un tipo Java a una representación determinada. Cualquier clase que quiera proporcionar dicho servicio debe implementar la interfaz `MessageBodyWriter` y debe anotarse con `@Provider` para poder ser "detectada" de forma automática por el *runtime* de JAX-RS.

La secuencia lógica de pasos seguidos por una implementación de JAX-RS cuando se mapea un valor de retorno de un método del recurso a una entidad del cuerpo de un mensaje HTTP es la siguiente:

1. Se obtiene el objeto que será mapeado a la entidad del cuerpo del mensaje
2. Se determina el *media type* de la respuesta
3. Se localiza la clase `MessageBodyWriter` que soporta el objeto que será mapeado a la entidad del cuerpo del mensaje HTTP, y se utiliza su método `writeTo()` para realizar dicho mapeado
4. Si no es posible encontrar el `MessageBodyWriter` adecuado se genera la excepción `InternalServerErrorException` (que es una subclase de `WebApplicationException`) con el código 500

## 3.2. Proveedores de entidad estándar incluidos en JAX-RS

Cualquier implementación de JAX-RS debe incluir un conjunto de implementaciones de `MessageBodyReader` y `MessageBodyWriter` de forma predeterminada para ciertas combinaciones de tipos Java y *media types*.

### Table 3. Proveedores de entidades estándar de una implementación JAX-RS

| Tipo Java | Media Type |
| --- | --- |
| byte[] | */* (Cualquier *media type*) |
| java.lang.String | */* (Cualquier *media type*) |
| java.io.InputStream | */* (Cualquier *media type*) |

| Tipo Java | Media Type |
|---|---|
| java.io.Reader | */* (Cualquier *media type*) |
| java.io.File | */* (Cualquier *media type*) |
| javax.activation.DataSource | */* (Cualquier *media type*) |
| javax.xml.transform.Source | text/xml, application/xml, application/*+xml (tipos basados en xml) |
| javax.xml.bind.JAXBElement and application-supplied JAXB classes | text/xml, application/xml, application/*+xml (tipos basados en xml) |
| MultivaluedMap<String,String> | application/x-www-form-urlencoded (Contenido de formularios) |
| StreamingOutput | */* (Cualquier *media type*) (Sólo `MessageBodyWriter`) |
| java.lang.Boolean, java.lang.Character, java.lang.Number | text/plain |

A continuación comentaremos algunos de estos *proveedores de entidades* estándar, o "conversores" por defecto, que permiten convertir el cuerpo del mensaje HTTP a objetos Java de diferentes tipos y viceversa.

## javax.ws.rs.core.StreamingOutput

*StreamingOutput* es una interfaz *callback* que implementamos cuando queremos tratar como un flujo continuo (*streaming*) el cuerpo de la respuesta. Constituye una alternativa "ligera" al uso de `MessageBodyWriter`.

```
public interface StreamingOutput {
  void write(OutputStream output)
     throws IOException, WebApplicationException;
}
```

Implementamos una instancia de esta interfaz, y la utilizamos como tipo de retorno de nuestros métodos de recursos. Cuando el *runtime* de JAX-RS está listo para escribir el cuerpo de respuesta del mensaje, se invoca al método `write()` de la instancia de `StreamingOutput`. Veamos un ejemplo:

```
@Path("/miservicio") public class MiServicio {
  @GET
  @Produces("text/plain")
  StreamingOutput get() {
    return new StreamingOutput() {
      public void write(OutputStream output)
            throws IOException, WebApplicationException {
                output.write("hello world".getBytes());
      }
  };
}
```

Hemos utilizado una clase interna anónima que implementa la interfaz `StreamingOutput` en lugar de crear una clase pública separada. La razón de utilizar una clase interna, es porque

en este caso, al contener tan pocas líneas de código, resulta beneficioso mantener dicha lógica "dentro" del método del recurso JAX-RS, de forma que el código sea más fácil de "seguir". Normalmente no tendremos necesidad de reutilizar la lógica implementada en otros métodos, por lo que no tiene demasiado sentido crear otra clase específica.

¿Y por qué no inyectamos un `OutputStream` directamente? ¿Por qué necesitamos un objeto `callback`? La razón es que así dejamos que el runtime de JAX-RS maneje la salida de la manera que quiera. Por ejemplo, por razones de rendimiento, puede ser conveniente que JAX-RS utilice un *thread* para responder, diferente del *thread* de petición.

## java.io.InputStream, java.io.Reader

Para leer el cuerpo de un mensaje de entrada, podemos utilizar las clases `InputStream` o `Reader`. Por ejemplo:

```
@Path("/")
public class MiServicio {
  @PUT
  @Path("/dato")
  public void modificaDato(InputStream is) {
     byte[] bytes = readFromStream(is);
     String input = new String(bytes);
     System.out.println(input);
  }

  private byte[] readFromStream(InputStream stream)
                           throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    byte[] buffer = new byte[1000];
    int wasRead = 0;
    do {
       wasRead = stream.read(buffer);
       if (wasRead > 0) {
          baos.write(buffer, 0, wasRead);
       }
    } while (wasRead > -1);
    return baos.toByteArray();
  }
}
```

En este caso estamos leyendo *bytes* a partir de un `java.io.InputStream`, para convertirlo en una cadena de caracteres que mostramos por pantalla.

En el siguiente ejemplo, creamos un `java.io.LineNumberReader` a partir de un objeto `Reader` e imprimimos cada línea del cuerpo del mensaje de entrada:

```
@PUT
@Path("/maslineas")
public void putMasLineas(Reader reader) {
    LineNumberReader lineReader = new LineNumberReader(reader);
    do{
        String line = lineReader.readLine();
        if (line != null) System.out.println(line);
```

```
    } while (line != null);
}
```

No estamos limitados solamente a utilizar instancias de `InputStream` y/o `Reader` para leer el cuerpo de los mensajes de entrada. También podemos devolver dichos objetos como respuesta. Por ejemplo:

```
@Path("/fichero")
public class FicheroServicio {
  private static final String basePath = "...";

  @GET
  @Path("{rutafichero: .*}")
  @Produces("text/plain")
  public InputStream getFichero(@PathParam("rutafichero")
                                String path) {
    FileInputStream is = new FileInputStream(basePath + path);
    return is;
  }
}
```

Aquí estamos inyectando un valor `@PathParam` para crear una referencia a un fichero real de nuestro disco duro. Creamos una instancia de `java.io.FileInputStream` a partir del valor de la ruta inyectada como parámetro y la devolvemos como cuerpo de nuestro mensaje de respuesta. La implementación de JAX-RS leerá la respuesta de este *stream* de entrada y la almacenará en un *buffer* para posteriormente "escribirla" de forma incremental en el *stream* de salida de la respuesta. En este caso debemos especificar la anotación @Produces para que la implementación de JAX-RS conozca el valor que debe asignar a la cabecera `Content-Type` en la respuesta.

## java.io.File

Se pueden utilizar instancias de la clase `java.io.File` para **entrada y salida** de cualquier MIME-TYPE (especificado en `Content-Type` y/o `Accept`, y en las anotaciones `@Produces` y/o `@Consumes`) . El siguiente código, por ejemplo, devuelve una referencia a un fichero en nuestro disco:

```
@Path("/fichero")
public class FicheroServicio {
  private static final String baseRuta = "...";

  @GET
  @Path("{rutafichero: .*}")
  @Produces("text/plain")
  public File getFichero(@PathParam("rutafichero")
                                String ruta) {
    return new File(baseRuta + ruta);
  }
}
```

En este caso inyectamos el valor de la ruta del fichero con la anotación `@PathParam`. A partir de dicha ruta creamos un objeto `java.io.File` y lo devolvemos como cuerpo del mensaje

de respuesta. La implementación JAX-RS "leerá" la información "abriendo" un `InputStream` basado en esta referencia al fichero y la "escribirá" en un *buffer*. Posteriormente y de forma incremental, volverá a escribir el contenido del *buffer* en el *stream* de salida de la respuesta. Al igual que en el ejemplo anterior, debemos especificar la anotación @Produces para que JAX-RS sepa cómo "rellenar" la cabecera `Content-Type` de la respuesta.

También podemos inyectar instancias de `java.io.File` a partir del cuerpo del mensaje de la petición. Por ejemplo:

```
@POST
@Path("/masdatos")
public void post(File fichero) {
    Reader reader = new Reader(new FileInputStream(fichero));
    LineNumberReader lineReader = new LineNumberReader(reader);
    do{
      String line = lineReader.readLine();
      if (line != null) System.out.println(line);
    } while (line != null);
}
```

En este caso la implementación de JAX-RS crea un fichero temporal en el disco para la entrada. Lee la información desde el *buffer* de la red y guarda los *bytes* leídos en este fichero temporal. En el ejemplo, los datos leídos desde la red están representados por el el objeto File inyectado por el *runtime* de JAX-RS (recuerda que sólo puede haber un parámetro sin anotaciones en los métodos del recurso y que éste representa el cuerpo del mensaje de la petición HTTP). A continuación, el método `post()` crea un `java.io.FileInputStream` a partir del objeto `File` inyectado. Finalmente utilizamos éste *stream* de entrada para crear un objeto `LineNumberReader` y mostrar los datos por la consola.

## byte[]

Podemos utilizar un array de bytes como **entrada y salida** para cualquier tipo especificado como *media-type*. A continuación mostramos un ejemplo:

```
@Path("/")
public class MiServicio {
   @GET
   @Produces("text/plain")
   public byte[] get() {
         return "hello world".getBytes();
   }

   @POST
   @Consumes("text/plain")
   public void post(byte[] bytes) {
       System.out.println(new String(bytes)); }
   }
}
```

Para cualquier método de recurso JAX-RS que devuelva un array de bytes, debemos especificar la anotación @Produces para que JAX-RS sepa qué valor asignar a la cabecera `Content-Type`.

## String, char[]

La mayor parte de formatos en internet están basados en texto. JAX-RS puede convertir cualquier formato basado en texto a un `String` o a cualquier array de caracteres. Por ejemplo:

```
@Path("/")
public class MiServicio {
  @GET
  @Produces("application/xml")
  public String get() {
    return "<customer><name>Sergio Garcia</name></customer>";
  }

  @POST
  @Consumes("text/plain")
  public void post(String str) {
    System.out.println(str);
  }
}
```

Para cualquier método de recurso JAX-RS que devuelva un `Sring` o un array de caracteres debemos especificar la anotación `@Produces` para que JAX-RS sepa que valor asignar a la cabecera `Content-Type`.

## MultivaluedMap<String, String> y formularios de entrada

Los formularios HTML son usados habitualmente para enviar datos a servidores web. Los datos del formulario están codificados con el *media type* `application/x-www-form-urlencoded`. Ya hemos visto como utilizar la anotación `@FormParam` para inyectar parámetros individuales de un formulario de las peticiones de entrada. También podremos inyectar una instancia de `MultivaluedMap<String, String>`, que representa todos los datos del formulario enviado en la petición. Por ejemplo:

```
@Path("/")
  public class MiServicio {
    @POST @Consumes("application/x-www-form-urlencoded")
    @Produces("application/x-www-form-urlencoded")
    public MultivaluedMap<String,String> post(
                        MultivaluedMap<String, String> form) {
        //el formulario tiene los campos "fieldName1" y "fieldName2"
        System.out.println(form.getFirst("fieldName1"));
        System.out.println(form.getFirst("fieldName2"));
        return form; }
}
```

En este código, nuestro método `post()` acepta peticiones POST y recibe un `MultivaluedMap<String,String>` que contiene todos los datos de nuestro formulario. En este caso también devolvemos una instancia de un formulario como respuesta.

Los datos del formulario pueden representarse en el cuerpo de la petción como pares `nombre=valor` separados por `&`. Por ejemplo:

```
fieldName1=valor%20con%20espacios&fielName2=otroValor
```

Los espacios en blanco se codifican como `%20`. No es necesario poner comillas.

## 3.3. Múltiples representaciones de recursos

Por defecto, un recurso RESTful se produce o consume con el tipo MIME "/". Un recurso RESTful puede restringir los *media types* que soporta, tanto en la petición como en la respuesta, utilizando las anotaciones @Consumes y @Produces, respectivamente. Estas anotaciones pueden especificarse, como ya hemos visto, a nivel de clase o de método de recurso. Las anotaciones especificadas sobre el método prevalecen sobre las de la clase. La ausencia de estas anotaciones es equivalente a su inclusión con el tipo MIME *("/")*, es decir, su ausencia implica que se soporta cualquier tipo.

A continuación mostramos un ejemplo en el que un *Pedido* puede "producirse" tanto en formato xml como en formato json:

```
@GET
@Path("{id}")
@Produces({"application/xml", "application/json"})
public Pedido getPedido(@PathParam("id")int id) { . . . }
```

El método *getPedido()* puede generar ambas representaciones para el pedido. El tipo exacto de la respuesta viene determinado por la cabecera HTTP **Accept** de la petición.

Otro ejemplo, en el que pueden "consumirse" varios tipos MIME, puede ser el siguiente:

```
@POST
@Path("{id}")
@Consumes({"application/xml", "application/json"})
public Pedido addPedido(@PathParam("id")int id) { . . . }
```

En este caso el formato "consumido" vendrá dado por el valor de la cabecera HTTP **Content-Type** de la petición.

JAX-RS 2.0 nos permite indicar la preferencia por un *media type*, en el lado del servidor, utilizando el parámetro **qs** (**q**uality on **s**ervice). **qs** toma valores entre 0.000 y 1.000, e indica la calidad relativa de una representación comparado con el resto de representaciones disponibles. Una representación con un valor de *qs* de 0.000 nunca será elegido. Una representación sin valor para el parámetro *qs* se asume que dicho valor es 1.000

Ejemplo:

```
@POST
@Path("{id}")
@Consumes({"application/xml; qs=0.75", "application/json; qs=1"})
public Pedido addPedido(@PathParam("id")int id) { . . . }
```

Si un cliente realiza una petición y no manifiesta ninguna preferencia por ninguna representación en particular, o con una cabecera **Accept** con valor *application/**, entonces el

servidor seleccionará la representación con el valor de *qs* más alto (en este caso *application/json*). Los valores de *qs* son relativos, y como tales, solamente son comparables con otros valores *qs* dentro de la misma instancia de la anotación @Consumes (o @Produces).

Los clientes pueden indicar también sus preferencias utilizando otro factor relativo de calidad, en forma de parámetro denominado **q**. El valor del parámetro **q** se utiliza para ordenar el conjunto de tipos aceptados. **q** toma valores entre 0.000 y 1.000 (máxima preferencia). Al igual que antes, Los valores de *q* son relativos, y como tales, solamente son comparables con otros valores *q* dentro de la misma cabecera *Accept* o *Content-type*.

Las preferencias del servidor (valores de los parámetros **qs**) sólo se tienen en cuenta si el cliente acepta múltiples *media types* con el **mismo valor de q**

Veamos un ejemplo:

```
@GET
@Path("{id}")
@Produces({"application/xml"; qs=1,
           "application/json"; qs=0.75})
public Pedido getPedido(@PathParam("id")int id) { . . . }
```

Supongamos que un cliente lanza una petción GET con una valor para la cabecera **Accept** de *application/*; q=0.5, text/html*. En este caso, el servidor determina que los tipos MIME *application/xml* y *application/json* tienen la misma preferencia por parte del cliente (con valor de 0.5), por lo tanto, el servidor elegirá la representación *application/json*, ya que tiene un valor de *qs* mayor.

## 3.4. Introducción a JAXB

JAXB (**J**ava **A**rchitecture for **X**ML **B**inding) es una especificación Java antigua (JSR 222[3]) y no está definida por JAX-RS. JAXB es un *framework* de anotaciones que mapea clases Java a XML y esquemas XML. Es extremadamente útil debido a que, en lugar de interactuar con una representación abstracta de un documento XML, podemos trabajar con objetos Java reales que están más cercanos al dominio que estamos modelando. JAX-RS proporciona soporte para JAXB, pero antes de revisar los manejadores de contenidos JAXB incuidos con JAX-RS, veamos una pequeña introducción al *framework* JAXB.

Como ya hemos dicho, si queremos mapear una clase Java existente a XML, podemos utilizar JAXB, a través de un conjunto de anotaciones. Veámoslo mejor con un ejemplo:

```
@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    @XmlAttribute
    protected int id;

    @XmlElement
    protected String nombre;

    public Customer() {}
```

[3] https://jcp.org/aboutJava/communityprocess/mrel/jsr222/index2.html

```
    public int getId() { return this.id; }
    public void setId(int id) { this.id = id; }

    public String getNombre() { return this.nombre; }
    public void setNombre(String nombre} { this.nombre = nombre; }
}
```

La anotación `@javax.xml.bind.annotation.XmlRootElement` se utiliza en clases java para denotar que representan elementos XML (etiqueta XML raíz). En este caso estamos diciendo que la clase Java representa un documento XML que tiene como etiqueta raíz `<cliente>`. Las clases java anotadas con `@XmlRootElement` se denomina `beans JAXB`.

La anotación `@javax.xml.bind.annotation.XmlAttribute` la hemos asociado al campo `id` de nuestra clase `Cliente`. Esta anotación indica que el campo `id` de la clase debe mapearse como el atributo `id` del elemento raíz `<cliente>` del documento XML. La anotación `@XmlAttribute` tiene un atributo `name`, de forma que podemos especificar el nombre exacto del atributo XML dentro del documento. Por defecto, tiene el mismo nombre que el campo anotado.

Hemos utilizado la anotación `@javax.xml.bind.annotation.XmlElement` en el campo `nombre` de la clase `Cliente`. Esta anotación indica a JAXB que debe mapearse el campo `nombre` como el elemento `<nombre>` anidado en la etiqueta raíz `<cliente>`. Igual que antes, podemos especificar el nombre concreto del elememto XML. Por defecto toma el mismo nombre que el correspondiente campo anotado.

La anotación `@javax.xml.bind.annotation.XmlAccessorType` permite controlar la serialización por defecto de los atributos de la clase. Esta anotación sólo puede ser usada conjuntamente con `@XmlRootElement` (y alguna otra anotación que no mostramos aquí). Hemos usado como valor `XmlAccessType.FIELD`, lo que significa que por defecto se deben serializar **todos** los **campos** (*fields*) de la clase (estén anotados o no), y las **propiedades** (*properties*) de la clase que tengan anotaciones JAXB (a menos que la anotación sea *@XMLTransient*).

Si alguno de los **campos** de la clase no tiene anotaciones JAXB asociadas, por defecto se serializarán como *elementos* (*etiquetas*) en el documento XML correspondiente. Según la documentación[4] de JAXB, un *campo* es una variable de instancia no estática (normalmente privada).

Las **propiedades** de la clase vienen dadas por las combinaciones getter/setter de los atributos de la clase. El código anterior tiene dos propiedades "nombre" (dado por el par *getNombre*/ setNombre_) e "id" (par *getId*/*setId*). Normalmente se anotan los métodos *getter*. Dichas propiedades no están anotadas, por lo que JAXB no las serializará.

Al proceso de serializar (convertir) un objeto Java en un documento XML se le denomina **marshalling**. El proceso inverso, la conversión de XML a objetos Java se denomina **unmarshalling**.

Con las anotaciones anteriores, un ejemplo de una instancia de nuestra clase `Cliente` con un `id` de 42, y el valor de `nombre` `Pablo Martinez`, tendría el siguiente aspecto:

```
<cliente id="42">
```

[4] https://jaxb.java.net/nonav/2.2.6/docs/api/

```
    <nombreCompleto>Pablo Martinez</nombreCompleto>
</cliente>
```

Observamos que se han serializado los campos (variables de instancia de la clase).

Si no especificamos la anotación `@XmlAccessorType`, por defecto se utilizará:

```
@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
```

El valor `XmlAccessType.PUBLIC_MEMBER` indica a JAXB que se deben serializar todos los campos públicos de la clase, todos los campos anotados, y todas las propiedades (pares *getter/setter*), a menos que estén anotadas con *@XMLTransient*.

También podríamos utilizar:

```
@XmlAccessorType(XmlAccessType.NONE)
```

En este caso, la anotación `XmlAccessType.NONE` indica sólo se deben serializar aquellas propiedades y/o campos de la clase que estén anotados.

A continuación indicamos, en forma de tabla, el uso de los diferentes `XmlAccessType`:

### Table 4. Valores utilizados para `@XmlAccessorType()`, conjuntamente con `@XmlRootElement()`:

| Valor | Significado |
|---|---|
| XmlAccessType.PUBLIC_MEMBER | Serialización por defecto si no se especifica `@XmlAccessorType()`. Serializa las propiedades (pares *getter/setter*) y campos públicos, a menos que estén anotados con `@XMLTransient`. Si algún campo no público está anotado, también se serializa. |
| XmlAccessType.FIELD | Serializa todos los campos (públicos o privados) a menos que estén anotados con `@XMLTransient`. |
| XmlAccessType.NONE | Solamente serializa aquellos campos y propiedades que estén anotadas con anotaciones JAXB |
| XmlAccessType.PROPERTY | Serializa cada par *getter/setter*, a menos que estén anotados con `@XMLTransient`. Si algún campo (no público) está anotado también se serializa |

Podemos utilizar la anotación `@XmlElement` para anidar otras clases anotadas con JAXB. Por ejemplo, supongamos que queremos añadir una clase `Direccion` a nuestra clase `Cliente`:

```
@XmlRootElement(name="direccion")
```

```
@XmlAccessorType(XmlAccessType.FIELD)
public class Direccion {
  @XmlElement
  protected String calle;

  @XmlElement
  protected String cludad;

  @XmlElement
  protected String codPostal;

  // getters y setters
  ...
}
```

Simplemente tendríamos que añadir el campo de tipo `Direccion` a nuestra clase Cliente, de la siguiente forma:

```
@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    @XmlAttribute
    protected int id;

    @XmlElement
    protected String nombreCompleto;

    @XmlElement
    protected Direccion direccion;

    public Customer() {}

    // getters y setters
    ...
}
```

En este caso, una instancia de un `Cliente` con valores `id`=56, `nombre`="Ricardo Lopez", `calle`="calle del oso, 35", `ciudad`="Alicante", y `código_postal`="01010", sería serializado como:

```
<cliente id="56">
   <nombre>Ricardo Lopez</nombre>
   <direccion>
      <calle>calle del oso, 35</calle>
      <ciudad>Alicante</ciudad>
      <codPostal>01010</codPostal>
   </direccion>
</cliente>
```

Veamos otro ejemplo. Supongamos que tenemos el recurso `EstadoResource`, con métodos que responden a peticiones http GET y PUT:

```
@Path("/estado")
```

```
public class EstadoResource {

    private static EstadoBean estadoBean = new EstadoBean();

    @GET
    @Produces("application/xml")
    public EstadoBean getEstado() {
        return estadoBean;
    }

    @PUT
    @Consumes("application/xml")
    public void setEstado(EstadoBean estado) {
       this.estadoBean = estado;
    }
}
```

En este caso, la clase `EstadoBean` debe utilizar anotaciones JAXB para poder ser convertida automáticamente por el *runtime* de JAX-RS en un documento XML, y viceversa:

```
@XmlRootElement(name = "estadoImpresora")
public class EstadoBean {

   public String estado = "Idle";
   public int tonerRestante = 25;
   public List<TareaBean> tareas =
             new ArrayList<TareaBean>();
}
```

Por defecto, la anotación `@XmlRootElement` realiza la serialización de la clase `EstadoBean` en formato xml utilizando los campos públicos y propiedades definidas en la clase (`estado`, `tonerRestante`, y `tareas`). Vemos que el campo tareas, a su vez, es una colección de elementos de tipo `TareaBean`, que también necesitan ser serializados. A continuación mostramos la implementación de la clase `TareaBean.java`:

```
@XmlRootElement(name = "tarea")
public class TareaBean {
   public String nombre;
   public String estado;
   public int paginas;

   public TareaBean() {}; ❶

   public TareaBean(String nombre, String estado,
                 int paginas) {
      this.nombre = nombre;
      this.estado = estado;
      this.paginas = paginas;
   }
}
```

❶ Para serializar la clase, es necesario que la clase tenga un constructor sin parámetros

Si accedemos al servicio anterior, nos devolverá la información sobre el estado de la siguiente forma (resultado devuelto por el método `getEstado()`, anotado con @GET):

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<estadoImpresora>
    <estado>Idle</estado>
    <tonerRestante>25</tonerRestante>
    <tareas>
        <nombre>texto.doc</nombre>
        <estado>imprimiendo...</estado>
        <paginas>13</paginas>
    </tareas>
    <tareas>
        <nombre>texto2.doc</nombre>
        <estado>en espera...</estado>
        <paginas>5</paginas>
    </tareas>
</estadoImpresora>
```

Podemos observar que se utiliza el elemento xml *<tareas>* para representar cada una de las instancias de *TareaBean*, las cuales forman parte de de la lista del campo *tareas* de nuestra clase *EstadoBean*.

Vamos a usar las anotaciones `@XmlAttribute` y `@XmlElement` de la siguiente forma:

```
@XmlRootElement(name="estadoImpresora")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
             new ArrayList<TareaBean>();
}
```

En este caso, el XML resultante quedaría de la siguiente forma:

```
<estadoImpresora valor="Idle" toner="25">
    <tarea>
        <nombre>texto.doc</nombre>
        <estado>imprimiendo...</estado>
        <paginas>13</paginas>
    </tarea>
    <tarea>
        <nombre>texto2.doc</nombre>
        <estado>en espera...</estado>
        <paginas>5</paginas>
    </tarea>
</estadoImpresora>
```

Si no se indica lo contrario, por defecto se convierten los campos a elementos del XML.

En caso de que los campos no sean públicos, etiquetaremos los *getters* correspondiente (propiedades de la clase).

Hemos visto que para las listas el nombre que especificamos en `@XmlElement` se utiliza para nombrar cada elemento de la lista. Si queremos que además se incluya un elemento que envuelva a toda la lista, podemos utilizar la etiqueta `@XmlElementWrapper`:

```
@XmlRootElement(name="estadoImpresora")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElementWrapper(name="tareas")
   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
             new ArrayList<TareaBean>();
}
```

En este caso tendremos un XML como el que se muestra a continuación:

```
<estadoImpresora valor="Idle" toner="25">
    <tareas>
        <tarea>
            <nombre>texto.doc</nombre>
            <estado>imprimiendo...</estado>
            <paginas>13</paginas>
        </tarea>
        <tarea>
            <nombre>texto2.doc</nombre>
            <estado>en espera...</estado>
            <paginas>5</paginas>
        </tarea>
    </tareas>
</estadoImpresora>
```

Para etiquetar una lista también podemos especificar distintos tipos de elemento según el tipo de objeto contenido en la lista. Por ejemplo, supongamos que en el ejemplo anterior la clase `TareaBean` fuese una clase abstracta que tiene dos posible subclases: `TareaSistemaBean` y `TareaUsuarioBean`. Podríamos especificar una etiqueta distinta para cada elemento de la lista según el tipo de objeto del que se trate con la etiqueta `@XmlElements`, de la siguiente forma:

```
@XmlElementWrapper(name="tareas")
   @XmlElements({
       @XmlElement(name="usuario",type=TareaUsuarioBean.class},
```

```
        @XmlElement(name="sistema",type=TareaSistemaBean.class}})
    public List<TareaBean> tareas =
              new ArrayList<TareaBean>();
```

De esta forma podríamos tener un XML como el siguiente:

```
<estadoImpresora valor="Idle" toner="25">
    <tareas>
        <usuario>
            <nombre>texto.doc</nombre>
            <estado>imprimiendo...</estado>
            <paginas>13</paginas>
        </usuario>
        <sistema>
            <nombre>texto2.doc</nombre>
            <estado>en espera...</estado>
            <paginas>5</paginas>
        </sistema>
    </tareas>
</estadoImpresora>
```

Hemos visto que por defecto se serializan todos los campos. Si queremos excluir alguno de ellos de la serialización, podemos hacerlo anotándolo con `@XmlTransient`. Como alternativa, podemos cambiar el comportamiento por defecto de la serialización de la clase etiquetándola con `@XmlAccessorType`. Por ejemplo:

```
@XmlAccessorType(NONE)
@XmlRootElement(name="estadoImpresora")
public class EstadoBean {
    ...
}
```

En este último caso, especificando como tipo `NONE`, no se serializará por defecto ningún campo, sólo aquellos que hayamos anotado explícitamente con `@XmlElement` o `@XmlAttribute`. Los campos y propiedades (*getters*) anotados con estas etiquetas se serializarán siempre. Además de ellos, también podríamos especificar que se serialicen por defecto todos los campos públicos y los *getters* (`PUBLIC_MEMBER`), todos los *getters* (`PROPERTY`), o todos los campos, ya sean públicos o privados (`FIELD`). Todos los que se serialicen por defecto, sin especificar ninguna etiqueta, lo harán como elemento.

Por último, si nos interesa que toda la representación del objeto venga dada únicamente por el valor de uno de sus campos, podemos etiquetar dicho campo con `@XmlValue`.

## Clase JAXBContext

Para serializar clases Java a y desde XML, es necesario interactuar con la clase `javax.xml.bind.JAXBContext`. Esta clase nos permite "inspeccionar" las clases Java para "comprender" la estructura de nuestras clases anotadas. Dichas clases se utilizan como factorías para las interfaces `javax.xml.bind.Marshaller`, y `javax.xml.bind.Unmarshaller`. Las instancias de *Marshaller* se utilizan para crear documentos XML a partir de objetos Java. Las instancias de *Unmarshaller* se utilizan para crear objetos Java a partir de documentos XML. A continuación mostramos un ejemplo de uso de

JAXB para convertir una instancia de la clase `Cliente`, que hemos definido anteriormente, a formato XML, para posteriormente volver a crear el objeto de tipo `Cliente`:

```
Cliente cliente = new Cliente();
cliente.setId(42);
cliente.setNombre("Lucia Arg");  ❶

JAXBContext ctx = JAXBContext.newInstance(Cliente.class); ❷
StringWriter writer = new StringWriter();

ctx.createMarshaller().marshal(cliente, writer);
String modString = writer.toString();  ❸

cliente = (Cliente)ctx.createUnmarshaller().
                unmarshal(new StringReader(modString)); ❹
```

❶ Creamos e inicializamos una instancia de tipo `Cliente`

❷ Inicializamos `JAXBContext para que pueda "analizar" la clase `Cliente`

❸ Utilizamos una instancia de *Marshaller* para escribir el objeto `Cliente` como un `String` de Java (`StringWriter` es un *stream* de caracteres que utilizaremos para construir un `String`)

❹ Utilizamos una instancia de *Unmarshaller* para recrear el objeto `Cliente` a partir del `String` que hemos obtenido en el paso anterior

La clase `JAXBContext` constituye un punto en entrada al API JAXB para los clientes de nuestros servicios RESTful. Proporciona una abstracción para gestionar el enlazado (*binding*) de información XML/Java, necesaria para implementar las operaciones de *marshalling* y *unmarshalling*

- **Unmarshalling**: La clase `Unmarshaller` proporciona a la aplicación cliente la capacidad para convertir datos XML en un "árbol" de objetos Java.
- **Marshalling**: La clase `Marshaller` proporciona a la aplicación cliente la capacidad para convertir un "árbol" con contenidos Java, de nuevo en datos XML.

Una vez que hemos proporcionado una visión general sobre cómo funciona JAXB, vamos a ver cómo se integra con JAX-RS

## Manejadores JAX-RS para JAXB

La especificación de JAX-RS indica que cualquier implementación debe soportar de forma automática el proceso de *marshalling* y *unmarshalling* de clases anotadas con `@XmlRootElement`. A continuación mostramos un ejemplo de la implementación de un servicio que hace uso de dichos manejadores. Para ello utilizamos la clase *Cliente* que hemos anotado previamente con @XmlRootElement y que hemos mostrado en apartados anteriores:

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Path("{id}")
```

```
   @Produces("application/xml")
   public Cliente getCliente(@PathParam("id") int id) {
      Cliente cli = findCliente(id);
      return cust;
   }

   @POST
   @Consumes("application/xml")
   public void crearCliente(Cliente cli) {
   ...
   }
}
```

Como podemos ver, una vez que aplicamos las anotaciones JAXB a nuestras clases Java ( en este caso a la clase `Cliente` ), es muy sencillo intercambiar documentos XML entre el cliente y nuestro servicio web. Los manejadores JAXB incluidos en la implementación de JAX-RS gestionarán el *marshalling/unmarshalling* de cualquier clase con anotaciones JAXB, para los valores de `Content-Type` `application/xml` , `text/xml` , o `application/*+xml` . Por defecto, también se encargan de la creación e inicialización de instancias `JAXBContext` . Debido a que la creación de las instancias `JAXBContext` puede ser "cara", la implementación de JAX-RS normalmente las "guarda" después de la primera inicialización.

## JAXB y JSON

JAXB es lo suficientemente flexible como para soportar otros formatos además de XML. Aunque la especificación de JAX-RS no lo requiere, muchas implementaciones de JAX-RS incluyen adaptadores de JAXB para soportar el formato JSON, además de XML. JSON es un formato basado en texto que puede ser interpretado directamente por Javascript. De hecho es el formato de intercambio preferido por las aplicaciones Ajax.

JSON es un formato mucho más simple que XML. Los objetos están entre llaves, `{}` , y contienen pares de *clave/valor* separados por comas. Los valores pueden ser cadenas de caracteres, booleanos ( `true` o `false` ), valores numéricos, o *arrays* de los tipos anteriores.

Supongamos que tenemos la siguiente descripción de un producto en formato XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<producto>
   <id>1</id>
   <nombre>iPad</nombre>
   <descripcion>Dispositivo móvil</descripcion>
   <precio>500</precio>
</producto>
```

La representación JSON equivalente sería:

```
{
  "id":"1",
  "nombre":"iPad",
  "descripcion":"Dispositivo móvil",
  "precio":500
}
```

El formato JSON asociado, por ejemplo, al siguiente objeto `Cliente` :

```
<cliente id="56">
   <nombre>Ricardo Lopez</nombre>
   <direccion>
      <calle>calle del oso, 35</calle>
      <ciudad>Alicante</ciudad>
      <codPostal>01010</codPostal>
   </direccion>
</cliente>
```

quedaría como sigue:

```
{
  "nombre": "Ricardo Lopez",
  "direccion": { "calle": "calle del oso, 35",
                 "ciudad": "Alicante",
                 "codPostal": "01010"
               }
}
```

Como vemos en el ejemplo, el formato JSON consiste básicamente en objetos situados entre llaves, los cuales están formados por pares clave/valor, separadas por comas. Cada clave y valor está separado por ":". Los *valores* pueden cadenas de caracteres, booleanos (*true* o *false*), valores numéricos, o vectores de los tipos anteriores (los vectores están delimitados por corchetes).

Podemos añadir el formato `application/json`, o bien `MediaType.APPLICATION_JSON` a la anotación `@Produces` en nuestros métodos de recurso para generar respuestas en formato JSON:

```
@GET
@Path("/get")
@Produces({"application/xml","application/json"})
public Producto getProducto() { ... }
```

En este ejemplo, se elegirá el formato JSON en la respuesta si el cliente realiza una petición GET que incluye en la cabecera:

```
Accept: application/json
```

El tipo de respuesta es de tipo `Producto`. En este caso `Producto` debe ser un *bean JAXB*, es decir, una clase anotada con `@XmlRootElement`.

Los métodos de recurso pueden aceptar también datos JSON para clases con anotaciones JAXB:

```
@POST
@Path("/producto")
@Consumes({"application/xml","application/json"})
```

```
public Response crearProducto(Producto prod) { ... }
```

En este caso el cliente debería incluir la siguiente cabecera cuando realice la petición POST que incluya los datos JSON anteriores en el cuerpo del mensaje:

```
Content-Type: application/json
```

Hablaremos con más detalle del formato JSON en una sesión posterior.

Finalmente, y como resumen de lo anterior:

### JAXB

Para dar soporte al serializado y deserializado XML se utilizan beans JAXB.Por ejemplo las clases anteriores `EstadoBean`, `TareaBean`, `Cliente` y `Producto` son ejemplos de beans JAXB. La clase que se serializará como un recurso XML o JSON se tiene que anotar con `@XmlRootElement`. Si en algún elemento de un recurso se retorna un elemento de esa clase y se etiqueta con los tipos `@Produces({MediaType.APPLICATION_XML,Mediatype.APPLICATION_JSON})`, éste se serializa automáticamente utilizando el tipo de representación aceptada por el cliente. Se puede consultar el artículo de Lars Vogel[5] para más información.

## 3.5. Respuestas del servidor

Vamos a explicar cuál es el comportamiento por defecto de los métodos de recursos JAX-RS, en particular veremos cuáles son los códigos de respuesta HTTP por defecto teniendo en cuenta situaciones de éxito, así como de fallo.

Dado que, en ocasiones, vamos a tener que enviar cabeceras de respuesta específicas ante condiciones de error complejas, también vamos a explicar cómo podemos elaborar respuestas complejas utilizando el API JAX-RS.

### Códigos de respuesta por defecto

Los códigos de respuesta por defecto se corresponden con el comportamiento indicado en la especificación[6] de la definición de los métodos HTTP 1.1. Vamos a examinar dichos códigos de respuesta con el siguiente ejemplo de recurso JAX-RS:

```
@Path("/clientes")
public class ClienteResource {

  @Path("{id}")
  @GET
  @Produces("application/xml")
  public Cliente getCliente(@PathParam("id") int id) {...}
```

[5] http://www.vogella.de/articles/JAXB/article.html
[6] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

```
    @POST
    @Produces("application/xml")
    @Consumes("application/xml")
    public Cliente crearCliente(Cliente nuevoCli) {...}

    @PUT
    @Path("{id}")
    @Consumes("application/xml")
    public void updateCliente(@PathParam("id") int id, Cliente cli) {...}

    @Path("{id}")
    @DELETE
    public void borrarCliente(@PathParam("id") int id) {...}
  }
```

## Respuestas que indican éxito

Los números de código de respuestas HTTP con éxito se sitúan en el rango de 200 a 399:

- Para los métodos `crearCliente()` y `getCliente()` se devolverá una respuesta HTTP con el código "200 OK", si el objeto `Cliente` que devuelven dichos métodos **no es** `null`.
- Para los métodos `crearCliente()` y `getCliente()` se devolverá una respuesta HTTP con el código "204 No Content", si el objeto `Cliente` que devuelven dichos métodos es `null`. El código de respuesta 204 no indica una condición de error. Solamente avisa al cliente de que todo ha ido bien, pero que el mensaje de respuesta no contiene nada en el cuerpo de la misma. Según ésto, si un método de un recurso devuelve `void`, por defecto se devuelve el código de respuesta "204 No content". Este es el caso para los métodos `updateCliente()`, y `borrarCliente()` de nuestro ejemplo.

La especificación HTTP es bastante consistente para los métodos PUT, POST, GET y DELETE. Si una respuesta exitosa HTTP contiene información en el cuerpo del mensaje de respuesta, entonces el código de respuesta es "200 OK". Si el cuerpo del mensaje está vacío, entonces se debe devolver "204 No Content".

## Respuestas que indican una situación de fallo

Es habitual que las respuestas "fallidas" se programen de forma que se lance una excepción. Lo veremos en un apartado posterior. Aquí comentaremos algunas condiciones de error por defecto.

Los números de código de error de respuesta estándar en HTTP se sitúan en el rango entre 400 y 599. En nuestro ejemplo, si un cliente se equivoca tecleando la URI, y ésta queda por ejemplo como `http://…/cliente`, entonces el servidor no encontrará ningún método del recurso que pueda servir dicha petición (la URI correcta sería `http://…/clientes`). En este caso, se enviará como respuesta el código "404 Not Found".

Para los métodos `getCliente()` y `crearCliente()` de nuestro ejemplo, si el cliente solicita una respuesta con el tipo MIME `text/html`, entonces la implementación de JAX-RS devolverá automáticamente "406 Not Acceptable", con un mensaje de respuesta con el cuerpo de dicho mensaje vacío. Esta respuesta indica que JAX-RS puede encontrar una ruta de URI relativa, que coincide con la petición, pero no encuentra ningún método del recurso que devuelva la respuesta con ese tipo MIME.

Si el cliente invoca una petición HTTP sobre una URI válida, para la que no se puede encontrar un método de recurso asociado, entonces el *runtime* de JAX-RS devolverá el código "405 Method Not Allowed". Así, en nuestro ejemplo, si el cliente solicita una operación PUT, GET, o DELETE sobre la URI `/clientes`, obtendrá como respuesta "405 Method Not Allowed", puesto que POST es el único método HTTP que puede dar soporte a dicha URI. La implementación de JAX-RS también devolverá una cabecera de respuesta `Allow` con la lista de métodos HTTP que pueden dar soporte a dicha URI. Por lo tanto, si nuestra aplicación cliente realiza la siguiente petición de entrada:

```
GET /clientes
```

el servidor devolverá la siguiente respuesta:

```
HTTP/1.1 405, Method Not Allowed
Allow: POST
```

## Elaboración de respuestas con la clase Response

Como ya hemos visto, por ejemplo para peticiones GET, si todo va bien se estará devolviendo un código de respuesta 200 (Ok), junto con el contenido especificado en el tipo de datos utilizado en cada caso. Si devolvemos void, el código de respuesta será 204 (No Content).

Sin embargo, en ocasiones, el servicio web que estamos diseñando no puede implementarse utilizando el comportamiento por defecto de petición/respuesta inherente a JAX-RS. En estos casos necesitaremos controlar de forma explícita la respuesta que se le envía al cliente (cuerpo del mensaje de la respuesta HTTP). Por ejemplo, cuando creamos un nuevo recurso con POST deberíamos devolver 201 (Created). Para tener control sobre este código nuestros recursos JAX-RS podrán devolver instancias de `javax.ws.rs.core.Response`:

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Response getClientes() {
  ClientesBean clientes = obtenerClientes();
  return Response.ok(clientes).build(); ❶
}

@POST
@Consumes(MediaType.APPLICATION_XML)
public Response addCliente(ClienteBean cliente,
                           @Context UriInfo uriInfo) {
  String id = insertarCliente(cliente); ❷
  URI uri = uriInfo
            .getAbsolutePathBuilder()
            .path("{id}")
            .build(id); ❸
  return Response.created(uri).build(); ❹
}
```

❶ Al crear una respuesta con Response, podemos especificar una entidad, que podrá ser un objeto de cualquiera de los tipos vistos anteriormente, y que representa los datos a

devolver como contenido. Por ejemplo, cuando indicamos **ok(clientes)**, estamos creando una respuesta con código 200 (Ok) y con el contenido generado por nuestro bean JAXB clientes. Esto será equivalente a haber devuelto directamente ClientesBean como respuesta, pero con la ventaja de que en este caso podemos controlar el código de estado de la respuesta.

❷ Insertamos una instancia de `ClienteBean` en la base de datos, y obtenemos la clave asociada a dicho cliente.

❸ En la sesión anterior hemos hablado de la interfaz `uriInfo`. Es una interfaz inyectable que proporciona acceso a información sobre la URI de la aplicación o la URI de las peticiones recibidas. En este caso, estamos construyendo una nueva URI formada por la ruta absoluta de la petición de entrada http POST, añadiéndole la plantilla "{id}" y finalmente sustituyendo el parámetro de la plantilla por el valor `id`. Supongamos que la petición POST contiene la uri `http://localhost:8080/recursos/clientes`. Y que el valor de `id` es 8. El valor de la variable **uri** será, por tanto: `http://localhost:8080/recursos/clientes/8`

❹ Devolvemos la respuesta incluyendo la URI anterior en la cabecera HTTP `Location

Veamos con más detalle la clase `Response`:

```
public abstract class Response {
  public abstract Object getEntity();  ❶
  public abstract int getStatus();  ❷
  public abstract MultivaluedMap<String, Object> getMetadata();  ❸
  public abstract URI getLocation();  ❹
  public abstract MediaType getMediaType();  ❺
  public abstract void close(); ❻
    ...
}
```

❶ El método `getEntity()` devuelve el objeto Java correspondiente al cuerpo del mensaje HTTP.

❷ El método `getStatus()` devuelve el código de respuesta HTTP.

❸ El método `getMetadata()` devuelve una instancia de tipo `MultivaluedMap` con las cabeceras de la respuesta.

❹ El método `getLocation()` devuelve la URI de la cabecera Location de la respuesta.

❺ El método `getMediaType()` devuelve el *mediaType* del cuerpo de la respuesta

❻ El método `close()` cierra el *input stream* correspondiente a la entidad asociada del cuerpo del mensaje (en el caso de que esté disponible y "abierto"). También libera cualquier otro recurso asociado con la respuesta (como por ejemplo datos posiblemente almacenados en un *buffer*)

Estos métodos típicamente serán invocados dese el cliente, tal y como veremos más adelante, cuando expliquemos el API cliente.

Los objetos `Response` no pueden crearse directamente. Tienen que crearse a partir de instancias de `javax.ws.rs.core.Response.ResponseBuilder` devueltas por uno de los siguientes métodos estáticos de `Response`:

```
public abstract class Response { ...
  public abstract void close() {...}
  public static ResponseBuilder status(Status status) {...}
  public static ResponseBuilder status(int status) {...}
  public static ResponseBuilder ok() {...}
```

```
  public static ResponseBuilder ok(Object entity) {...}
  public static ResponseBuilder ok(Object entity, MediaType type) {...}
  public static ResponseBuilder ok(Object entity, String type) {...}
  public static ResponseBuilder ok(Object entity, Variant var) {...}
  public static ResponseBuilder serverError() {...}
  public static ResponseBuilder created(URI location) {...}
  public static ResponseBuilder noContent() {...}
  public static ResponseBuilder notModified() {...}
  public static ResponseBuilder notModified(EntityTag tag) {...}
  public static ResponseBuilder notModified(String tag) {...}
  public static ResponseBuilder seeOther(URI location) {...}
  public static ResponseBuilder temporaryRedirect(URI location) {...}
  public static ResponseBuilder notAcceptable(List<Variant> variants)
 {...}
  public static ResponseBuilder fromResponse(Response response) {...}
  ...
}
```

Veamos por ejemplo el método `ok()`:

```
ResponseBuilder ok(Object entity, MediaType type) {...}
```

Este método recibe como parámetros un objeto Java que queremos convertir en una respuesta HTTP y el `Content-Type` de dicha respuesta. Como valor de retorno se obtiene una instancia de tipo `ResponseBuilder` pre-inicializada con un código de estado de `200 OK`.

El método `created()` devuelve un *ResponseBuilder* para un recurso creado, y asigna a la cabecera `Location` de la respuesta el valor de la URI proporionado como parámetro.

La clase `ResponseBuilder` es una factoría utilizada para crear instancias individuales de tipo `Response`:

```
public static abstract class ResponseBuilder {
  public abstract Response build();
  public abstract ResponseBuilder clone();

  public abstract ResponseBuilder status(int status);
  public ResponseBuilder status(Status status) {...}

  public abstract ResponseBuilder entity(Object entity);
  public abstract ResponseBuilder type(MediaType type);
  public abstract ResponseBuilder type(String type);

  public abstract ResponseBuilder variant(Variant variant);
  public abstract ResponseBuilder variants(List<Variant> variants);

  public abstract ResponseBuilder language(String language);
  public abstract ResponseBuilder language(Locale language);

  public abstract ResponseBuilder location(URI location);
  public abstract Response.ResponseBuilder header(String name, Object
 value)
  public abstract Response.ResponseBuilder link(URI uri, String rel)
```

```
    public abstract Response.ResponseBuilder link(String uri, String rel)
    ...
}
```

Vamos a mostrar un ejemplo sobre cómo crear respuestas utilizando un objeto `Response`. En este caso el método `getLibro()` devuelve un `String` que representa el libro en el que está interesado nuestro cliente:

```
@Path("/libro")
public class LibroServicio {
    @GET
    @Path("/restfuljava")
    @Produces("text/plain")
    public Response getLibro() {
        String libro = ...; ❶
        ResponseBuilder builder = Response.ok(libro); ❷
        builder.language("fr").header("Some-Header", "some value"); ❸
        return builder.build(); ❹
    }
}
```

❶ Recuperamos los datos del libro solicitado. En este caso vamos a devolver una cadena de caracteres que representa el libro en el que estamos interesados.
❷ Inicializamos el cuerpo de la respuesta utilizando el método `Response.ok()`. El código de estado de `ResponseBuilder` se inicializa de forma automática con 200.
❸ Usamos el método `ResponseBuilder.language()` para asignar el valor de la cabecera `Content-Languaje` a *francés*. También usamos el método `ResponseBuilder.header()` para asignar un valor concreto a otra cabecera.
❹ Finalmente, creamos y devolvemos el objeto `Response` usando el método `ResponseBuilder.build()`.

Un detalle que es interesante destacar en este código es que **no** indicamos ningún valor para el `Content-Type` de la respuesta. Debido a que ya hemos especificado esta información en la anotación `@Produces` del método, el *runtime* de JAX-RS devolverá el valor adecuado del *media type* de la respuesta por nosotros.

## Inclusión de *cookies* en la respuesta

JAX-RS proporciona la clase `javax.ws.rs.core.NewCookie`, que utilizaremos para crear nuevos valores de *cookies* y enviarlos en las respuestas.

Para poder incluir las *cookies* en nuestro objeto `Response`, primero crearemos las instancias correspondientes de tipo `NewCookie` las pasaremos al método `ResponseBuilder.cookie()`. Por ejemplo:

```
@Path("/myservice")
public class MyService {
    @GET
    public Response get() {
        NewCookie cookie = new NewCookie("nombre", "pepe"); ❶
        ResponseBuilder builder = Response.ok("hola", "text/plain"); ❷
        return builder.cookie(cookie).build();
```

```
}
```

❶ Creamos una nueva *cookie* con el nombre de clave `nombre` y le asignamos el valor `pepe`

❷ En este caso, al no indicar el tipo mime de la respuesta con la anotación @Produces, necesitamos indicarlo (en este caso como un parámetro del método `ok()`)

## El tipo enumerado de códigos de estado

JAX-RS proporciona el tipo enumerado `javax.ws.rs.core.Status` para representar códigos de respuesta específicos:

```
public enum Status {
   OK(200, "OK"),
   CREATED(201, "Created"),
   ACCEPTED(202, "Accepted"),
   NO_CONTENT(204, "No Content"),
   MOVED_PERMANENTLY(301, "Moved Permanently"),
   SEE_OTHER(303, "See Other"),
   NOT_MODIFIED(304, "Not Modified"),
   TEMPORARY_REDIRECT(307, "Temporary Redirect"),
   BAD_REQUEST(400, "Bad Request"),
   UNAUTHORIZED(401, "Unauthorized"),
   FORBIDDEN(403, "Forbidden"),
   NOT_FOUND(404, "Not Found"),
   NOT_ACCEPTABLE(406, "Not Acceptable"),
   CONFLICT(409, "Conflict"),
   GONE(410, "Gone"),
   PRECONDITION_FAILED(412, "Precondition Failed"),
   UNSUPPORTED_MEDIA_TYPE(415, "Unsupported Media Type"),
   INTERNAL_SERVER_ERROR(500, "Internal Server Error"),
   NOT_IMPLEMENTED(501, "Not Implemented"),
   SERVICE_UNAVAILABLE(503, "Service Unavailable");
   public enum Family {
      INFORMATIONAL, SUCCESSFUL, REDIRECTION,
      CLIENT_ERROR, SERVER_ERROR, OTHER
   }

   public Family getFamily()
   public int getStatusCode()
   public static Status fromStatusCode(final int statusCode)
}
```

Cada valor del tipo `Status` se asocia con una familia específica de códigos de respuesta HTTP. Estas familias se identifican por el enumerado `Status.Family`:

- Los códigos en el rango del **100** se consideran **informacionales**
- Los códigos en el rango del **200** se consideran **exitosos**
- Los códigos en el rango del **300** son códigos con éxito pero dentro de la categoría **redirección**
- Los códigos de error pertenecen a los ragos 400 y 500. En el rango de **400** se consideran **errores del cliente** y en el rango de **500** son **errores del servidor**

Tanto el método `Response.status()` como `ResponseBuilder.status()` pueden aceptar un valor enumerado de tipo `Status`. Por ejemplo:

```
@DELETE
Response delete() {
   ...
   return Response.status(Status.GONE).build();
}
```

En este caso, estamos indicando al cliente que lo que queremos borrar ya no existe (410).

### La clase javax.ws.rs.core.GenericEntity

Cuando estamos creando objetos de tipo `Response`, se nos plantea un problema cuando queremos devolver tipos genéricos, ya que el manejador JAXB necesita extraer la información del tipo parametrizado de la respuesta en tiempo de ejecución. Para estos casos, JAX-RS proporciona la clase `javax.ws.rs.core.GenericEntity`. Veamos su uso con un ejemplo:

```
@GET
@Produces("application/xml")
public Response getListaClientes() {
   List<Cliente> list = new ArrayList<Cliente>();
   list.add(new Cliente(...));

   GenericEntity entity =
      new GenericEntity<List<Cliente>>(list){}; ❶

   return Response.ok(entity).build();
}
```

❶ La clase `GenericEntity` es también una clase genérica. Lo que hacemos es crear una clase anónima que extiende `GenericEntity`, inicializando la "plantilla" de `GenericEntity` con el tipo genérico que estemos utilizando.

## 3.6. Manejadores de excepciones

Vamos a explicar cómo podemos tratar las excepciones en nuestros servicios RESTful.

Los errores pueden enviarse al cliente, bien creando y devolviendo el objeto `Response` adecuado, o lanzando una excepción. Podemos lanzar cualquier tipo de excepción, tanto las denominadas *checked* (clases que heredan de `java.lang.Exception`), como las excepciones *unchecked* (clases que extienden `java.lang.RuntimeException`). Las excepciones generadas son manejadas por el *runtime* de JAX-RS si tenemos registrado un *mapper* de excepciones. Dichos *mappers* (o mapeadores) de excepciones pueden convertir una excepción en una respuesta HTTP. Si las excepciones no están gestionadas por un *mapper*, éstas se propagan y se gestionan por el contenedor (de *servlets*) en el que se está ejecutando JAX-RS. JAX-RS proporciona también la clase `javax.ws.rs.WebApplicationException`. Esta excepción puede lanzarse por el código de nuestra aplicación y será procesado automáticamente por JAX-RS sin necesidad de disponer de forma explícita de ningún *mapper*. Vamos a ver cómo utilizar esta clase.

## La clase javax.ws.rs.WebApplicationException

JAX-RS incluye una excepción *unchecked* que podemos lanzar desde nuestra aplicación RESTful (ver la documentación del http://docs.oracle.com/javaee/7/api/javax/ws/rs/WebApplicationException.html [API]). Esta excepción se puede pre-inicializar con un objeto `Response`, o con un código de estado particular:

**Clase javax.ws.rs.WebApplicationException**

```
public class WebApplicationException extends RuntimeException {
   //Constructores
   public WebApplicationException() {...}
   public WebApplicationException(Response response) {...} ❶
   public WebApplicationException(int status) {...}
   public WebApplicationException(Response.Status status) {...} ❷
   public WebApplicationException(String message) ❸
   public WebApplicationException(Throwable cause) {...}
   public WebApplicationException(Throwable cause, Response response) {...}
   public WebApplicationException(Throwable cause, int status) {...}
   public WebApplicationException(Throwable cause, Response.Status status)
  {...}

   public Response getResponse() {...]
}
```

❶ Podemos crear una instancia a partir de un objeto Response

❷ Creación de una instancia a partir de un código de estado

❸ Creación de una instancia a partir de un *String*. Por defecto se incluye el código de estado 500. El *String* que se pasa como parámetro se almacena para su posterior recuperación a través del mensaje `getMessage()

Cuando JAX-RS detecta que se ha lanzado la excepción `WebApplicationException`, la captura y realiza una llamada al método `WebApplicationException.getResponse()` para obtener un objeto `Response` que enviará al cliente. Si la aplicación ha inicializado la excepción `WebApplicationException` con un código de estado, o un objeto `Response`, dicho código o `Response` se utilizarán para crear la respuesta HTTP real. En otro caso, la excepción `WebApplicationException` devolverá el cliente el código de respuesta "500 Internal Server Error".

Por ejemplo, supongamos que tenemos un servicio web que permite a los usuarios solicitar información de nuestros clientes, utilizando una representación XML:

```
@Path("/clientes")
public class ClienteResource {
   @GET
   @Path("{id}")
   @Produces("application/xml")
   public Cliente getCliente(@PathParam("id") int id) {
      Cliente cli = recuperarCliente(id);
      if (cli == null) {
           throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
      return cli;
   }
```

```
}
```

En este ejemplo, si no encontramos una instancia de `Cliente` con el `id` proporcionado, lanzaremos una `WebApplicationException` que provocará que se le envíe al cliente como código de respuesta "404 Not Found".

## Mapeado de excepciones

Normalmente, las aplicaciones tienen que "tratar" con multitud de excepciones lanzadas desde nuestro código de aplicación o por *frameworks* de terceros. Dejar que el *servlet* JAX-RS que reside en el servidor maneje la excepción no nos proporciona demasiada flexibilidad. Si capturamos y "redirigimos" todas estas excepciones a `WebApplicationException` podría resultar bastante tedioso. De forma alternativa, podemos implementar y registrar instancias de `javax.ws.rs.ext.ExceptionMapper`. Estos objetos "saben" cómo mapear una excepción lanzada por la aplicación a un objeto `Response`:

```
public interface ExceptionMapper<E extends Throwable> {
{
   Response toResponse(E exception);
}
```

Las clases que implementan la interfaz `ExceptionMapper<T>` son "proveedores de mappings de excepciones" (*Exception Mapping Providers*) y mapean una excepción *runtime* o *checked* a una instancia de `Response`.

Cuando un recurso JAX-RS lanza una excepción para la que existe un proveedor de mapping de excepciones, éste último se utiliza para devolver una instancia de `Response`. Esta respuesta resultante se procesa como si hubiese sido generada por el recurso.

Por ejemplo, una excepción bastante utilizada por aplicaciones de bases de datos que utilizan JPA (**J**ava **P**ersistence **A**pi) es `javax.persistence.EntityNotFoundException`. Esta excepción se lanza cuando JPA no puede encontrar un objeto particular en la base de datos. En lugar de escribir código que maneje dicha excepción de forma explícita, podemos escribir un `ExceptionMapper` para que gestione dicha excepción por nosotros. Veámos cómo:

```
@Provider ❶
public class EntityNotFoundMapper
         implements ExceptionMapper<EntityNotFoundException> {  ❷
   public Response toResponse(EntityNotFoundException e) {  ❸
      return Response.status(Response.Status.NOT_FOUND).build();  ❹
   }
}
```

❶ Nuestra implementación de `ExceptionMapper` debe anotarse con `@Provider`. Esto le indica al *runtime* de JAX-RS que esta clase es un componente REST.
❷ La clase que implementa `ExceptionMapper` debe proporcionar el tipo que se quiere parametrizar. JAX-RS utiliza esta información para emparejar las excepciones de tipo `EntityNotFoundException` con nuestra clase `EntityNotFoundMapper`
❸ El método `toResponse()` recibe la excepción lanzada y
❹ crea un objeto `Response` que se utilizará para construir la respuesta HTTP

Otro ejemplo es la excepción *EJBException*, lanzada por aplicaciones que utilizan EJBs.

## Jerarquía de excepciones

JAX-RS 2.0 proporciona una jerarquía de excepciones para varias condiciones de error para las peticiones HTTP. La idea es que, en lugar de crear una instancia de `WebApplicationException` e inicializarla con un código de estado específico, podemos uilizar en su lugar una de las excepciones de la jeraquía (clases que heredan de la clase *WebApplicationException*). Por ejemplo, podemos cambiar el código anterior que utilizaba `WebApplicationException`, y en su lugar, usar `javax.ws.rs.NotFoundException`:

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Cliente getCliente(@PathParam("id") int id) {
     Cliente cli = recuperarCliente(id);
     if (cli == null) {
          throw new NotFoundException();
     }
     return cli;
  }
}
```

Al igual que el resto de excepciones de la jerarquía, la excepción `NotFoundException` hereda de `WebApplicationException`. La siguiente tabla muestra algunas excepciones que podemos utilizar, todas ellas del paquete `javax.ws.rs`. Las que incluyen un código de estado en el rango de 400, son subclases de `ClientErrorException`, y como ya hemos indicado, representan errores en las peticiones. Las que presentan un código de estado en el rango de 500, son subclases de `ServerErrorException`, y representan errores del servidor.

### Table 5. Jerarquía de excepciones JAX-RS:

| Excepción | Código de estado | Descripción |
|---|---|---|
| BadRequestException | 400 | Mensaje mal formado |
| NotAuthorizedException | 401 | Fallo de autenticación |
| ForbiddenException | 403 | Acceso no permitido |
| NotFoundException | 404 | No se ha podido encontrar el recurso |
| NotAllowedException | 405 | Método HTTP no soportado |
| NotAcceptableException | 406 | *Media type* solicitado por el cliente no soportado (cabecera **Accept** de la peticion) |
| NotSupportedException | 415 | El cliente ha incluido un *Media type* no soportado (cabecera **Content-Type** de la peticion) |

| Excepción | Código de estado | Descripción |
|---|---|---|
| InternalServerErrorException | 500 | Error general del servidor |
| ServiceUnavailableException | 503 | El servidor está ocupado o temporalmente fuera de servicio |

- La excepción `BadRequestException` se utiliza cuando el cliente envía algo al servidor que éste no puede interpretar. Ejemplos de escenarios concretos que provocan que el *runtime* JAX-RS son: cuando una petición PUT o POST contiene un cuerpo del mensaje con un documento XML o JSON mal formado, de forma que falle el "parsing" del documento, o cuando no puede convertir un valor especificado en la cabecera o *cookie* al tipo deseado. Por ejemplo:

```
@HeaderParam("Cabecera-Particular") int cabecera;
@CookieParam("miCookie") int cookie;
```

  Si el valor de la cabecera HTTP de la petición o el valor `miCookie` no puede convertirse en un entero, se lanzará la excepción `BadRequestException`.

- La excepción `NotAuthorizedException` (código **401**), se usa cuando queremos escribir nuestros propios protocolos de autorización, y queremos indicar al cliente que éste necesita autenticarse con el servidor.
- La excepción `ForbiddenException` (código **403**),se usa generalmente cuando el cliente realiza una invocación para la que no tiene permisos de acceso. Esto ocurre normalmente debido a que el cliente no tiene el rol requerido.
- La excepción `NotFoundException` (código **404**), se usa cuando queremos comunicar al cliente que el recurso que está solicitando no existe. Esta excepción también se generará de forma automática por el *runtime* de JAX-RS cuando a éste no le sea posible inyectar un valor en un @PathParam, @QueryParam, o @MatrixParam. Al igual que hemos comentado para `BadRequestException` esto puede ocurrir si intentamos convertir el valor del parámetro a un tipo que no admite esta conversión.
- La excepción `NotAllowedException` (código **405**), se usa cuando el método HTTP que el cliente está intentando invocar no está soportado por el recurso al que el cliente está accediendo. El *runtime* de JAX-RS lanza automáticamente esta excepción si no encuentra ningún método que pueda "emparejar" con el método HTTP invocado.
- La excepción `NotAcceptableException` (código **406**), se usa cuando un cliente está solicitando un formato específico a través de la cabecera `Accept`. El *runtime* de JAX-RS lanza automáticamente esta excepción si no hay un método con una anotación @Produces que sea compatible con la cabecera `Accept` del cliente.
- La excepción `NotSupportedException` (código **415**),se usa cuando un cliente está enviando una representación que el servidor no "comprende". El *runtime* de JAX-RS lanza automáticamente esta excepción si no hay un método con una anotación @Consumes que coincida con el valor de la cabecera `Content-Type` de la petición.
- La excepción `InternalServerErrorException` (código **500**),es una excepción de propósito general lanzada por el servidor. Si en nuestra aplicación queremos lanzar esta excepción deberíamos hacerlo si se ha producido una condición de error que realmente no "encaja" en ninguna de las situaciones que hemos visto. El *runtime* de JAX-RS lanza automáticamente esta excepción si falla un `MessageBodyWriter` o si se lanza alguna excepción desde algún `ExceptionMapper`.

- La excepción `ServiceUnavailableException` (código **503**),se usa cuando el servidor está ocupado o temporalmente fuera de servicio. En la mayoría de los casos, es suficiente con que el cliente vuelva a intentar realizar la llamada un tiempo más tarde.

## 3.7. Ejercicios

### Servicio REST ejemplo

Para familiarizarnos con las el uso de diferentes manejadores de contenidos y manejo de excepciones proporcionamos el módulo el **MÓDULO s3-ejemplo-rest**, con la implementación de un servicio rest sencillo que podéis probar con la herramienta **postman**.

En el directorio `src/main/resources` de dicho módulo tenéis un fichero de texto con las instrucciones (*instrucciones.txt*) para construir, desplegar y probar la aplicación de ejemplo.

### Plantillas que se proporcionan

Para esta sesión proporcionamos un proyecto como plantilla con el nombre `s3-filmoteca` que tendrás utilizar como punto de partida para aplicar lo que hemos aprendido en esta sesión.

Se trata de una implementación parcial para gestionar una filmoteca con información de películas y actores.

La estructura lógica del proyecto proporcionado es la siguiente:

- Paquete `org.expertojava.domain`: es la capa que contiene los objetos del dominio de la aplicación. Por simplicidad, no usamos una base de datos real, sino que trabajamos con datos en memoria.
- Paquete `org.expertojava.service`: contiene la implementación de los servicios de nuestra aplicación, que serán accedidos desde la capa rest
- Paquete `org.expertojava.rest`: constituye la capa rest de nuestra aplicación. Esta capa es cliente de la capa de servicios.

En la carpeta `src/main/resources` tenéis un fichero de texto (*instrucciones.txt*) con información detallada sobre el API rest implementado.

### Uso de JAXB (0,5 puntos)

Utiliza las anotaciones JAXB oportunas para realizar el serializado de las entidades java a xml y json, de forma que la lista de películas de la filmoteca en formato xml sea:

#### Petición rest: GET /peliculas

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<peliculas>
   <pelicula duracion="120" estreno="2015-12-08T00:00:00+01:00" id="1">
     <director>Stanley Kubrick</director>
     <titulo>El resplandor</titulo>
   </pelicula>
   <pelicula duracion="155" estreno="2015-18-07T00:00:00+01:00" id="2">
     <director>Director 2</director>
     <titulo>Pelicula 2</titulo>
   </pelicula>
</peliculas>
```

Por defecto, JAXB serializa los tipos `Date` con el formato ISO 8061[7] para los contenidos en el cuerpo de la petión/respuesta. Dicho formato es YYYY-MM-DD (seguido de HH:MM:SS). Por otro lado, si añadimos actores a las películas, éstos deberán mostrarse bajo la etiqueta *<actores>*, tal y como mostramos en el siguiente ejemplo.

Los datos mostrados para una película en formato xml tienen que presentar el siguiente aspecto:

source,java] .Petición rest: GET /peliculas/1

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<pelicula duracion="120" estreno="2015-12-08T00:00:00+01:00" id="1">
    <actores>
        <actor nombre="Jack Nicholson" personaje="Jack Torrance"/>
    </actores>
    <director>Stanley Kubrick</director>
    <titulo>El resplandor</titulo>
</pelicula>
```

Puedes comprobar si has hecho bien el ejercicio utilizando Postman.

## Uso de manejadores de contenidos y clase Response (0,75 puntos)

**Implementa una nueva petición POST** que reciba los datos de una nueva película desde un formulario. Recuerda que los datos de un formulario pueden ponerse en el cuerpo de la petición HTTP como pares `nombre=valor` separados por `&`. Los espacios en blanco se codifican como `%20`. No es necesario poner comillas. Por ejemplo:

```
nombre1=valor%20con%20espacios&nombre2=valor
```

Se proporciona el fichero index.html con un formulario para utilizarlo como alternativa a postman. Para acceder al formulario usaremos *http://localhost:8080/s3-filmoteca/*

**Modifica las peticiones POST** sobre `películas` y `actores` de forma que devuelvan en la cabecera Location la URI del nuevo recurso creado. Por ejemplo

```
http://localhost:8080/s3-filmoteca/peliculas/3
```

en el caso de una nueva película, y

```
http://localhost:8080/s3-filmoteca/peliculas/3/actores/Sigourney%20Weaver
```

si por ejemplo hemos añadido un nuevo actor a la película con id=3.

**Modifica los métodos GET** para que devuelvan el estado: 204 No Content en los casos en los que la película y/o actor consultado no exista.

[7] https://es.wikipedia.org/wiki/ISO_8601

**Modifica los métodos GET** para que devuelvan el estado: 404 Not Found, en los casos en los que las listas de películas y/o actores estén vacías.

**Implementa** el código para añadir un nuevo actor. Tendrás que obtener la información de la película y nombre del actor de la URI de la petición.

Puedes comprobar si has hecho bien el ejercicio utilizando Postman.

## Manejo de excepciones (0,75 puntos)

**Modifica el método** `addPelicula` de la capa de servicio (paquete *org.expertojava.service*) para que lance una excepción de tipo *ServiceException* con el mensaje "El título de la película no puede ser nulo ni vacío" cuando se intente añadir una película con un título con valor *null*, o vacío.

El método `addPelicula` debe lanzar también una excepción de tipo *ServiceException* con el mensaje "La película ya existe", cuando se intente añadir una película con un título que ya existe.

**Modifica el método** `addActor` de la capa de servicio para que lance las excepciones de tipo *ServiceException* con los mensajes "El título de la película no puede ser nulo ni vacío" cuando se intente añadir un actor a una película cuyo título no existe, o bien el mensaje "EL actor ya existe" si intentamos añadir un actor a una película que ya habíamos añadido previamente.

**Implementa** un *mapper* para capturar las excepciones de la capa de servicio, de forma que se devuelva el estado 500, "Internal Server error", y como entidad del cuerpo de la respuesta el mensaje asociado a las excepciones generadas por el servicio (*"El título de la película no puede ser nulo ni vacío"*, *"EL actor ya existe"*, …). La nueva clase pertencerá a la capa "rest". Puedes ponerle el nombre *ServiceExceptionMapper*.

Puedes comprobar si has hecho bien el ejercicio utilizando Postman.

# 4. HATEOAS y Seguridad

En esta sesión trataremos uno de los principios REST de obligado cumplimiento para poder hablar de un servicio RESTful. Nos referimos a HATEOAS. Hasta ahora hemos visto cómo los clientes pueden cambiar el estado de los recursos (el nombre del recurso se especifica en la URI de la petición) a través de los contenidos del cuerpo del mensaje, o utilizando parámetros, o cabeceras de petición. A su vez, los servicios comunican el estado resultante de la petición a los clientes a través del contenido del cuerpo del mensaje, códigos de respuesta, y cabeceras de respuesta. Pues bien, teniendo en cuenta lo anterior, HATEOAS hace referencia a que, cuando sea necesario, también deben incluirse los enlaces a los recursos (URI) en el cuerpo de la respuesta (o en las cabeceras), para así poder recuperar el recurso en cuestión, o los recursos relacionados.

En esta sesión también explicaremos algunos conceptos básicos para poder dotar de seguridad a nuestros servicios REST

## 4.1. ¿Qué es HATEOAS?

Comúnmente se hace referencia a Internet como "la *Web*" (*web* significa red, telaraña), debido a que la información está interconectada mediante una serie de hiperenlaces embebidos dentro de los documentos HTML. Estos enlaces crean una especie de "hilos" o "hebras" entre los sitios web relacionados en Internet. Una consecuencia de ello es que los humanos pueden "navegar" por la Web buscando elementos de información relacionados de su interés, haciendo "click" en los diferentes enlaces desde sus navegadores. Los motores de búsqueda pueden "trepar" o "desplazarse" por estos enlaces y crear índices enormes de datos susceptibles de ser "buscados". Sin ellos, Internet no podría tener la propiedad de ser escalable. No habría forma de indexar fácilmente la información, y el registro de sitios web sería un proceso manual bastante tedioso.

Además de los enlaces (*links*), otra característica fundamental de Internet es HTML. En ocasiones, un sitio web nos solicita que "rellenemos" alguna información para comprar algo o registrarnos en algún servicio. El servidor nos indica a nosotros como clientes qué información necesitamos proporcionar para completar una acción descrita en la página web que estamos viendo. El navegador nos muestra la página web en un formado que podemos entender fácilmente. Nosotros leemos la página web y rellenamos y enviamos el formulario. Un formulario HTML es un formato de datos interesante debido a que auto-describe la interacción entre el cliente y el servidor.

El principio arquitectónico que describe el proceso de enlazado (*linking*) y el envío de formularios se denomina **HATEOAS**. Las siglas del término HATEOAS significan **H**ypermedia **A**s **T**he **E**ngine **O**f **A**pplication **S**tate (es decir, el uso de Hipermedia como mecanismo de máquina de estados de la aplicación). La idea de HATEOAS es que el formato de los datos proporciona información extra sobre cómo cambiar el estado de nuestra aplicación. En la Web, los enlaces HTML nos permiten cambiar el estado de nuestro navegador. Por ejemplo cuando estamos leyendo una página web, un enlace nos indica qué posibles documentos (estados) podemos ver a continuación. Cuando hacemos "click" sobre un enlace, el estado del navegador cambia al visitar y mostrar una nueva página web. Los formularios HTML, por otra parte, nos proporcionan una forma de cambiar el estado de un recurso específico de nuestro servidor. Por último, cuando compramos algo en Internet, por ejemplo, estamos creando dos nuevos recursos en el servicio: una transacción con tarjeta de crédito y una orden de compra.

## 4.2. HATEOAS y Servicios Web

Cuando aplicamos HATEOAS a los servicios web la idea es incluir enlaces en nuestros documentos XML o JSON. La mayoría de las aplicaciones RESTful basadas en XML utilizan el formato **Atom Syndication Format**[8] para implementar HATEOAS.

### Enlaces Atom

Los enlaces Atom constituyen un mecanismo estándar para incluir enlaces (*links*) en nuestros documentos XML. Veamos un ejemplo:

```
<clientes>
   <link rel="next"
         href="http://ejemplo.com/clientes?inicio=2&total=2"
        type="application/xml"/>
   <cliente id="123">
      <nombre>Juan Garcia</nombre>
   </cliente>
   <cliente id="332">
      <nombre>Pablo Bozo</nombre>
   </cliente>
</clientes>
```

El documento anterior representa una lista de clientes, y el elemento `<link>` , que contiene un enlace, indica la forma de obtener los siguientes clientes de la lista.

Un enlace Atom es simplemente un elemento XML (elemento `<link>`) con unos atributos específicos.

- El atributo `rel` Se utiliza para indicar la relación del enlace con el elemento XML en el que anidamos dicho enlace. Es el **nombre lógico** utilizado para referenciar el enlace. Este atributo tiene el mismo significado para la URL que estamos enlazando, que la etiqueta HTML <a> tiene para la URL sobre la que estamos haciendo *click* con el ratón en el navegador. Si el enlace hace referencia al propio elemento XML en el que incluimos el enlace, entonces asignaremos el valor del atributo `self`
- El atributo `href` es la **URL** a la que podemos acceder para obtener nueva información o cambiar el estado de nuestra aplicación
- El atributo `type` indica el *media type* asociado con el recurso al que apunta la URL

Cuando un cliente recibe un documento con enlaces Atom, éste busca la relación en la que está interesado (atributo `rel`) e invoca la URI indicada en el atributo `href`.

### Ventajas de utilizar HATEOAS con Servicios Web

Resulta bastante obvio por qué los enlaces y los formularios tienen mucho que ver en la prevalencia de la Web. Con un navegador, tenemos una "ventana" a todo un mundo de información y servicios. Las máquinas de búsqueda "rastrean" Internet e indexan sitios web para que todos los datos estén al alcance de nuestros "dedos". Esto es posible debido a que la Web es auto-descriptiva. Cuando accedemos a un documento, conocemos cómo recuperar información adicional "siguiendo" los enlaces situados en dicho documento. Por ejemplo,

[8] http://www.w3.org/2005/Atom

conocemos cómo realizar una compra en Amazon, debido a que los formularios HTML nos indican cómo hacerlo.

Cuando los clientes son "máquinas" en lugar de personas (los servicios Web también se conocen como "Web para máquinas", frente a la "Web para humanos" proporcionada por el acceso a un servidor web a través de un navegador) el tema es algo diferente, puesto que las máquinas no pueden tomar decisiones "sobre la marcha", cosa que los humanos sí pueden hacer. Las máquinas requieren que los programadores les digan cómo interpretar los datos recibidos desde un servicio y cómo realizar transiciones entre estados como resultado de las interacciones entre clientes y servidores.

En este sentido, HATEOAS proporciona algunas ventajas importantes para contribuir a que los clientes sepan cómo utilizar los servicios a la vez que acceden a los mismos. Vamos a comentar algunas de ellas.

### Transparencia en la localización

En un sistema RESTful, gracias a HATEOAS, sólo es necesario hacer públicas unas pocas URIs. Los servicios y la información son representados con enlaces que están "embebidos" en los formatos de los datos devueltos por las URIs públicas. Los clientes necesitan conocer los nombres lógicos de los enlaces para "buscar" a través de ellos, pero no necesitan conocer las ubicaciones reales en la red de los servicios a los que acceden.

Los enlaces proporcionan un nivel de indirección, de forma que los servicios subyacentes pueden cambiar sus localizaciones en la red sin alterar la lógica ni el código del cliente.

### Desacoplamiento de los detalles de la interacción

Consideremos una petición que nos devuelve una lista de clientes en una base de datos: `GET /clientes`. Si nuestra base de datos tiene miles de datos, probablemente no querremos devolver todos ellos de una sóla vez. Lo que podemos hacer es definir una vista en nuestra base de datos utilizando parámetros de consulta, por ejemplo:

```
/customers?inicio={indiceInicio}&total={numeroElementosDevueltos}
```

El parámetro `inicio` identifica el índice inicial de nuestra lista de clientes. El parámetro `total` especifica cuántos clientes queremos que nos sean devueltos como respuesta.

Lo que estamos haciendo, en realidad, es incrementar la cantidad de conocimiento que el cliente debe tener predefinido para interactuar con el servicio (es decir, no sólo necesita saber la URI, sino además conocer la existencia de estos parámetros). Supongamos que en el futuro, el servidor decide que necesita cambiar la forma en la que se accede al número de datos solicitados por el cliente. Si el servidor cambia la interfaz, los clientes "antiguos" dejarán de funcionar a menos que cambien su código.

En lugar de publicar la interfaz REST anterior para obtener datos de los clientes, podemos incluir dicha información en el documento de respuesta, por ejemplo:

```
<clientes>
   <link rel="next"
         href="http://ejemplo.com/clientes?inicio=2&total=2"
        type="application/xml"/>
   <cliente id="123">
```

```
        <nombre>Juan Garcia</nombre>
    </cliente>
    <cliente id="332">
        <nombre>Pablo Bozo</nombre>
    </cliente>
</clientes>
```

Cuando incluimos un enlace Atom en un documento, estamos asignando un **nombre lógico** a una **transición de estados**. En el ejemplo anterior, la transición de estados es el siguiente conjunto de clientes a los que podemos acceder. En lugar de tener que recordar cuáles son los parámetros de la URI que tenemos que utilizar en la siguiente invocación para obtener más clientes, lo único que tenemos que hacer es "seguir" el enlace proporcionado. El cliente no tiene que "contabilizar" en ningún sitio la interacción, ni tiene que recordar qué "sección" de la base de datos estamos consultando actualmente.

Además, el XML devuelto es auto-contenido. ¿Qué pasa si tenemos que "pasar" este documento a un tercero? Tendríamos que "decirle" que se trata de una vista parcial de la base de datos y especificar el ídice de inicio. Al incluir el enlace en el documento, ya no es necesario proporcionar dicha información adicional, ya que forma parte del propio documento

### Reducción de errores de transición de estados

Los enlaces no se utilizan solamente como un mecanismo para agregar información de "navegación". También se utilizan para cambiar el estado de los recursos. Pensemos en una aplicación de comercio web a la que podemos acceder con la URI `pedidos/333`:

```
<pedido id="333">
    <cliente id="123">...</cliente>
    <importe>99.99</importe>
    <lineas-pedido>
      ...
    </lineas-pedido>
</pedido>
```

Supongamos que un cliente quiere cancelar su pedido. Podría simplemente invocar la petición HTTP `DELETE /pedidos/333`. Esta no es siempre la mejor opción, ya que normalmente el sistema necesitará "retener" el pedido para propósitos de almacenaje. Por ello, podríamos considerar una nueva representación del pedido con un elemento `cancelado` a true:

```
PUT /pedidos/333 HTTP/1.1
Content-Type: application/xml
<pedido id="333">
    <cliente id="123">...</cliente>
    <importe>99.99</importe>
    <cancelado>true</cancelado>
    <lineas-pedido>
        ...
    </lineas-pedido>
</pedido>
```

Pero, ¿qué ocurre si el pedido no puede cancelarse? Podemos tener un cierto estado en nuestro proceso de pedidos en donde esta acción no está permitida. Por ejemplo, si el pedido

ya ha sido enviado, entonces no puede cancelarse. En este caso, realmente no hay nigún código de estado HTTP de respuesta que represente esta situación. Una mejor aproximación es incluir un enlace para poder realizar la cancelación:

```
<pedido id="333">
   <cliente id="123">...</cliente>
   <importe>99.99</importe>
   <cancelado>false</cancelado>
   <link rel="cancelar"
        href="http://ejemplo.com/pedidos/333/cancelado"/>
   <lineas-pedido>
           ...
   </lineas-pedido>
</pedido>
```

El cliente podría invocar la orden: `GET  /pedidos/333` y obtener el documento XML que representa el pedido. Si el documento contiene el enlace **cancelar**, entonces el cliente puede cambiar el estado del pedido a "cancelado" enviando una orden PUT vacía a la URI referenciada en el enlace. Si el documento no contiene el enlace, el cliente sabe que esta operación no es posible. Esto permite que el servicio web controle en tiempo real la forma en la que el cliente interactua con el sistema.

### Enlaces en cabeceras frente a enlaces Atom

Una alternativa al uso de enlaces Atom en el cuerpo de la respuesta, es utilizar enlaces en las cabeceras de la respuesta (http://tools.ietf.org/html/rfc5988). Vamos a explicar ésto con un ejemplo.

Consideremos el ejemplo de cancelación de un pedido que acabamos de ver. En lugar de utilizar un enlace Atom para especificar si se permite o no la cancelación del pedido, podemos utilizar la cabecera `Link` (es uno de los posibles campos que podemos incluir como cabecera en una respuesta HTTP)). De esta forma, si un usuario envía la petición `GET /pedidos/333`, recibirá la siguiente respuesta HTTP:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Link: <http://ejemplo.com/pedidos/333/cancelado>; rel=cancel

<pedido id="333">
   ...
</pedido>
```

La cabecera `Link` tiene las mismas características que un enlace Atom. La URI está entre los signos `<` y `>` y está seguida por uno o más atributos delimitados por `;`. El atributo `rel` es obligatorio y tiene el mismo significado que el correspondiente atributo Atom com el mismo nombre. En el ejemplo no se muestra, pero podríamos especificar el *media type* utilizando el atributo `type`.

## 4.3. HATEOAS y JAX-RS

JAX-RS no proporciona mucho soporte para implementar HATEOAS. HATEOAS se define por la aplicación, por lo que no hay mucho que pueda aportar ningún *framework*. Lo que sí

proporciona JAX-RS son algunas clases que podemos utilizar para construir las URIs de los enlaces HATEOAS.

## Construcción de URIs con UriBuilder

Una clase que podemos utilizar es `javax.ws.rs.core.UriBuilder`. Esta clase nos permite construir URIs elemento a elemento, y también permite incluir plantillas de parámetros (segmentos de ruta variables).

#### Clase UriBuilder: métodos para instanciar objetos de la clase

```
public abstract class UriBuilder {
  public static UriBuilder fromUri(URI uri)
                throws IllegalArgumentException
  public static UriBuilder fromUri(String uri)
                throws IllegalArgumentException
  public static UriBuilder fromPath(String path)
                throws IllegalArgumentException
  public static UriBuilder fromResource(Class<?> resource)
                throws IllegalArgumentException
  public static UriBuilder fromLink(Link link)
                throws IllegalArgumentException
```

Las **instancias** de `UriBuilder` se obtienen a partir de métodos estáticos con la forma `fromXXX()`. Podemos inicializarlas a partir de una URI, una cadena de caracteres, o la anotación @Path de una clase de recurso.

Para extraer, modificar y/o componer una URI, se pueden utilizar métodos como:

#### Clase UriBuilder: métodos para "manipular" las URIs

```
public abstract UriBuilder clone(); // crea una copia

// crea una copia con la información de un objeto URI
public abstract UriBuilder uri(URI uri)
                  throws IllegalArgumentException;

// métodos para asignar/modificar valores de
// los atributos de los objetos UriBuilder
public abstract UriBuilder scheme(String scheme)
                  throws IllegalArgumentException;
public abstract UriBuilder userInfo(String ui);
public abstract UriBuilder host(String host)
                  throws IllegalArgumentException;
public abstract UriBuilder port(int port)
                  throws IllegalArgumentException;
public abstract UriBuilder replacePath(String path);

// métodos que añaden elementos a nuestra URI
public abstract UriBuilder path(String path)
public abstract UriBuilder segment(String... segments)
public abstract UriBuilder matrixParam(String name,
                                       Object... values)
public abstract UriBuilder queryParam(String name,
                                      Object... values)
```

```
// método que instancia el valor de una plantilla de la URI
public abstract UriBuilder resolveTemplate(String name,
                                            Object value)
#...
```

Los métodos `build()` construyen la URI. Ésta puede contener plantillas de parámetros ( segmentos de ruta variables), que deberemos inicializar utilizando pares nombre/valor, o bien una lista de valores que reemplazarán a los parámetros de la plantilla en el orden en el que aparezcan.

### Clase UriBuilder: métodos buildXXX() para construir las URIs

```
public abstract URI buildFromMap(Map<String, ? extends Object> values)
                throws IllegalArgumentException, UriBuilderException;

public abstract URI build(Object... values)
                throws IllegalArgumentException, UriBuilderException;
...
}
```

Veamos algún ejemplo que muestra cómo crear, inicializar, componer y construir una URI utilizando un `UriBuilder`:

```
UriBuilder builder = UriBuilder.fromPath("/clientes/{id}");
builder.scheme("http")
       .host("{hostname}")
       .queryParam("param={param}");
```

Con este código, estamos definiendo una URI como:

```
http://{hostname}/clientes/{id}?param={param}
```

Puesto que tenemos plantillas de parámetros, necesitamos inicializarlos con valores que pasaremos como argumentos para crear la URI final. Si queremos reutilizar la URI que contiene las plantillas, deberíamos realizar una llamada a `clone()` antes de llamar al método `build()`, ya que éste reemplazará los parámetros de las plantillas en la estructura interna del objeto:

```
UriBuilder clone = builder.clone();
URI uri = clone.build("ejemplo.com", "333", "valor");
```

El código anterior daría lugar a la siguiente URI:

```
http://ejemplo.com/clientes/333?param=valor
```

También podemos definir un objeto de tipo `Map` que contenga los valores de las plantillas:

```
Map<String, Object> map = new HashMap<String, Object>();
```

```
map.put("hostname", "ejemplo.com");
map.put("id", 333);
map.put("param", "valor");
UriBuilder clone = builder.clone();
URI uri = clone.buildFromMap(map);
```

Otro ejemplo interesante es el de crear una URI a partir de las expresiones @Path definidas en las clases JAX-RS anotadas. A continuación mostramos el código:

```
@Path("/clientes")
public class ServicioClientes {

   @Path("{id}")
   public Cliente getCliente(@PathParam("id") int id) {...}
}
```

Podemos referenciar esta clase y el método `getCliente()` a través de la clase `UriBuilder` de la siguiente forma:

```
UriBuilder builder = UriBuilder.fromResource(ServicioClientes.class);
builder.host("{hostname}")
builder.path(ServicioClientes.class, "getCliente");
```

El código anterior define la siguiente plantilla para la URI:

```
http://{hostname}/clientes/{id}
```

A partir de esta plantilla, podremos construir la URI utilizando alguno de los métodos `buildXXX().

También podemos querer utilizar UriBuilder para crear URIS a partir de plantillas. Para ello disponemos de métodos `resolveTemplateXXX()`, que nos facilitan el trabajo:

#### Clase UriBuilder: métodos resolveTemplateXXX() para crear URIs a partir de plantillas

```
public abstract UriBuilder resolveTemplate(String name, Object value);
public abstract UriBuilder resolveTemplate(String name, Object value,
                                          boolean encodeSlashInPath);
public abstract UriBuilder resolveTemplateFromEncoded(String name,Object
 value);
public abstract UriBuilder resolveTemplates(Map<String, Object>
 templateValues);
public abstract UriBuilder resolveTemplates(
             Map<String,Object> templateValues, boolean
 encodeSlashInPath)
                                          throws IllegalArgumentException;
public abstract UriBuilder resolveTemplatesFromEncoded(
                                   Map<String, Object> templateValues);
// Devuelve la URI de la plantilla como una cadena de caracteres
public abstract String toTemplate()
```

Funcionan de forma similar a los métodos `build()` y se utilizan para resolver parcialmente las plantillas contenidas en la URI. Cada uno de los métodos devuelve una nueva instancia de UriBuilder, de forma que podemos "encadenar" varias llamadas para resolver todas las plantillas de la URI. Finalmente, usaremos el método `toTemplate()` para obtener la nueva plantilla en forma de `String`:

```
String original = "http://{host}/{id}";
String nuevaPlantilla = UriBuilder.fromUri(original)
                        .resolveTemplate("host", "localhost")
                        .toTemplate();
```

El valor de *nuevaPlantilla* para el código anterior sería: "http://localhost/{id}"

## URIs relativas mediante el uso de UriInfo

Cuando estamos escribiendo servicios que "distribuyen" enlaces, hay cierta información que probablemente no conozcamos cuando estamos escribiendo el código. Por ejemplo, podemos no conocer todavía los *hostnames* de los enlaces, o incluso los *base paths* de las URIs, en el caso de que estemos enlazando con otros servicios REST.

JAX-RS proporciona una forma sencilla de solucionar estos problemas utilizando la interfaz `javax.ws.rs.core.UriInfo`. Ya hemos introducido algunas características de esta interfaz en sesiones anteriores. Además de poder consultar información básica de la ruta, también podemos obtener instancias de UriBuilder preinicializadas con la URI base utilizada para definir los servicios JAX-RS, o la URI utilizada para invocar la petición HTTP actual:

```
public interface UriInfo {
  public URI getRequestUri();
  public UriBuilder getRequestUriBuilder();
  public URI getAbsolutePath();
  public UriBuilder getAbsolutePathBuilder();
  public URI getBaseUri();
  public UriBuilder getBaseUriBuilder();
```

Por ejemplo, supongamos que tenemos un servicio que permite acceder a Clientes desde una base de datos. En lugar de tener una URI base que devuelva todos los clientes en un único documento, podemos incluir los enlaces `previo` y `sigiente`, de forma que podamos "navegar" por los datos. Vamos a mostrar cómo crear estos enlaces utilizando la URI para invocar la petición:

```
@Path("/clientes")
public class ServicioClientes {
  @GET
  @Produces("application/xml")
  public String getCustomers(@Context UriInfo uriInfo) { ❶
    UriBuilder nextLinkBuilder = uriInfo.getAbsolutePathBuilder(); ❷
    nextLinkBuilder.queryParam("inicio", 5);
    nextLinkBuilder.queryParam("total", 10);
    URI next = nextLinkBuilder.build();
    //... rellenar el resto del documento ...
  }
```

```
  ...
}
```

❶ Para acceder a la instancia `UriInfo` que representa al petición, usamos la anotación `javax.ws.rs.core.Context`, para inyectarla como un parámetro del método del recurso REST

❷ Obtenemos un `UriBuilder` preininicializado con la URI utilizada para acceder al servicio

Para el código anterior, y dependiendo de cómo se despliegue el servicio, la URI creada podría ser:

```
http://org.expertojava/jaxrs/clientes?inicio=5&total=10
```

## Construcción de enlaces (Links) en documentos XML y en cabeceras HTTP

JAX-RS proporciona cierto soporte para construir los enlaces y devolverlos en las cabeceras de respuesta, o bien incluirlos en los documentos XML. Para ello podemos utilizar las clases `java.ws.rs.core.Link` y `java.ws.rs.core.Link.Builder`.

### Clase abstracta javax.ws.rs.core.Link

```
public abstract class Link {
   public abstract URI getUri();
   public abstract UriBuilder getUriBuilder();
   public abstract String getRel();
   public abstract List<String> getRels();
   public abstract String getTitle();
   public abstract String getType();
   public abstract Map<String, String> getParams();
   public abstract String toString();
}
```

`Link` es una clase abstracta que representa todos los metadatos contenidos en una cabecera o en un enlace Atom. El método `getUri()` representa el atributo `href` del enlace Atom. El método `getRel()` representa el atributo `rel`, y así sucesivamente. Podemos referenciar a todos los atributos a través del método `getParams()`. Finalmente, el método `toString()` convertirá la instancia `Link` en una cadena de caracteres con el formato de una cabecera `Link`.

Para crear instancias de `Link` utilizaremos un `Link.Builder`, que crearemos con alguno de estos métodos:

```
public abstract class Link {
    public static Builder fromUri(URI uri)
    public static Builder fromUri(String uri)
    public static Builder fromUriBuilder(UriBuilder uriBuilder)
    public static Builder fromLink(Link link)
    public static Builder fromPath(String path)
    public static Builder fromResource(Class<?> resource)
    public static Builder fromMethod(Class<?> resource, String method)
    ...
```

```
}
```

Los métodos `fromXXX()` funcionan de forma similar a `UriBuilder.fromXXX()`. Todos inicializan el `UriBuilder` subyacente que utilizaremos para construir el atributo `href` del enlace.

Los métodos `link()`, `uri()`, y `uriBuilder()` nos permiten sobreescribir la URI subyacente del enlace que estamos creando:

```
public abstract class Link {
  interface Builder {
      public Builder link(Link link);
      public Builder link(String link);
      public Builder uri(URI uri);
      public Builder uri(String uri);
      public Builder uriBuilder(UriBuilder uriBuilder);
      ...
```

Los siguientes métodos nos permiten asignar valores a varios atributos del enlace que estamos construyendo:

```
      ...
      public Builder rel(String rel);
      public Builder title(String title);
      public Builder type(String type);
      public Builder param(String name, String value);
      ...
```

Finalmente, él método `build()` nos permitirá construir el enlace:

```
public Link build(Object... values);
```

El objeto `Link.Builder` tiene asociado una `UriBuilder` subyacente. Los valores pasados como parámetros del método `build()` son utilizados por el `UriBuilder` para crear una URI para el enlace. Veamos un ejemplo:

```
Link link = Link.fromUri("http://{host}/raiz/clientes/{id}")
                .rel("update").type("text/plain")
                .build("localhost", "1234");
```

Si realizamos una llamada a `toString()` sobre la instancia del enlace (`link`), obtendremos lo siguiente:

```
http://localhost/raiz/clientes/1234>; rel="update"; type="text/plain"
```

A continuación mostramos dos ejemplos que muestran cómo crear instancias `Link` en las cabeceras, y en el cuerpo de la respuesta como un enlace Atom:

### Escritura de enlaces en cabeceras HTTP

```
@Path
@GET
Response get() {
    Link link = Link.fromUri("a/b/c").build();
    Response response = Response.noContent()
                                .links(link)
                                .build();
    return response; }
```

**Inclusión de un enlace Atom en el documento XMl de respuesta**

```
import javax.ws.rs.core.Link;

@XmlRootElement
public class Cliente {
  private String nombre;
  private List<Link> enlaces = new ArrayList<Link>();

  @XmlElement
  public String getNombre() {
    return nombre;
  }

  public void setNombre(String nom) {
    this.nombre = nom;
  }

  @XmlElement(name = "enlace")
  @XmlJavaTypeAdapter(Link.JaxbAdapter.class) ❶
  public List<Link> getEnlaces() {
    return enlaces; }
  }
```

❶ La clase `Link` contiene también un `JaxbAdapter`, con una implementación de la clase JAXB `XmlAdapter`, que "mapea" los objetos JAX-RS de tipo `Link` a un valor que puede ser serializado y deserializado por JAXB

El código de este ejemplo permite construir cualquier enlace y añadirlo a la clase `Cliente` de nuestro dominio. Los enlaces serán convertidos a elementos XML, que se incluirán en el documento XML de respuesta.

## 4.4. Seguridad

Es importante que los servicios rest permitan un acceso seguro a los datos y funcionalidades que proporcionan. Especialmente para servicios que permiten la realización de actualizaciones en los datos. También es interesante asegurarnos de que terceros no lean nuestros mensajes, e incluso permitir que ciertos usuarios accedan a determinadas funcionalidades pero a otras no.

Además de la especificación JAX-RS, podemos aprovechar los servicios de seguridad que nos ofrece la web y Java EE, y utilizarla en nuestros servicios REST. Estos incluyen:

**Autentificación**

Hace referencia a la validación de la **identidad** del cliente que accede a los servicios. Normalmente implica la comprobación de si el cliente ha proporcionado unos credenciales

válidos, tales como el *password*. En este sentido, podemos utilizar los mecanismos que nos proporciona la web, y las facilidades del contenedor de servlets de Java EE, para configurar los protocolos de autentificación.

**Autorización**
Una vea que el cliente se ha autenticado (ha validado su identidad), querrá interactuar con nuestro servicio REST. La autorización hace referencia a decidir si un cierto usuario puede acceder e invocar un determinado método sobre una determinada URI. Por ejemplo, podemos habilitar el acceso a operaciones PUT/POST/DELETE para ciertos usuarios, pero para otros no. En este caso, utilizaremos las facilidades que nos propociona el contenedor de servlets de Java EE, para realizar autorizaciones.

**Encriptado**
Cuando un cliente está interaccionando con un servicio REST, es posible que alguien intercepte los mensajes y los "lea", si la conexión HTTP no es segura. Los datos "sensibles" deberían protegerse con servicios criptográficos, tales como SSL.

## Autentificación en JAX-RS

Hay varios protocolos de autentificación. En este caso, vamos a ver cómo realizar una autenticación básica sobre HTTP (y que ya habéis utilizado para *servlets*). Este tipo de autentificación requiere enviar un nombre de usuario y password, codificados como Base-64, en una cabecera de la petición al servidor. El servidor comprueba si existe dicho usuario en el sistema y verifica el password enviado. Veámoslo con un ejemplo:

Supongamos que un cliente no autorizado quiere acceder a nuestros servicios REST:

```
GET /clientes/333 HTTP/1.1
```

Ya que la petición no contiene información de autentificación, el servidor debería responder la siguiente respuesta:

```
HTTP/1.1 401 Unauthorized
WWW-Autenticate: Basic realm="Cliente Realm"
```

La respuesta `401` nos indica que el cliente no está autorizado a acceder a dicha URI. La cabecera `WWW-Autenticate` especifica qué protocolo de autentificación se debería usar. En este caso, `Basic` significa que se debería utilizar una autentificación de tipo `Basic`. El atributo `realm` identifica una colección de recursos seguros en un sitio web. En este ejemplo, indica que solamente están autorizados a acceder al método GET a través de la URI anterior, todos aquellos uarios que pertenezcan al realm `Cliente Realm`, y serán autentificados por el servidor mediante una autentificación básica.

Para poder realizar la autentificación, el cliente debe enviar una petición que incluya la cabecera `Authorization`, cuyo valor sea `Basic`, seguido de la siguiente cadena de caracteres *login:password* codificada en Base64 (el valor de *login* y *password* representa el login y password del usuario). Por ejemplo, supongamos que el nombre del usuario es `felipe` y el password es `locking`, la cadena `felipe:locking` codificada como Base64 es `ZmVsaXBlOmxvY2tpbmc=`. Por lo tanto, nuestra petición debería ser la siguiente:

```
GET /clientes/333 HTTP/1.1
```

```
Authorization: Basic ZmVsaXBlOmxvY2tpbmc=
```

El cliente debería enviar esta cabecera con todas y cada una de las peticiones que haga al servidor.

El inconveniente de esta aproximación es que si la petición es interceptada por alguna entidad "hostil" en la red, el *hacker* puede obtner fácilmente el usuario y el password y utilizarlos para hacer sus propias peticiones. Utilizando una conexión HTTP encriptada (HTTPS), se soluciona este problema.

## Creación de usuarios y roles

Para poder utilizar la autentificación básica necesitamos tener creados previamente los *realms* en el servidor de aplicaciones Wildfly, y registrar los usuarios que pertenecen a dichos *realms*. La forma de hacerlo es idéntica a lo que ya habéis visto en la asignatura de Componentes Web (a través del comando `add-user.sh`).

Utilizaremos el realm por defecto "ApplicationRealm" de Wildfly, que nos permitirá además, controlar la autorización mediante la asignación de roles a usuarios.

Lo único que tendremos que hacer es añadir los usuarios a dicho realm, a través de la herramienta `$WILDFLY_HOME/bin/add-user.sh`

Al ejecutarla desde línea de comandos, deberemos elegir el ream "ApplicationRealm" e introducir los datos para cada nuevo usuario que queramos añadir, indicando su login, password, y el grupo (rol) al que queremos que pertenezca dicho usuario.

Los datos sobre los nuevos usuarios creados se almacenan en los ficheros: `application-users.properties` y `application-roles.properties`, tanto en el directorio `$WILDFLY_HOME/standalone/configuration/`, como en `$WILDFLY_HOME/domain/configuration/`

Una vez creados los usuarios, tendremos que incluir en el fichero de configuración `web.xml`, la siguiente información:

```
<web-app>
  ...
  <login-config> ❶
    <auth-method>BASIC</auth-method>
    <realm-name>ApplicationRealm</realm-name> ❷
  </login-config>

  <security-constraint>
    <web-resource-collection>
       <web-resource-name>customer creation</web-resource-name>
       <url-pattern>/rest/resources</url-pattern> ❸
       <http-method>POST</http-method> ❹
    </web-resource-collection>
    ...
  </security-constraint>
  ...
</web-app>
```

❶ El elemento <login-config> define cómo queremos autentificar nuestro despliegue. El subelemento <auth-method> puede tomar los valores `BASIC`, `DIGEST`, or `CLIENT_CERT`, correspondiéndose con la autentificación *Basic*, *Digest*, y *Client Certificate*, respectivamente.

❷ El valor de la etiqueta `<realm-name>` es el que se mostrará como valor del atributo **realm** de la cabecera `WWW-Autenticate`, si intentamos acceder al recurso sin incluir nuestras credenciales en la petición.

❸ El elemento `<login-config>` realmente NO "activa" la autentificación. Por defecto, cualquier cliente puede acceder a cualquier URL proporcionada por nuestra aplicación web sin restricciones. Para **forzar** la autentificación, debemos especificar el patrón URL que queremos asegurar (elemento `<url-pattern>`)

❹ El elemento `<http-method>` nos indica que solamente queremos asegurar las peticiones `POST` sobre esta URL. Si no incluimos el elemento `<http-method>`, todos los métodos HTTP serán seguros. En este ejemplo, solamente queremos asegurar los métodos `POST` dirigidos a la URL `/rest/resources`

## Autorización en JAX-RS

Mientras que la autentificación hacer referencia a establecer y verificar la identidad del usuario, la autorización tiene que ver con los permisos. ¿El usuario X está autorizado para acceder a un determinado recurso REST?

JAX-RS se basa en las especificaciones Java EE y de servlets para definir la forma de autorizar a los usuarios. En Java EE, la autorización se realiza asociando uno o más roles con un usuario dado y, a continuación asignando permisos basados en dicho rol. Ejemplos de roles pueden ser: `administrador`, `empleado`. Cada rol tiene asignando unos permisos de acceso a determinados recursos, por lo que asignaremos los permisos utilizando cada uno de los roles.

Para poder realizar la autorización, tendremos que incluir determinadas etiquetas en el fichero de configuración `web.xml` (tal y como ya habéis visto en la asignatura de Componentes Web). Veámoslo con un ejemplo (en el que también incluiremos autentificación):

Volvamos a nuestra aplicación de venta de productos por internet. En esta aplicación, es posible crear nuevos clientes enviando la información en formato XML a un recurso JAX-RS localizado por la anotación `@Path("/clientes")`. El servicio REST es desplegado y escaneado por la clase `Application` anotada con `@ApplicationPath("/servicios")`, de forma que la URI completa es `/servicios/clientes`. Queremos proporcionar seguridad a nuestro servicio de clientes de forma que solamente los administradores puedan crear nuevos clientes. Veamos cuál sería el contenido del fichero `web.xml`:

```
<?xml version="1.0"?>
<web-app>
  <security-constraint>
    <web-resource-collection>
       <web-resource-name>creacion de clientes</web-resource-name>
       <url-pattern>/servicios/clientes/*</url-pattern>
       <http-method>POST</http-method>
    </web-resource-collection>
    <auth-constraint> ❶
       <role-name>admin</role-name>
    </auth-constraint>
  </security-constraint>
```

```
  <login-config>
        <auth-method>BASIC</auth-method>
        <realm-name>ApplicationRealm</realm-name>
  </login-config>

  <security-role> ❷
    <role-name>admin</role-name>
  </security-role>
</web-app>
```

❶ Especificamos qué roles tienen permiso para acceder mediante POST a la URL `/services/customers`. Para ello utilizamos el elemento `<auth-constraint>` dentro de `<security-constraint>`. Este elemento tiene uno o más subelementos `<role-name>`, que definen qué roles tienen permisos de acceso definidos por `<security-constraint>`. En nuestro ejemplo, estamos dando al rol `admin` permisos para acceder a la URL `/services/customers/` con el método POST. Si en su lugar indicamos un `<role-name>` con el valor `*`, cualquier usuario podría acceder a dicha URL. En otras palabras, un `<role-name>` con el valor `*` significa que cualquier usuario que sea capaz de autentificarse, puede acceder al recurso.

❷ Para cada `<role-name>` que usemos en nuestras declaraciones `<auth-constraints>`, debemos definir el correspondiente `<security-role>` en el descriptor de despliegue.

Una limitación cuando estamos declarando las `<security-contraints>` para los recursos JAX-RS es que el elemento `<url-pattern>` solamente soporta el uso de `*` en el patrón url especificado. Por ejemplo: `/*`, `/rest/*`, `*.txt`.

## Encriptación

Por defecto, la especificación de servlets no requiere un acceso a través de HTTPS. Si queremos forzar un acceso HTTPS, podemos especificar un elemento `<user-data-constraint>` como parte de nuestra definición de restricciones de seguridad (`<security-constraint>`). Vamos a modificar nuestro ejemplo anterior para forzar un acceso a través de HTTPS:

```
<web-app>
...
  <security-constraint>

    <web-resource-collection>
       <web-resource-name>creacion de clientes</web-resource-name>
       <url-pattern>/servicios/clientes/*</url-pattern>
       <http-method>POST</http-method>
    </web-resource-collection>

    <auth-constraint>
       <role-name>admin</role-name>
    </auth-constraint>

    <user-data-constraint>
      <transport-guarantee>CONFIDENTIAL</transport-guarantee> ❶
    </user-data-constraint>
  </security-constraint>
...
```

```
</web-app>
```

❶ Todo lo que tenemos que hacer es declarar un elemento `<transport-guarantee>` dentro de `<user-data-constraint>` con el valor `CONFIDENTIAL`. Si un usuario intenta acceder a una URL con el patrón especificado a través de HTTP, será redirigido a una URL basada en HTTPS.

## Anotaciones JAX-RS para autorización

Java EE define un conjunto de anotaciones para definir metadatos de autorización. La especificación JAX-RS sugiere, aunque no es obligatorio, que las implementaciones por diferentes vendedores den soporte a dichas anotaciones. Éstas se encuentran en el paquete `javax.annotation.security` y son: @RolesAllowed, @DenyAll, @PermitAll, y @RunAs.

La anotación `@RolesAllowed` define los roles permitidos para ejecutar una determinada operación. Si anotamos una clase JAX-RS, define el acceso para todas las operaciones HTTP definidas en la clase JAX-RS. Si anotamos un método JAX-RS, la restricción se aplica solamente al método que se está anotando.

La anotación @PermitAll especifica que cualquier usuario autentificado puede invocar a nuestras operaciones. Al igual que @RolesAllowed, esta anotación puede usarse en la clase, para definir el comportamiento por defecto de toda la clase, o podemos usarla en cada uno de los métodos. Veamos un ejemplo:

```
@Path("/clientes")
@RolesAllowed({"ADMIN", "CLIENTE"}) ❶
public class ClienteResource {

  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Cliente getClienter(@PathParam("id") int id) {...}

  @RolesAllowed("ADMIN") ❷
  @POST
  @Consumes("application/xml")
  public void crearCliente(Customer cust) {...}

  @PermitAll ❸
  @GET
  @Produces("application/xml")
  public Customer[] getClientes() {}
}
```

❶ Por defecto, solamente los usuarios con rol ADMIN y CLIENTE pueden ejecutar los métodos HTTP definidos en la clase ClienteResource

❷ Sobreescribimos el comportamiento por defecto. Para el método `crearCliente()` solamente permitimos peticiones de usuarios con rol ADMIN

❸ Sobreescribimos el comportamiento por defecto. Para el método `getClientes()` de forma que cualquier usuario autentificado puede acceder a esta operación a través de la URI correspondiente, con el método GET.

La ventaja de utilizar anotaciones es que nos permite una mayor flexibilidad que el uso del fichero de configuración `web.xml`, pudiendo definir diferentes autorizaciones a nivel de método.

## Seguridad programada

Hemos visto como utilizar una seguridad declarativa, es decir, basándonos en meta-datos definidos estáticamente antes de que la aplicación se ejecute. JAX-RS proporciona una forma de obtener información de seguridad que nos permite implementar seguridad de forma programada en nuestras aplicaciones.

Podemos utilizar la interfaz `javax.ws.rs.core.SecurityContext` para determinar la identidad del usuario que realiza la invocación al método proporcionando sus credenciales. También podemos comprobar si el usuario pertenece o no a un determinado rol: Esto nos permite implementar seguridad de forma programada en nuestras aplicaciones.

```
public interface SecurityContext {
   public Principal getUserPrincipal();
   public boolean isUserInRole(String role);
   public boolean isSecure();
  public String getAuthenticationScheme();
}
```

El método `getUserPrincipal()` devuelve un objeto de tipo `javax.security.Principal`, que representa al usuario que actualmente está realizando la petición HTTP

El método `isUserInRole()` nos permite determinar si el usuario que realiza la llamada actual pertenece a un determinado rol.

El método `isSecure()` devuelve cierto si la petición actual es una conexión segura.

El método `getAuthenticationScheme()` nos indica qué mecanismo de autentificación se ha utilizado para asegurar la petición (valores típicos devueltos por el método son: `BASIC`, `DIGEST`, `CLIENT_CERT`, y `FORM`).

Podemos acceder a una instancia de SecurityContext inyectándola en un campo, método *setter*, o un parámetro de un recurso, utilizando la anotación `@Context`. Veamos un ejemplo. Supongamos que queremos obtener un fichero de log con todos los accesos a nuestra base de datos de clientes hechas por usuarios que no son administradores:

```
@Path("/clientes")
public class CustomerService {
  @GET
  @Produces("application/xml")
  public Cliente[] getClientes(@Context SecurityContext sec) {
    if (sec.isSecure() && !sec.isUserInRole("ADMIN")) {
        logger.log(sec.getUserPrincipal()
            + " ha accedido a la base de datos de clientes");
    }
    ...
  }
}
```

En este ejemplo, inyectamos una instancia de `SecurityContext` como un parámetro del método `getClientes()`. Utilizamos el método `SecurityContext.isSecure()` para determinar si se trata de una petición realizada a través de un canal seguro (como HTTPS). A continuación utilizamos el método `SecurityContext.isUserInRole()` para determinar si el usuario que realiza la llamada tiene el rol ADMIN o no. Finalmente, imprimimos el resultado en nuestro fichero de logs.

Con la introducción del API de filtros en JAX-RS 2.0, podemos implementar la interfaz `SecurityContext` y sobreescribir la petición actual sobre `SecurityContext`, utilizando el método `ContainerRequestContext.setSecurityContext()`. Lo interesante de esto es que podemos implementar nuestros propios protocolos de seguridad. Por ejemplo:

```
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.PreMatching;
import javax.ws.rs.core.SecurityContext;
import javax.ws.rs.core.HttpHeaders;

@PreMatching
public class CustomAuth implements ContainerRequestFilter {
  protected MyCustomerProtocolHandler customProtocol = ...;

  public void filter(ContainerRequestContext requestContext)
                                         throws IOException {
      String authHeader =
 request.getHeaderString(HttpHeaders.AUTHORIZATION);
      SecurityContext newSecurityContext =
 customProtocol.validate(authHeader);
      requestContext.setSecurityContext(authHeader);
  }
}
```

Este filtro no muestra todos los detalles, pero sí la idea. Extrae la cabecera `Authorization` de la petición y la pasa a nuestro propio servicio `customerProtocol`. Éste devuelve una implementación de `SecurityContext`. Finalmente sobreescribimos el SecurityContext por defecto utilizando la nueva implementación.

No vamos a explicar el API de filtros de JAS-RS 2.0. Como ya habéis visto en la asignatura de Componentes Web, los filtros son objetos que se "interponen" entre el procesamiento de las peticiones, tanto del servidor como del cliente.

El filtro mostrado en el ejemplo es un filtro **de petición** en la parte del **servidor**. Este tipo de filtros se ejecuta antes de que se invoque a un método JAX-RS.

## 4.5. Ejercicios

Para los ejercicios de esta sesión proporcionamos el **MÓDULO s4-foroAvanzado**, que tendréis que usar como plantilla para realizar las tareas planteadas.

El proyecto está estructurado lógicamente en los siguientes paquetes:

- org.expertojava.negocio
- org.expertojava.rest

A su vez, cada uno de ellos contiene los subpaquetes `api` y `modelo`, con las clases relacionadas con los servicios proporcionados, y los datos utilizados por los servicios, respectivamente.

El API rest implementado es el siguiente:

- Recurso *UsuariosResource.java*
    - GET /usuarios, proporciona un listado con los usuarios del foro
- Subrecurso *UsuarioResource.java*
    - GET /usuarios/login, proporciona información sobre el usuario cuyo login es "login"
    - PUT /usuarios/login, actualiza los datos de un usuario
    - DELETE /usuarios/login, borra los datos de un usuario
    - GET /usuarios/login/mensajes, obtiene un listado de los mensajes de un usuario
- Recurso *MensajesResource.java*
    - GET /mensajes, proporciona un listado con los mensajes del foro
    - POST /mensajes, añade un mensaje nuevo en el foro
    - GET /mensajes/id, proporciona información sobre el mensaje cuyo id es "id"
    - PUT /mensajes/id, modifica un mensaje
    - DELETE /mensajes/id, borra un mensaje

Una vez desplegada la aplicación, podéis añadir datos a la base de datos del foro, utilizando los datos del fichero /src/main/resources/foro.sql. Para ello simplemente tendréis que invocar la goal Maven correspondiente desde la ventana *Maven Projects > s4-foroAvanzado > Plugins > sql > sql:execute*

En el directorio `src/main/resources` tenéis un fichero de texto (`instrucciones.txt`) con información adicional sobre la implementación proporcionada.

A partir de las plantillas, se pide:

### Uso de Hateoas (1 puntos)

Vamos a añadir a los servicios enlaces a las operaciones que podemos realizar con cada recurso, siguiendo el estilo Hateoas.

a. Para los usuarios:

- En el listado de usuarios añadir a cada usuario un enlace con relación `self` que apunte a la dirección a la que está mapeado el usuario individual.
- En la operación de obtención de un usuario individual, incluir los enlaces para ver el propio usuario (self), modificarlo (usuario/modificar), borrarlo (usuario/borrar), o ver los mensajes que envió el usuario (usuario/mensajes).

b. Para los mensajes:

- En el listado de mensajes añadir a cada mensaje un enlace con relación `self` que apunte a la dirección a la que está mapeado el mensaje individual.
- En la operación de obtención de un mensaje individual, incluir los enlaces para ver el propio mensaje (self), modificarlo (mensaje/modificar), borrarlo (mensaje/borrar), o ver los datos del usuario que envió el mensaje (mensaje/usuario).

Utiliza postman para comprobar las modificaciones realizadas.

## Ejercicio seguridad (1 punto)

Vamos ahora a restringir el acceso al servicio para que sólo usuarios registrados puedan realizar modificaciones. Se pide:

a. Añadir al usuario "pepe" en el "ApplicationRealm" de wildfly, con la contraseña "pepe", y perteneciente al grupo (rol) "registrado"
b. Configurar, mediante seguridad declarativa, para que las operaciones de modificación (POST, PUT y DELETE) sólo la puedan realizar los usuarios con rol **registrado**. Utilizar autentificación de tipo BASIC.
c. Ahora vamos a hacer que la modificación o borrado de usuarios sólo pueda realizarlas el mismo usuario que va a modificarse o borrarse. Para ello utilizaremos seguridad programada. En el caso de que el usuario que va a realizar la modificación o borrado quiera borrar/modificar otros usuarios lanzaremos la excepción `WebApplicationException(Status.FORBIDDEN)`
d. Vamos a hacer lo mismo con los mensajes. Sólo podrá modificar y borrar mensajes el mismo usuario que los creó, y al publicar un nuevo mensaje, forzaremos que el login del mensaje sea el del usuario que hay autentificado en el sistema.

Utiliza postman para comprobar las modificaciones realizadas.

# 5. Api cliente. Procesamiento JSON y Pruebas

Hasta ahora hemos hablado sobre la creación de servicios web RESTful y hemos "probado" nuestros servicios utilizando el cliente que nos proporciona IntelliJ, curl, o Postman, para realizar peticiones y observar las respuestas. JAX-RS 2.0 proporciona un API cliente para facilitar la programación de clientes REST, que presentaremos en esta sesión.

En sesiones anteriores, hemos trabajado con representaciones de texto y xml, fundamentalmente. Aquí hablaremos con más detalle de JSON, que constituye otra forma de representar los datos de las peticiones y respuestas de servicios REST muy extendida.

Finalmente, veremos cómo implementar pruebas sobre nuestro servicio utilizando el API cliente y el *framework junit*.

## 5.1. API cliente. Visión general

La especificación JAX-RS 2.0 incorpora un API cliente HTTP, que facilita enormemente la implementación de nuestros clientes RESTful, y constituye una clara alternativa al uso de clases Java como `java.net.URL`, librerías externas (como la de Apache) u otras soluciones propietarias.

El API cliente está contenido en el paquete `javax.ws.rs.client`, y está diseñado para que se pueda utilizar de forma "fluida" (*fluent*). Esto significa, como ya hemos visto, que lo utilizaremos "encadenando" una sucesión de llamadas a métodos del API, permitiéndonos así escribir menos líneas de código. Básicamente está formado por tres clases principales: **Client**, **WebTarget** y **Response** (ya hemos hablado de esta última en sesiones anteriores).

Para acceder a un recurso REST mediante el API cliente es necesario seguir los siguientes pasos:

1. Obtener una instancia de la interfaz `Client`
2. Configurar la instancia `Client` a través de un *target* (instancia de `WebTarget`)
3. Crear una petición basada en el *target* anterior
4. Invocar la petición

Vamos a mostrar un código ejemplo para ilustrar los pasos anteriores. En este caso vamos a invocar peticiones POST y GET sobre una URL (*target*) para crear y posteriormente consultar un objeto `Cliente` que representaremos en formato XML:

```
...
Client client = ClientBuilder.newClient(); ❶

WebTarget target =
      client.target("http://expertojava.org/clientes"); ❷

Response response =
      target
        .request()  ❸
        .post(Entity.xml(new Cliente("Alvaro", "Gomez"))); ❹

response.close(); ❺
```

```
Cliente cliente = target.queryParam("nombre", "Alvaro Gomez")
                        .request()
                        .get(Cliente.class);  ❻
client.close();
...
```

❶ Obtenemos una instancia `javax.ws.rs.client.Client`
❷ Creamos un `WebTarget`
❸ Creamos la petición
❹ Realizamos una invocación POST
❺ Cerramos (liberamos) el *input stream* para esta respuesta (en el caso de que esté disponible y abierto). Es una operación *idempotente*, es decir, podemos invocarla múltiples veces con el mismo efecto
❻ A partir de un `WebTarget`, establecemos los valores de los `queryParams` de la URI de la petición, creamos la petición, y realizamos una invocación GET

A continuación explicaremos con detalle los pasos a seguir para implementar un cliente utilizando el API de JAX-RS.

## Obtenemos una instancia Client

La interfaz `javax.ws.rs.client.Client` es el principal punto de entrada del API Cliente. Dicha interfaz define las acciones e infraestructura necesarias requeridas por un cliente REST para "consumir" un servicio web RESTful. Los objetos `Client` se crean a partir de la clase `ClientBuilder`:

### Clase *ClientBuilder*: utilizada para crear objetos *Client*

```
public abstract class ClientBuilder implements Configurable<ClientBuilder>
 {
  public static Client newClient() {...}
  public static Client newClient(final Configuration configuration) {...}

  public static ClientBuilder newBuilder() {...}

  public abstract ClientBuilder sslContext(final SSLContext sslContext);
  public abstract ClientBuilder keyStore(final KeyStore keyStore,
                                        final char[] password);
  public ClientBuilder keyStore(final KeyStore keyStore,
                                final String password) {}
  public abstract ClientBuilder trustStore(final KeyStore trustStore);
  public abstract ClientBuilder hostnameVerifier(final HostnameVerifier
 verifier);

  public abstract Client build();
}
```

La forma más sencilla de crear un objeto `Client` es mediante `ClientBuilder.newClient()`. Este método proporciona una instancia pre-inicializada de tipo `Client` lista para ser usada. La clase *ClientBuilder* nos proporciona métodos adicionales, con los que podremos configurar diferentes propiedades del objeto.

Veamos un ejemplo de uso de `ClientBuilder.newBuilder()` utilizando además alguno de los métodos proporcionados para configurar nuestra instancia de tipo `Client` que vamos

a crear. Los métodos `register()` y `property()` son métodos de la interfaz *Configurable* (y que son implementados por *ClientBuilder*).

### Ejemplo de uso de *ClientBuilder*

```
Client cliente = ClientBuilder.newBuilder() ❶
                    .property("connection.timeout", 100) ❷
                    .sslContext(sslContext) ❸
                    .register(JacksonJsonProvider.class) ❹
                    .build(); ❺
```

❶ Creamos un `ClientBuilder` invocando al método estático `ClientBuilder.newBuilder()`
❷ Asignamos una propiedad específica de la implementación concreta de JAX-RS que estemos utilizando que controla el *timeout* de las conexiones de los *sockets*
❸ Especificamos el *sslContext* que queremos utilizar para gestionar las conexiones HTTP
❹ Registramos a través del método `register()` (de la interfaz *Configurable*) una clase anotada con `@Provider`. Dicha clase "conoce" cómo serializar objetos Java a JSON y viceversa
❺ Finalmente, realizamos una llamada a `build()` para crear la instancia `Client`

Las instancias de `Client` gestionan conexiones con el cliente utilizando *sockets* y son objetos bastante *pesados*. Se deberían reutilizar las instancias de esta interfaz en la medida de lo posible, ya que la inicialización y destrucción de dichas instancias consume mucho tiempo. Por lo tanto, por razones de rendimiento, debemos limitar el número de instancias `Client` en nuestra aplicación.

```
Client client = ClientBuilder.newClient(); ❶
...
client.close(); ❷
```

❶ Obtenemos una instancia de tipo `Client` invocando al método `ClientBuilder.newClient()`
❷ Utilizamos el método `close()` para "cerrar" la instancia `Client` después de realizar todas las invocaciones sobre el *target* del recurso. De esta forma, "cerramos" la conexión de forma que se liberan sus recursos, y ya no podremos seguir usándola.

Recuerda siempre invocar el método `close()` sobre nuestros objetos `Client` después de que hayamos realizado todas las invocaciones sobre el *target* del/los recurso/s REST. A menudo, los objetos `Client` reutilizan conexiones por razones de rendimiento. Si no los cerramos después de utilizarlos, estaremos desaprovechando recursos del sistema muy "valiosos". **Cerrar la conexión implica cerrar el socket**

Igualmente, si el resultado de una invocación a un servicio rest es una instancia de `Response` debemos invocar el método `close()` sobre dichos objetos `Response` para liberar la conexión. Liberar una conexion significa permitir que ésta esté disponible para otro uso por una instancia *Client*. **Liberar la conexión no implica cerrar el socket**.

La interfaz `Client` es una sub-interfaz de `Configurable`. Esto nos permitirá utililizar los métodos `property()` y `register()` para cambiar la configuración y registrar componentes en la parte del cliente en tiempo de ejecución.

**Interfaz *Client* (es una subinterfaz de *Configurable*)**

```
public interface Client extends Configurable<Client> {

    public void close();

    public WebTarget target(String uri);
    public WebTarget target(URI uri);
    public WebTarget target(UriBuilder uriBuilder);
    public WebTarget target(Link link);
    ...
}
```

Sin embargo, el principal propósito de `Client` es crear instancias de `WebTarget`, como veremos a continuación.

## Configuramos el *target* del cliente (URI)

La interfaz `javax.ws.rs.client.WebTarget` **representa la URI** específica que queremos invocar para acceder a un recurso REST particular.

**Interfaz WebTarget (es una subinterfaz de *Configurable*)**

```
public interface WebTarget extends Configurable<WebTarget> {

   public URI getUri();
   public UriBuilder getUriBuilder();

   public WebTarget path(String path);
   public WebTarget resolveTemplate(String name, Object value);
   ...
   public WebTarget resolveTemplates(Map<String, Object> templateValues);
   ...
   public WebTarget matrixParam(String name, Object... values);
   public WebTarget queryParam(String name, Object... values);
   ...
}
```

La interfaz `WebTarget` tiene métodos para extender la URI inicial que hayamos construido. Podemos añadir, por ejemplo, segmentos de *path* o parámetros de consulta invocando a los métodos `WebTarget.path()`, o `WebTarget.queryParam()`, respectivamente. Si la instancia de `WebTarget` contiene plantillas de parámetros, los métodos `WebTarget.resolveTemplate()` pueden asignar valores a las variables correspondientes. Por ejemplo:

**Ejemplo para crear la URI http://ejemplo.com/clientes/123?verboso=true**

```
WebTarget target =
        client
           .target("http://ejemplo.com/clientes/{id}") ❶
           .resolveTemplate("id", "123") ❷
           .queryParam("verboso", true); ❸
```

❶ Inicializamos un `WebTarget` con una URI que contiene una plantilla con un parámetro: `{id}`. El objeto `client` es una instancia de la clase `Client
❷ El método `resolveTemplate()` "rellena" la expresión `id` con el valor "123"
❸ Finalmente añadimos a la URI un parámetro de consulta: `?verboso=true`

Las instancias de `WebTarget` son **inmutables** con respecto a la URI que contienen. Esto significa que los métodos para especificar segmentos de `path` adicionales y parámetros devuelven **una nueva instancia** de `WebTarget`. Sin embargo, las instancias de WebTarget son **mutables** respecto a su configuración. Por lo tanto, la configuración de objetos `WebTarget` **no crea nuevas instancias**.

Veamos otro ejemplo:

```
WebTarget base = cliente.target("http://expertojava.org/"); ❶
WebTarget clienteURI = base.path("cliente"); ❷
clienteURI.register(MyProvider.class); ❸
```

❶ `base` es una instancia de `WebTarget` con el valor de URI `http://exertojava.org/`
❷ `clienteURI` es una instancia de `WebTarget` con el valor de URI `http://exertojava.org/cliente`
❸ Configuramos `clienteURI` registrando la clase `MyProvider`

En este ejemplo creamos dos instancias de `WebTarget`. La instancia `clienteURI` hereda la configuración de `base` y posteriormente modificamos la configuramos registrando una clase `Provider`. Los cambios sobre la configuración de `clienteURI` no afectan a la configuración de `base`, ni tampoco se crea una nueva instancia de `WebTarget`.

Los beneficios del uso de `WebTarget` se hacen evidentes cuando construimos URIs complejas, por ejemplo cuando extendemos nuestra URI base con segmentos de `path` adicionales o plantillas. El siguiente ejemplo ilustra estas situaciones:

```
WebTarget base = cliente.target("http://expertojava.org/"); ❶
WebTarget saludo = base.path("hola").path("{quien}"); ❷
Response res = saludo.resolveTemplate("quien", "mundo").request().get();
```

❶ `base` representa la URI: `http://expertojava.org`
❷ `saludo` representa la URI: `http://expertojava/hola/{quien}`

En el siguiente ejemplo, utilizamos una URI base, y a partir de ella construimos otras URIs que representan servicios diferentes proporcionados por nuestro recurso REST.

```
Client cli = ClientBuilder.newClient();
WebTarget base = client.target("http://ejemplo/webapi");
WebTarget lectura = base.path("leer"); ❶
WebTarget escritura = base.path("escribir"); ❷
```

❶ `lectura` representa la uri: `http://ejemplo/webapi/leer`
❷ `escritura` representa la uri: `http://ejemplo/webapi/escribir`

El método `WebTarget.path()` crea una nueva instancia de `WebTarget` añadiendo a la URI actual el segmento de ruta que se pasa como parámetro.

## Construimos y Realizamos la petición

Una vez que hemos creado y configurado convenientemente el `WebTarget`, que representa la URI que queremos invocar, tenemos que construir la petición y finalmente realizarla.

Para **construir la petición** podemos Utilizar uno de los métodos `WebTarget.request()`, que mostramos a continuación:

### Interfaz WebTarget: métodos para comenzar a construir la petición

```
public interface WebTarget extends Configurable<WebTarget> {
  ...
  public Invocation.Builder request();
  public Invocation.Builder request(String... acceptedResponseTypes);
  public Invocation.Builder request(MediaType... acceptedResponseTypes);
}
```

Normalmente invocaremos `WebTarget.request()` pasando como parámetro el `media type` aceptado como respuesta, en forma de `String` o utilizando una de las constantes de `javax.ws.rs.core.MediaType`. Los métodos `WebTarget.request()` devuelven una instancia de `Invocation.Builder`, una interfaz que proporciona métodos para preparar la petición del cliente y también para invocarla.

La interface *Invocation.Builder* Contiene un conjunto de métodos que nos permiten construir diferentes tipos de cabeceras de peticiones. Así, por ejemplo, proporciona varios métodos `acceptXXX()` para indicar diferentes tipos MIME, lenguajes, o "encoding" aceptados. También proporciona métodos `cookie()` para especificar *cookies* para enviar al servidor. Finalmente proporciona métodos `header()` para especificar diferentes valores de cabeceras.

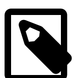
Ejemplos de uso de esta interfaz para construir la petición:

```
Client cli = ClientBuilder.newClient();
cli.invocation(Link.valueOf("http://ejemplo/rest")).accept("application/
json").get();
//si no utilizamos el método invocation, podemos hacerlo así:
cli.target("http://ejemplo/rest").request("application/json").get();
```

```
Client cliente = ClientBuilder.newClient();
WebTarget miRecurso = client.target("http://ejemplo/webapi/mensaje")
  .request(MediaType.TEXT_PLAIN);
```

El uso de una constante `MediaType` es equivalente a utilizar el `String` que define el tipo MIME:

```
Invocation.Builder builder = miRecurso.request("text/plain");
```



Hemos visto que *WebTarget* implementa métodos *request()* cuyos parámetros especifican el tipo MIME de la cabecera *Accept* de la petición. El código puede resultar más "legible" si usamos en su lugar el método

> *Invocation.Builder.accept()*. En cualquier caso es una cuestión de gustos personales.

Después de determinar el *media type* de la respuesta, **invocamos la petición** realizando una llamada a uno de los métodos de la instancia de `Invocation.Builder` que se corresponde con el tipo de petición HTTP esperado por el recurso REST, al que va dirigido dicha petición. Estos métodos son:

- `get()`
- `post()`
- `delete()`
- `put()`
- `head()`
- `options()`

La interfaz *Invocation.Builder* es una subinterfaz de la interfaz *SyncInvoker*, y es la que especifica los métodos anteriores (get, post, …) para realizar peticiones **síncronas**, es decir, que hasta que no nos conteste el servidor, no podremos continuar procesando el código en la parte del cliente.

Las peticiones **GET** tienen los siguientes prototipos:

#### Interface SyncInvoker: peticiones GET síncronas

```
public interface SyncInvoker {
  ...
  <T> T get(Class<T> responseType);
  <T> T get(GenericType<T> responseType);
  Response get();
  ...
}
```

Los primeros dos métodos genéricos convertirán una respuesta HTTP con éxito a tipos Java específicos indicados como parámetros del método. El tercero devuelve una instancia de tipo *Response*. Por ejemplo:

#### Ejemplos de peticiones GET utilizando el API cliente jasx-rx 2.0

```
Client cli = ClientBuilder.newClient();

//petición get que devuelve una instancia de Cliente
Cliente cliRespuesta = cli.target("http://ejemplo/clientes/123")
                          .request("application/json")
                          .get(Cliente.class); ❶

//petición get que devuelve una lista de objetos Cliente
List<Cliente> cliRespuesta2 =
                 cli.target("http://ejemplo/clientes")
                    .request("application/xml")
                    .get(new GenericType<List<Cliente>>() {}); ❷

//petición get que devuelve un objeto de tipo Response
Response respuesta =
              cli.target("http://ejemplo/clientes/245")
```

```
                    .request("application/json")
                    .get();   ❸
try {
   if (respuesta.getStatus() == 200) {
      Cliente cliRespuesta =
                 respuesta.readEntity(Cliente.class); ❹
   }
} finally {
     respuesta.close();
}
```

❶ En la primera petición queremos que el servidor nos devuelva la respuesta en formato JSON, y posteriormente la convertiremos en el tipo `Cliente` utilizando un de los componentes `MessageBodyReader` registrados.

❷ En la segunda petición utilizamos la clase `javax.ws.rs.core.GenericType` para informar al correspondiente `MessageBodyReader` del tipo de objetos de nuestra Lista. Para ello creamos una clase anónima a la que le pasamos como parámetro el tipo genérico que queremos obtener.

❸ En la tercera petición obtenemos una instancia de *Response*, a partir de la cual podemos obtener el cuerpo del mensaje de respuesta del servidor

❹ El método *readEntity()* asocia el tipo Java solicitado (en este caso el tipo java *Cliente*) y el contenido de la respuesta recibida con el correspondiente proveedor de entidades (de tipo *MessageBodyReader*) para obtener dicho tipo Java a partir de la respuesta HTTP recibida. # En sesiones anteriores hemos utilizado la clase Response desde el servicio REST, para construir la respuesta que se envía al cliente.

Recordemos algunos de los métodos que podemos utilizar desde el cliente para analizar la respuesta que hemos obtenido:

### Métodos de la clase Response que utilizaremos desde el cliente

```
public abstract class Response {
   public abstract Object getEntity();  ❶
   public abstract int getStatus();  ❷
   public abstract Response.StatusType getStatusInfo() ❸
   public abstract MultivaluedMap<String, Object> getMetadata();  ❹
   public abstract URI getLocation();  ❺
   public abstract MediaType getMediaType();  ❻
   public MultivaluedMap<String,Object> getHeaders(); ❼
   public abstract <T> T readEntity(Class<T> entityType); ❽
   public abstract <T> T readEntity(GenericType<T> entityType); ❾
   public abstract void close(); ❿
    ...
}
```

❶ El método `getEntity()` devuelve el objeto Java correspondiente al cuerpo del mensaje HTTP.

❷ El método `getStatus()` devuelve el código de respuesta HTTP.

❸ El método `getStatusInfo()` devuelve la información de estado asociada con la respuesta.

❹ El método `getMetadata()` devuelve una instancia de tipo `MultivaluedMap` con las cabeceras de la respuesta.

❺ El método `getLocation()` devuelve la URI de la cabecera Location de la respuesta.
❻ El método `getMediaType()` devuelve el *mediaType* del cuerpo de la respuesta
❼ El método `getHeaders()` devuelve las cabeceras de respuesta con sus valores correspondientes.
❽ El método `readEntity()` devuelve la entidad del cuerpo del mensaje utilizando un *MessageBodyReader* que soporte el mapeado del *inputStream* de la entidad a la clase Java especificada como parámetro.
❾ El método `readEntity()` también puede devolver una clase genérica si se dispone del *MessageBodyReader* correspondiente.
❿ El método `close()` cierra el *input stream* correspondiente a la entidad asociada del cuerpo del mensaje (en el caso de que esté disponible y "abierto"). También libera cualquier otro recurso asociado con la respuesta (como por ejemplo datos posiblemente almacenados en un *buffer*).

Veamos otro ejemmplo. Si el recurso REST espera una petición HTTP GET, invocaremos el método `Invocation.Builder.get()`. El tipo de retorno del método debería corresponderse con la entidad devuelta por el recurso REST que atenderá la petición.

```
Client cliente = ClientBuilder.newClient();
WebTarget miRecurso = cliente.target("http://ejemplo/webapi/lectura");
String respuesta = miRecurso.request(MediaType.TEXT_PLAIN)
                            .get(String.class);
```

O también podríamos codificarlo como:

```
Client cliente = ClientBuilder.newClient();
String respuesta = cliente
                     .target("http://ejemplo/webapi/lectura")
                     .request(MediaType.TEXT_PLAIN)
                     .get(String.class);
```

Si el tipo de retorno de la petición GET es una colección, usaremos `javax.ws.rs.core.GenericType<T>` como parámetro del método, en donde `T` es el tipo de la colección:

```
List<PedidoAlmacen> pedidos = client
                              .target("http://ejemplo/webapi/lectura")
                              .path("pedidos")
                              .request(MediaType.APPLICATION_XML)
                              .get(new GenericType<List<PedidoAlmacen>>()
 {});
```

Si el recurso REST destinatario de la petición espera una petición de tipo HTTP POST, invocaremos el método `Invocation.Builder.post()`.

Las peticiones **POST** tienen los siguientes prototipos:

### Interface SyncInvoker: peticiones POST síncronas

```
public interface SyncInvoker {
  ...
  <T> T post(Entity<?> entity, Class<T> responseType);
```

```
    <T> T post(Entity<?> entity, GenericType<T> responseType)
    Response post(Entity<?> entity);
    ...
}
```

Los primeros dos métodos genéricos envían una entidad (clase java + tipo MIME asociado), indicada como primer parámetro del método, y como segundo parámetro se indica el tipo java al que se convertirá la respuesta recibida. El tercero envía una entidad y devuelve una instancia de tipo *Response*. Por ejemplo:

Veamos un ejemplo de invocación de peticiones POST.

### Ejemplo de petición POST utilizando el API cliente jasx-rx 2.0

```
Client cli = ClientBuilder.newClient();
Pedido pe = new PedidoAlmacen(...);
Pedido peRespuesta = cli
            .target(...)
            .request()
            .post(Entity.entity(new Pedido(), "application/json"),
                  Pedido.class);
```

En este caso estamos realizando una petición POST. Como *payload* del mensaje enviamos un objeto *Pedido* representado en formato json. La entidad esperada como respuesta debe ser de tipo Pedido.

Esto implica que en el lado del servidor, el método que atiende la petición @Post tendrá un parámetro de tipo Pedido y se deberán serializar los objetos de tipo Pedido a json, ya que es el tipo MIME asociado a esta entidad ( especificado en la cabera *Content-Type* de la petición HTTP).

La clase *Entity* encapsula los objetos Java que queremos enviar con las peticiones GET o POST. No tiene un constructor público. En su lugar tenemos que invocar uno de sus métodos estáticos:

### Clase javax.ws.rs.client.Entity

```
public final class Entity<T> {
   ...
   public static <T> Entity<T> entity(T entity, String mediaType) ❶
   public static <T> Entity<T> entity(T entity, MediaType mediaType) ❷
   public static <T> Entity<T> xml(final T entity) { } ❸
   public static <T> Entity<T> json(final T entity) { } ❹
   public static <T> Entity<T> text(T entity) { } ❺
   public static Entity<Form> form(final Form form) { } ❻
   ...
}
```

❶ El método estático *entity()* crea una entidad (clase Java) con un tipo MIME asociado dado por la cadena de caracteres *mediaType*

❷ El método estático *entity()* crea una entidad (clase Java) con un tipo MIME indicado en *mediaType*

❸ El método *xml* crea una entidad (clase Java) con el tipo MIME "application/xml"

❹ El método *json* crea una entidad (clase Java) con el tipo MIME "application/jsom"
❺ El método *text* crea una entidad (clase Java) con el tipo MIME "text/plain"
❻ El método *form* crea una entidad (clase Java) con el tipo MIME "application/x-www-form-urlencoded"

Veamos otro ejemplo de invocación POST que utiliza la clase *Entity*:

### Ejemplo de petición POST y uso de clase Entity

```
NumSeguimiento numSeg = client
          .target("http://ejemplo/webapi/escritura")
          .request(MediaType.APPLICATION_XML)  ❶
          .post(Entity.xml(pedido), NumeroSeguimiento.class); ❷
```

❶ Especificamos como parámetro de la petición *request()* el tipo MIME que aceptamos en la respuesta (cabecera HTTP *Accept*).
❷ Realizamos una petición POST. El cuerpo del mensaje se crea con la llamada `Entity.xml(pedido)`. El tipo Entity encapsula la entidad del mensaje (tipo Java *Pedido*) y el tipo MIME asociado (tipo MIME *application/xml*) .

Veamos un ejemplo en el que enviamos parámetros de un formulario en una petición POST:

### Ejemplo de envío de datos de un formulario en una petición POST

```
Form form = new Form().param("nombre", "Pedro")
                      .param("apellido", "Garcia");
...
Response response = client.target("http://ejemplo/clientes")
                    .request().
                    .post(Entity.form(form));
response.close();
```

La petición POST del código anterior envía los datos del formulario, y espera recibir como respuesta una entidad de tipo *Response*.

El código en el lado del servidor será similar a éste:

### Servicio rest que sirve una petición POST a partir de datos de un formulario

```
...
@POST
@Path("/clientes")
@Produces("text/html")
public Response crearCliente(@FormParam("nombre")String nom,
            @FormParam("apellido")String ape)
{
    ... //creamos el nuevo cliente
    return Response.ok(RESPONSE_OK).build();
}
```

## Manejo de excepciones

Veamos qué ocurre si se produce una excepción cuando utilizamos una forma de invocación que automáticamente convierte la respuesta en el tipo especificado. Supongamos el siguiente ejemplo:

```
Cliente cli = client.target("http://tienda.com/clientes/123")
                    .request("application/json")
                    .get(Cliente.class);
```

En este escenario, el *framework* del cliente convierte cualquier código de error HTTP en una de las excepciones que añade JAX-RS 2.0 (BadRequesException, ForbiddenException…) y que ya hemos visto. Podemos capturar dichas excepciones en nuestro código para tratarlas adecuadamente:

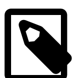
```
try {
  Cliente cli = client.target("http://tienda.com/clientes/123")
                      .request("application/json")
                      .get(Cliente.class);
} catch (NotAcceptableException notAcceptable) {
      ...
} catch (NotFoundException notFound) {
      ...
}
```

Si el servidor responde con un error HTTP no cubierto por alguna excepción específica JAX-RS, entonces se lanza una excepción de propósito general. La clase `ClientErrorException` cubre cualquier código de error en la franja del 400. La clase `ServerErrorException` cubre cualquier código de error en la franja del 500.

Si el servidor envía alguna de los códigos de respuesta HTTP 3xx (clasificados como códigos de la categoría *redirección*), el API cliente lanza una `RedirectionException`, a partir de la cual podemos obtener la URL para poder tratar la redirección nosotros mismos. Por ejemplo:

```
WebTarget target = client.target("http://tienda.com/clientes/123");
boolean redirected = false;

Cliente cli = null;
do {
     try {
          cli = target.request("application/json")
                      .get(Cliente.class);
     } catch (RedirectionException redirect) {
          if (redirected) throw redirect;
          redirected = true;
          target = client.target(redirect.getLocation());
   }
} while (cli == null);
```

En este ejemplo, volvemos a iterar si recibimos un código de respuesta 3xx. El código se asegura de que sólo permitimos un código de este tipo, cambiando el valor de la variable `redirect` en el bloque en el que capturamos la exceptión. A continuación cambiamos el `WebTarget` (en el bloque `catch`) al valor de la cabecera `Location` de la respuesta del servidor.



Los códigos de estado HTTP 3xx indican que es neceario realizar alguna acción adicional para que el servidor pueda completar la petición. La acción requerida puede llevarse a cabo sin necesidad de interactuar con el

> cliente sólo si el método utilizado en la segunda petición es GET o HEAD (ver http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html)

## 5.2. Procesamiento JSON

JSON (**J**ava**S**cript **O**bject **N**otation) es un formato para el intercambio de datos basado en texto, derivado de Javascript (Javascript disponde de una función nativa: `eval()` para convertir *streams* JSON en objetos con propiedades que son accesibles sin necesidad de manipular ninguna cadena de caracteres).

La especificación **JSR 353**[9] proporciona un API para el procesamiento de datos JSON (*parsing*, transformación y consultas).

La gramática de los objetos JSON es bastante simple. Sólo se requieren dos estructuras: objetos y *arrays*. Un **objeto** es un conjunto de pares *nombre-valor*, y un **array** es una lista de valores. JSON define siete tipos de valores: `string`, `number`, `object`, `array`, `true`, `false`, y `null`.

El siguiente ejemplo muestra datos JSON para un objeto que contiene pares *nombre-valor*.

**Ejemplo formato JSON**

```
{ "nombre": "John",
  "apellidos": "Smith",
  "edad": 25,
  "direccion": { "calle": "21 2nd Street",
                 "ciudad": "New York",
                 "codPostal": "10021"
               },
  "telefonos": [
         { "tipo": "fijo",
           "numero": "212 555-1234"
         },
         {
           "tipo": "movil",
           "numero": "646 555-4567"
         }
  ]
 }
```

El objeto anterior tiene cinco pares *nombre-valor*:

- Los dos primeros son `nombre` y `apellidos`, con el valor de tipo `String`
- El tercero es `edad`, con el valor de tipo `number`
- El cuarto es `direccion`, con el valor de tipo `object`
- El quinto es `telefonos`, cuyo valor es de tipo `array`, con dos objetos

JSON tiene la siguiente **sintaxis**:

- Los **objetos** están rodeados por llaves `{}`, sus **pares de elementos** *nombre-valor* están separados por una coma `,`, y el *nombre* y el *valor* de cada par están separados por dos puntos `:`. Los **nombres** en un objeto son de tipo `String`, mientras que sus **valores**

[9] https://jcp.org/aboutJava/communityprocess/final/jsr353/index.html

pueden ser cualquiera de los siete tipos que ya hemos indicado, incluyendo a otro objeto, u otro array.

- Los **arrays** están rodeados por corchetes `[]`, y sus valores están separados por una coma `,`. Cada valor en un *array* puede ser de un tipo diferente, incluyendo a otro objeto o array.
- Cuando los objetos y arrays contienen otros objetos y/o arrays, los datos adquieren una **estructura de árbol**

Los servicios web RESTful utilizan JSON habitualmente tanto en las peticiones, como en las respuestas. La cabecera HTTP utilizada para indicar que el contenido de una petición o una respuesta es JSON es la siguiente:

```
Content-Type: application/json
```

La representación JSON es normalmente más compacta que las representaciones XML debido a que JSON no tiene *etiquetas de cierre*. A diferencia de XML, JSON no tiene un "esquema" de definición y validación de datos ampliamente aceptado.

Actualmente, las aplicaciones Java utilizan diferentes librerías para producir/consumir JSON, que tienen que incluirse junto con el código de la aplicación, incrementando así el tamaño del archivo desplegado. El API de Java para procesamiento JSON proporciona un API estándar para analizar y generar JSON, de forma que las aplicaciones que utilicen dicho API sean más "ligeras" y portables.

Para generar y parsear datos JSON, hay dos modelos de programación, que son similares a los usados para documentos XML:

- El modelo de objetos: crea un árbol en memoria que representa los datos JSON
- El modelo basado en *streaming*: utiliza un *parser* que lee los datos JSON elemento a elemento (uno cada vez).

Java EE incluye soporte para JSR 353, de forma que el API de java para procesamiento JSON se encuentra en los siguientes paquetes:

- El paquete `javax.json` contiene interfaces para leer, escribir y construir datos JSON, según el modelo de objetos, así como otras utilidades.
- El paquete `javax.json.stream` contiene una interfaz para parsear y generar datos JSON para el modelo *streaming*

Vamos a ver cómo producir y consumir datos JSON utilizando cada uno de los modelos.

## 5.3. Modelo de procesamiento basado en el modelo de objetos

En este caso se crea un árbol en memoria que representa los datos JSON (todos los datos). Una vez construido el árbol, se puede navegar por él, analizarlo, o modificarlo. Esta aproximación es muy flexible y permite un procesamiento que requiera acceder al contenido completo del árbol. En contrapartida, normalmente es más lento que el modelo de *streaming* y requiere utilizar más memoria. El modelo de objetos genera una salida JSON navegando por el árbol entero de una vez.

El siguiente código muestra cómo crear un modelo de objetos a partir de datos JSON desde un fichero de texto:

**Creación de un modelos de objetos a partir de datos JSON**

```
import java.io.FileReader;
import javax.json.Json;
import javax.json.JsonReader;
import javax.json.JsonStructure;
...
JsonReader reader = Json.createReader(new FileReader("datosjson.txt"));
JsonStructure jsonst = reader.read();
```

El objeto `jsonst` puede ser de tipo `JsonObject` o de tipo `JsonArray`, dependiendo de los contenidos del fichero. `JsonObject` y `JsonArray` son subtipos de `JsonStructure`. Este objeto representa la raíz del árbol y puede utilizarse para navegar por el árbol o escribirlo en un *stream* como datos JSON.

Vamos a mostrar algún ejemplo en el que utilicemos un `StringReader`.

#### Objeto JSON con dos pares *nombre-valor*

```
jsonReader = Json.createReader(new StringReader("{"
                      + "  \"manzana\":\"roja\","
                      + "  \"plátano\":\"amarillo\""
                      + "}"));
JsonObject json = jsonReader.readObject();
json.getString("manzana"); ❶
json.getString("plátano");
```

❶ El método `getString()` devuelve el valor del *string* para la clave especificada como parámetro. Pueden utilizarse otros métodos `getXXX()` para acceder al valor correspondiente de la clave en función del tipo de dicho objeto.

Un *array* con dos objetos, cada uno de ellos con un par *nombre-valor* puede leerse como:

#### Array con dos objetos

```
jsonReader = Json.createReader(new StringReader("["
                        + "  { \"manzana\":\"rojo\" },"
                        + "  { \"plátano\":\"amarillo\" }"
                        + "]"));
JsonArray jsonArray = jsonReader.readArray(); ❶
```

❶ La interfaz JsonArray también tiene métodos `get` para valores de tipo `boolean`, `integer`, y `String` en el índice especificado (esta interfaz hereda de java.util.List)

## Creación de un modelos de objetos desde el código de la aplicación

A continuación mostramos un ejemplo de código para crear un modelo de objetos mediante programación:

#### Ejemplo de creación de un modelo de objetos JSON desde programación

```
import javax.json.Json;
import javax.json.JsonObject;
...
JsonObject modelo =
    Json.createObjectBuilder() ❶
        .add("nombre", "Duke")
```

```
        .add("apellidos", "Java")
        .add("edad", 18)
        .add("calle", "100 Internet Dr")
        .add("ciudad", "JavaTown")
        .add("codPostal", "12345")
        .add("telefonos",
              Json.createArrayBuilder() ❷
                  .add(Json.createObjectBuilder()
                           .add("tipo", "casa")
                           .add("numero", "111-111-1111"))
                  .add(Json.createObjectBuilder()
                           .add("tipo", "movil")
                           .add("numero", "222-222-2222")))
            .build();
```

❶ El tipo `JsonObject` representa un objeto JSON. El método `Json.createObjectBuilder()` crea un modelo de objetos en memoria añadiendo elementos desde el código de nuestra aplicación

❷ El método `Json.createArrayBuilder()` crea un modelo de arrays en memoria añadiendo elementos desde el código de nuestra aplicación

El objeto `modelo`, de tipo `JsonObject` representa la raíz del árbol, que es creado anidando llamadas a métodos ´add()´, y construyendo el árbol a través del método `build()`.

La estructura JSON generada es la siguiente:

**Ejemplo formato JSON generado mediante programación**

```
{ "nombre": "Duke",
  "apellidos": "Java",
  "edad": 18,
  "calle": "100 Internet Dr",
  "ciudad": "JavaTown",
  "codPostal": "12345",
  "telefonos": [
         { "tipo": "casa",
           "numero": "111-111-1111"
         },
         {
           "tipo": "movil",
           "numero": "222-222-2222"
         }
  ]
 }
```

## Navegando por el modelo de objetos

A continuación mostramos un código de ejemplo para navegar por el modelo de objetos:

```
import javax.json.JsonValue;
import javax.json.JsonObject;
import javax.json.JsonArray;
import javax.json.JsonNumber;
import javax.json.JsonString;
...
public static void navegarPorElArbol(JsonValue arbol, String clave) {
```

```
    if (clave != null)
      System.out.print("Clave " + clave + ": ");
    switch(arbol.getValueType()) {
      case OBJECT:
         System.out.println("OBJETO");
         JsonObject objeto = (JsonObject) arbol;
         for (String nombre : object.keySet())
            navegarPorElArbol(object.get(nombre), name);
         break;
      case ARRAY:
         System.out.println("ARRAY");
         JsonArray array = (JsonArray) arbol;
         for (JsonValue val : array)
           navegarPorElArbol(val, null);
         break;
      case STRING:
         JsonString st = (JsonString) arbol;
         System.out.println("STRING " + st.getString());
         break;
      case NUMBER:
         JsonNumber num = (JsonNumber) arbol;
         System.out.println("NUMBER " + num.toString());
         break;
      case TRUE:
      case FALSE:
      case NULL:
         System.out.println(arbol.getValueType().toString());
         break;
      }
  }
```

El método `navegarPorElArbol()` podemos usarlo con el ejemplo anterior de la siguiente forma:

```
navegarPorElArbol(modelo, "OBJECT");
```

El método `navegarPorElArbol()` tiene dos argumentos: un elemento JSON y una clave. La clave se utiliza para imprimir los pares *clave-valor* dentro de los objetos. Los elementos en el árbol se representan por el tipo `JsonValue`. Si el elemento es un **objeto** o un **array**, se realiza una nueva llamada a este método es invocada para cada elemento contenido en el *objeto* o el *array*. Si el elemento es un **valor**, éste se imprime en la salida estándar.

El método `JsonValue.getValueType()` identifica el elemento como un *objeto*, un *array*, o un *valor*. Para los objetos, el método `JsonObject.keySet()` devuelve un conjunto de `Strings` que contienene las claves de los objetos, y el método `JsonObject.get(String nombre)` devuelve el valor del elemento cuya *clave* es `nombre`. Para los *arrays*, `JsonArray` implementa la interfaz `List<JsonValue>`. Podemos utilizar bucles `for` mejorados, con el valor de `Set<String>` devuelto por `JsonObject.keySet()`, y con instancias de `JsonArray`, tal y como hemos mostrado en el ejemplo.

## Escritura de un modelo de objetos en un *stream*

Los modelos de objetos creados en los ejemplos anteriores, pueden "escribirse" en un *stream*, utilizando la clase `JsonWriter`, de la siguiente forma:

```
import java.io.StringWriter;
import javax.json.JsonWriter;
...
StringWriter stWriter = new StringWriter();
JsonWriter jsonWriter = Json.createWriter(stWriter); ❶
jsonWriter.writeObject(modelo); ❷
jsonWriter.close(); ❸

String datosJson = stWriter.toString();
System.out.println(datosJson);
```

❶ El método `Json.createWriter()` toma como parámetro un `OutputStream`

❷ El método `JsonWriter.writeObject()` "escribe" el objeto `JsonObject` en el *stream*

❸ El método `JsonWriter.close()` cierra el *stream* de salida

## Modelo de procesamiento basado en *streaming*

El modelo de *streaming* utiliza un parser basado en eventos que va leyendo los datos JSON de uno en uno. El parser genera eventos y detiene el procesamiento cuando un objeto o array comienza o termina, cuando encuentra una clave, o encuentra un valor. Cada elemento puede ser procesado o rechazado por el código de la aplicación, y a continuación el parser continúa con el siguiente evento. Esta aproximación es adecuada para un procesamiento local, en el cual el procesamiento de un elemento no requiere información del resto de los datos. El modelo de *streaming* genera una salida JSON para un determinado *stream* realizando una llamada a una función con un elemento cada vez.

A continuación veamos con ejemplos cómo utilizar el API para el modelo de *streaming*:

- Para leer datos JSON utilizando un *parser* (*JsonParser*)
- Para escribir datos JSON utilizando un *generador* (*JsonGenerator*)

### Lectura de datos JSON

El API para el modelo *streaming* es la aproximación más eficiente para "parsear" datos JSON utilizando eventos:

```
import javax.json.Json;
import javax.json.stream.JsonParser;
...
JsonParser parser = Json.createParser(new StringReader(datosJson));
while (parser.hasNext()) {
  JsonParser.Event evento = parser.next();
  switch(evento) {
    case START_ARRAY:
    case END_ARRAY:
    case START_OBJECT:
    case END_OBJECT:
    case VALUE_FALSE:
    case VALUE_NULL:
    case VALUE_TRUE:
        System.out.println(evento.toString());
```

```
            break;
        case KEY_NAME:
            System.out.print(evento.toString() + " " + parser.getString() + "
 - ");
            break;
        case VALUE_STRING:
        case VALUE_NUMBER:
            System.out.println(evento.toString() + " " + parser.getString());
            break;
    }
}
```

El ejemplo consta de tres pasos:

1. Obtener una instancia de un parser invocando el método estático `Json.createParser()`
2. Iterar sobre los eventos del parser utilizando los métodos `JsonParser.hasNext()` y `JsonParser.next()`
3. Realizar un procesamiento local para cada elemento

El ejemplo muestra los diez posibles tipos de eventos del parser. El método `JsonParser.next()` "avanza" al siguiente evento. Para los tipos de eventos `KEY_NAME`, `VALUE_STRING`, y `VALUE_NUMBER`, podemos obtener el contenido del elemento invocando al método `JsonParser.getString()`. Para los eventos `VALUE_NUMBER`, podemos también usar los siguientes métodos:

- `JsonParser.isIntegralNumber`
- `JsonParser.getInt`
- `JsonParser.getLong`
- `JsonParser.getBigDecimal`

El parser genera los eventos `START_OBJECT` y `END_OBJECT` para un objeto JSON vacío: `{ }`.

Para un objeto con dos pares *nombre-valor*:

```
{
  "manzaja":"roja", "plátano":"amarillo"
}
```

Mostramos los eventos generados:

```
{START_OBJECT
   "manzaja"KEY_NAME:"roja"VALUE_STRING,
   "plátano"KEY_NAME:"amarillo"VALUE_STRING
}END_OBJECT
```

Los eventos generados para un *array* con dos objetos JSON serían los siguientes:

```
[START_ARRAY
  {START_OBJECT "manzaja"KEY_NAME:"roja"VALUE_STRING }END_OBJECT,
```

```
  {START_OBJECT "plátano"KEY_NAME:"amarillo"VALUE_STRING }END_OBJECT
]END_ARRAY
```

### Escritura de datos JSON

El siguiente código muestra cómo escribir datos JSON en un fichero utilizando el API para el modelo de *streaming*:

**Ejemplo de escritura de datos JSON con el modelo de streaming**

```
FileWriter writer = new FileWriter("test.txt");
JsonGenerator gen = Json.createGenerator(writer);
gen.writeStartObject()
   .write("nombre", "Duke")
   .write("apellidos", "Java")
   .write("edad", 18)
   .write("calle", "100 Internet Dr")
   .write("ciudad", "JavaTown")
   .write("codPostal", "12345")
   .writeStartArray("telefonos")
      .writeStartObject()
         .write("tipo", "casa")
         .write("numero", "111-111-1111")
      .writeEnd()
      .writeStartObject()
         .write("tipo", "movil")
         .write("numero", "222-222-2222")
      .writeEnd()
   .writeEnd()
   .writeEnd();
gen.close();
```

Este ejemplo obtiene un generador JSON invocando al método estático `Json.createGenerator()`, que toma como parámetro un *output stream* o un *writer stream*. El ejemplo escribe los datos JSON en el fichero `text.txt` anidando llamadas a los métodos `write()`, `writeStartArray()`, `writeStartObject()`, and `writeEnd()`. El método `JsonGenerator.close()` cierra el *output stream* o *writer stream* subyacente.

## 5.4. Pruebas de servicios REST

Hasta ahora hemos visto varias formas de "probar" nuestros servicios REST: desde línea de comandos con *Curl*, desde IntelliJ con la herramienta *Test RESTFul Web Service*, y desde el navegador *Chrome*, con *Postman* (siendo esta última la que más hemos utilizado).

Vamos a ver cómo implementar tests para nuestros servicios REST utilizando Maven y JUnit. Para ello repasaremos algunas cuestiones básicas sobre los ciclos de vida de Maven[10].

### Ciclo de vida de Maven y tests JUnit

Un ciclo de vida en Maven **es una secuencia de acciones determinada**, que define el proceso de construcción de un proyecto en concreto. Como resultado del proceso de construcción de un proyecto obtendremos un *artefacto* (fichero), de un cierto tipo (por ejemplo .jar, .war, .ear,…). Por lo tanto, podríamos decir que un ciclo de vida está formado por

[10] https://maven.apache.org/ref/3.3.3/maven-core/lifecycles.html

las acciones necesarias para convertir nuestros archivos fuente que constituyen el proyecto en, por ejemplo un .jar, un .war,…

Maven propone 3 ciclos de vida, es decir, tres posibles secuencias de acciones, que podemos utilizar (y modificar a nuestra conveniencia) para construir nuestro proyecto. Dichos ciclos de vida son: *clean*, *site* y el denominado *default-lifecycle*.

Cada ciclo de vida está formado por **fases**. Una fase es un concepto abstracto, y define el tipo de acciones que se deberían llevar a cabo. Por ejemplo una fase del ciclo de vida por defecto es *compile*, para referirse a las acciones que nos permiten convertir los ficheros *.java* en los ficheros *.class* correspondientes.

Cada fase está formada por un conjunto de **goals**, que son las acciones que se llevarán a cabo en cada una de las fases. Las **goals** no "viven" de forma independiente, sino que cualquier *goal* siempre forma parte de un **plugin** Maven. Podríamos decir que un *plugin*, por lo tanto, es una agrupación lógica de una serie de *goals* relacionadas. Por ejemplo, el *plugin* wildfly, contiene una serie de *goals* para *desplegar*, *re-desplegar*, *deshacer-el-despliegue*, *arrancar el servidor*, etc., es decir, agrupa las acciones que podemos realizar sobre el servidor wildfly. Una goal se especifica siempre anteponiendo el nombre del *plugin* al que pertenece seguido de dos puntos, por ejemplo **wildfly:deploy** indica que se trata de la *goal deploy*, que pertenece al plugin *wildfly* de maven.

Pues bien, por defecto, Maven asocia ciertas *goals* a las fases de los tres ciclos de vida. Cuando se ejecuta una fase de un ciclo de vida, por ejemplo **mvn package** se ejecutan todas las *goals* asociadas a todas las fases anteriores a la fase **package**, en orden, y finalmente las *goals* asociadas a la fase **package**. Por supuesto, podemos alterar en cualquier momento este comportamiento por defecto, incluyendo los plugins y goals correspondientes dentro de la etiqueta *<build>* en nuestro fichero de configuración *pom.xml*.

Vamos a implementar tests JUnit. Los tests, como ya habéis visto en sesiones anteriores, en el directorio **src/test**. Algunas normas importantes son: que los tests pertenezcan al mismo paquete lógico al que pertencen las clases Java que estamos probando. Por ejemplo, si estamos haciendo pruebas sobre las clases del paquete *org.expertojava.rest*, los tests deberían pertenecer al mismo paquete, aunque físicamente el código fuente y sus pruebas estarán separados (el código fuente estará en src/main, y los tests en src/test).

Para realizar pruebas sobre nuestros servicios REST, necesitamos que el servidor Wilfly esté en marcha. También necesitamos empaquetar el código en un fichero **war** y desplegarlo en el servidor, todo esto ANTES de ejecutar los tests.

Las acciones para arrancar el servidor Wilfly y desplegar nuestra aplicación en él, NO forman parte de las acciones (o *goals*) incluidas por defecto en el ciclo de vida por defecto de Maven, cuando nuestro proyecto tiene que empaquetarse como un **war** (etiqueta *<packaging>* de nuestro *pom.xml*). Podéis consultar aquí[11] la lista de *goals* asociadas a las fases del ciclo de vida por defecto de Maven.

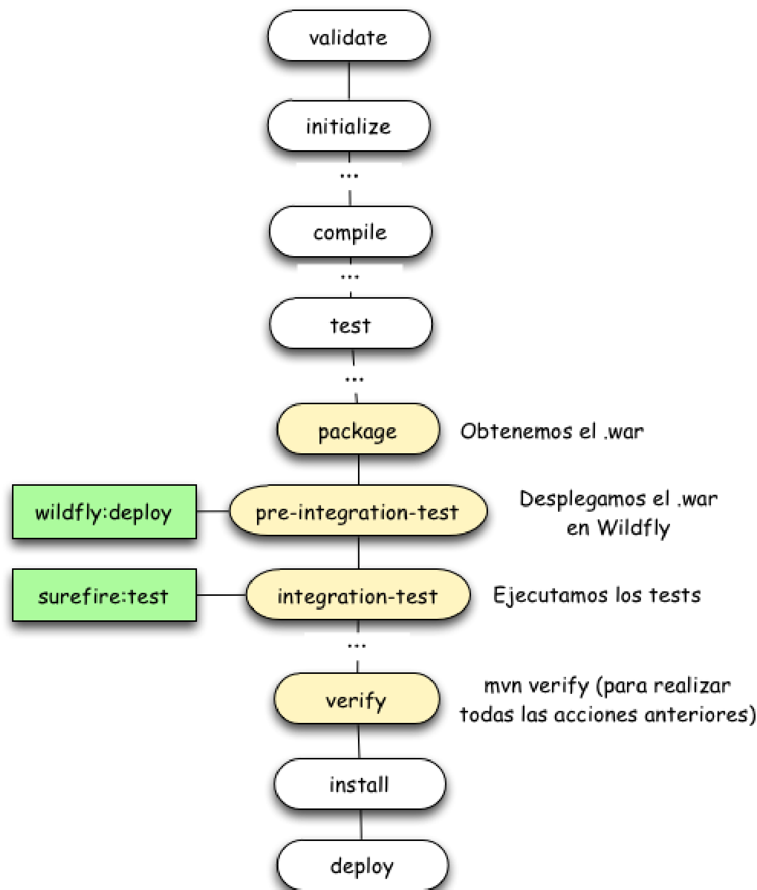
Por otro lado, en el ciclo de vida por defecto, se incluye una *goal* para ejecutar los tests, asociada a la fase *test*. Dicha goal es **surefire:test**. El problema es que, por defecto, la fase *test* se ejecuta ANTES de la fase *package* y por lo tanto, antes de empaquetar y desplegar nuestra aplicación en Wildfly.

Por lo tanto, tendremos que "alterar" convenientemente este comportamiento por defecto para que se ejecuten las acciones de nuestro proceso de construcción que necesitemos,

[11] https://maven.apache.org/ref/3.3.3/maven-core/default-bindings.html#Plugin_bindings_for_war_packaging

y en el orden en el que lo necesitemos. Como ya hemos indicado antes, esto lo haremos incluyendo dichas acciones en la etiqueta *<build>* de nuestro *pom.xml*, y configurandolas convenientemente para asegurarnos que el orden en el que se ejecutan es el que queremos.

La siguiente figura muestra parte de la secuencia de fases llevadas a cabo por Maven en su ciclo de vida por defecto. Para conseguir nuestros propósitos, simplemente añadiremos la "goals" **wildfly:deploy**, y la asociaremos a la fase *pre-integration-test*, y "cambiaremos" la fase a la que está asociada la goal **surefire:test** para que los tests se ejecuten DESPUÉS de haber desplegado el *war* en Wildfly.




A continuación mostramos los cambios que tenemos que realizar en el fichero de configuración **pom.xml**

### Adición de las goals wildfly:deploy y surefire:test a las fases pre-integration-test y surefire:test respectivamente

```
<!-- forzamos el despliegue del war generado durante la fase pre-
integration-test,
     justo después de obtener dicho .war-->
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>1.0.2.Final</version>
  <configuration>
    <hostname>localhost</hostname>
    <port>9990</port>
  </configuration>
```

```
    <executions>
      <execution>
        <id>wildfly-deploy</id>
        <phase>pre-integration-test</phase>
        <goals>
          <goal>deploy</goal>
        </goals>
        </execution>
    </executions>
  </plugin>

  <!--ejecutaremos los test JUnit en la fase integration-test,
      inmediatamente después de la fase pre-integration-test, y antes
      de la fase verify-->
  <plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-surefire-plugin</artifactId>
     <version>2.18</version>
     <configuration>
          <skip>true</skip>
     </configuration>
     <executions>
        <execution>
          <id>surefire-it</id>
          <phase>integration-test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <skip>false</skip>
          </configuration>
        </execution>
      </executions>
  </plugin>
```

También necesitamos incluir en el pom.xml las librerías de las que depende el código de pruebas de nuestro proyecto (clases **XXXXTest** situadas en **src/test**): librería JUnit, JAXB y el API cliente de JAX-RS. Por lo que añadimos en el las dependencias correspondientes:

**Dependencias del código src/test con JUnit y API cliente de JAX-RS**

```
 ...
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.12</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>3.0.13.Final</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
```

```
    <artifactId>resteasy-jaxb-provider</artifactId>
    <version>3.0.13.Final</version>
</dependency>
...
```

Dado que vamos a trabajar con el API Json, y dado que ejecutaremos los tests desde la máquina virtual de Java, y no dentro del servidor WildFly, necesitamos añadir también las siguientes librerías:

**Dependencias del código src/test con el API Json de jaxrs**

```
...
<!--Librerías para serializar/deserializar json -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-jackson-provider</artifactId>
    <version>3.0.13.Final</version>
</dependency>

<!--Jaxrs API json -->
<dependency>
    <groupId>org.jboss.resteasy</groupId>
    <artifactId>resteasy-json-p-provider</artifactId>
    <version>3.0.13.Final</version>
</dependency>
...
```

No hemos incluido en el pom.xml la orden para arrancar Wildfly. Vamos a hacer esto desde IntelliJ en un perfil de ejecución, como ya habéis hecho en sesiones anteriores. De esta forma, podremos ver desde IntelliJ la consola de logs del servidor. En este caso, podemos crear un perfil solamente para arrancar el servidor Wildfly (no es necesario que se incluya el despliegue del war generado, puesto que lo haremos desde la ventana *Maven Projects*). Antes de iniciar el proceso de construcción, por lo tanto, tendremos que asegurarnos de que hemos arrancado Wildlfly.

Con estos cambios en el pom.xml, y ejecutando el comando `mvn verify` se llevarán a cabo las siguientes acciones, en este orden:

- Después de compilar el proyecto, obtenemos el `.war` (fase `package`)
- El `.war` generado se despliega en el servidor de aplicaciones Wilfly (fase `pre-integration-test`)
- Se ejecutan los test JUnit sobre la aplicación desplegada en el servidor (fase `integration-test`)

## Anotaciones JUnit y aserciones AssertThat

JUnit 4 proporciona anotaciones para forzar a que los métodos anotados con @Test se ejecuten en el orden que nos interese (por defecto no está garantizado que se ejecuten en el orden en el que se escriben).

En principio, debemos programar los tests para que sean totalmente independientes unos de otros, y por lo tanto, el orden de ejecución no influya para nada en el resultado de su ejecución, tanto si se ejecuta el primero, como a mitad, o el último. El no hacer los tests independientes

hace que el proceso de testing "se alargue" y complique innecesariamente, ya que puede ser que unos tests "enmascaren" en resultado de otros, o que no podamos "saber" si ciertas partes del código están bien o mal implementadas hasta que los tests de los que dependemos se hayan superado con éxito.

Aún así, y dado que muchas veces se obtienen errores por hacer asunciones en el orden de la ejecución de los tests, JUnit nos permite fijar dicho orden. Para ello utilizaremos la anotación `@FixMethodOrder`, indicando el tipo de ordenación, como por ejemplo `MethodSorters.NAME_ASCENDING`, de forma que se ejecutarán los tests por orden lexicográfico.

Por ejemplo:

**Ejemplo para forzar el orden de ejecución de los test (orden lexicográfico)**

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class TestMethodOrder {

    @Test
    public void testB() {
       System.out.println("second");
     }

    @Test
    public void testA() {
        System.out.println("first");
    }

    @Test
    public void testC() {
        System.out.println("third");
    }
}
```

En ese caso, el orden de ejecución será: `testA()`, a continuación `testB()`, y finalmente `testC()`.

Otra aportación de JUnit 4 es la incorporación de aserciones de tipo `assertThat`. En sesiones anteriores habéis utilizado aserciones con métodos `Assert.assertEquals(resultado_esperado, resultado_real)`. Los nuevos métodos `Assert.assertThat()` permiten una mayor flexibilidad a la hora de expresar las aserciones realizadas en nuestros tests, así como una mayor legibilidad de los mismos. El prototipo general de las aserciones de este tipo es:

```
assertThat([value], [matcher statement]);
```

en donde `[value]` es el resultado real (valor sobre el que se quiere afirmar algo), y `[matcher statement]` es un `Matcher` u objeto que realiza operaciones de "emparejamiento" sobre una secuencia de caracteres según un determinado patrón.

Por ejemplo:

**Ejemplos de sentencias assertThat**

```
assertThat(x, is(not(4)));   ❶
```

```
assertThat(responseStringJson,
 either(containsString("nombre")).and(containsString("apellido"))); ❷
assertThat(myList, hasItem("3")); ❸
```

❶ Aquí utilizamos un *matcher* con el patrón "4", esta sentencia devuelve *false* si x != "4"
❷ Podemos "combinar" varios *matchers*, de forma que se tengan que satisfacer más de uno
❸ En este caso aplicamos el *matcher* sobre un conjunto de elementos

Hay varias librerías que implementan *Matchers*. JUnit incluye parte de los *matchers* de *Hamcrest* (*Hamcrest* es un *framework* para escribir objetos *matcher* permitiendo definir reglas de *matching* de forma declarativa). Otra librería interesante para realizar testing de servicios rest que utilizan representaciones Json es la librería: `hamcrest-json`, que podemos utilizar para realizar aserciones sobre dos objetos Json.

Por ejemplo, supongamos que nuestro objeto Json contiene una lista de enlaces Hateoas, de tipo `Link`. Los objetos `Link` serán serializados/deserializados (Wildfly utiliza *Jackson* para realizar estas tareas) convenientemente. Cuando serializamos un objeto `Link` (obtenemos su representación Json), veremos, además de los objetos "uri": "valor", "type": "valor" y "rel": "valor", que son los que básicamente utilizamos al crear los enlaces Hateoas, otros como "uriBuilder": {…}, "params":{…}, que puede que no nos interese consultar, o incluso que no les hayamos asignado ningún valor.

Si en nuestro test, queremos comprobar que el objeto Json que nos devuelve el servicio (resultado real) se corresponde con el valor esperado, tendremos que "comparar" ambas representaciones. Ahora bien, puede que solamente nos interese comparar ciertos valores contenidos en el objeto Json, no el objeto "completo".

Hacer esta comprobación "elemento a elemento" es bastante tedioso. La librería `hamcrest-json` nos proporciona lo que estamos buscando, con los métodos `sameJSONAs()`, `allowingExtraUnexpectedFields()`, y `allowingAnyArrayOrdering()`, de la siguiente forma:

**Método para comparar dos representaciones Json. Clase uk.co.datumedge.hamcrest.json.SameJSONAs**

```
Assert.assertThat("{\"age\":43, \"friend_ids\":[16, 52, 23]}",
                  sameJSONAs("{\"friend_ids\":[52, 23, 16]}")
                            .allowingExtraUnexpectedFields()
                            .allowingAnyArrayOrdering());
```

En este código tenemos una representación formada por dos objetos, uno de los cuales tiene como valor un array de enteros. Si el servicio rest devuelve un objeto Json con más elementos, o en otro orden, en este caso el resultado de la sentencia *assertThat* es *true*. Volviendo al ejemplo anterior de un objeto Json que contiene enlaces Hatehoas, podríamos realizar la siguiente comparación:

**Comparamos dos objetos Json que contienen hiperenlaces Hateoas (objetos Link)**

```
JsonObject json_object =
    client.target("http://localhost:8080/foro/usuarios")
          .request(MediaType.APPLICATION_JSON)
          .get(JsonObject.class);    ❶

String json_string = json_object.toString();   ❷
```

```
JsonObject usuarios =
   Json.createObjectBuilder()
       .add("usuarios",
           Json.createArrayBuilder()
              .add(Json.createObjectBuilder()
                  .add("nombre", "Pepe Lopez")
                  .add("links",
                      Json.createArrayBuilder()
                        .add(Json.createObjectBuilder()
                        .add("uri", "http://localhost:8080/foro/usuarios/
pepe")
                        .add("type", "application/xml,application/json")
                        .add("rel", "self"))))
              .add(Json.createObjectBuilder()
                  .add("nombre", "Ana Garcia")
                  .add("links",
                      Json.createArrayBuilder()
                        .add(Json.createObjectBuilder()
                        .add("uri", "http://localhost:8080/foro/usuarios/
ana")
                        .add("type", "application/xml,application/json")
                        .add("rel", "self")))))
    .build();    ❸

Assert.assertThat(json_string,
                  sameJSONAs(usuarios.toString())
                      .allowingExtraUnexpectedFields()
                      .allowingAnyArrayOrdering());    ❹
```

❶ Realizamos la llamada al servicio REST y recibimos como respuesta un objeto Json. En este caso nuestro objeto Json está formado por una lista de objetos.

❷ Obtenemos la representación de nuestro objeto Json (resultado real) en forma de cadena de caracteres

❸ Creamos un nuevo objeto Json con el resultado esperado

❹ Comparamos ambos objetos. Si el resultado real incluye más elementos que los contenidos en `json_string` o en otro orden consideraremos que hemos obtenido la respuesta correcta.

Para utilizar esta librería en nuestro proyecto, simplemente tendremos que añadirla como dependencia en la configuración de nuestro pom.xml:

**Librería para comparar objetos Json en los tests**

```
<!--Hamcrest Json -->
<dependency>
    <groupId>uk.co.datumedge</groupId>
    <artifactId>hamcrest-json</artifactId>
    <version>0.2</version>
</dependency>
```

## Observaciones sobre los tests y algunos ejemplos de tests

Recuerda que para utilizar el API cliente, necesitas utilizar instancias `javax.ws.rs.client.Client`, que debemos "cerrar" siempre después de su uso para cerrar el socket asociado a la conexión.

Para ello podemos optar por: * Crear una única instancia Client "antes" de ejecutar cualquier test (método @BeforeClass), y cerrar el socket después de ejecutar todos los tests (método @AfterClass) * Crear una única instancia Client "antes" de ejecutar CADA test (método @Before), y cerrar el socket después de ejecutar CADA tests (método @After)

Si el resultado de una invocación sobre la instancia *Client* es de tipo `javax.ws.rs.core.Response`, debemos liberar de forma explícita la conexión para que pueda ser usada de nuevo por dicha instancia *Client*.

Por ejemplo, supongamos que queremos realizar un test en el que realizamos una operación POST, y a continuación una operación GET para verificar que el nuevo recurso se ha añadidido correctamente:

**Ejemplo de Test que utiliza una instancia *Client* para todos los tests**

```
public class TestRESTServices {
  private static final String BASE_URL = "http://localhost:8080/rest/";
  private static URI uri = UriBuilder.fromUri(BASE_URL).build();
  private static Client client;

  @BeforeClass
  public static void initClient() {
    client = ClientBuilder.newClient(); ❶
  }

  @AfterClass
  public static void closeClient() {
    client.close(); ❷
  }

  @Test
  public void createAndRetrieveACustomer() {

    Customer customer = ... //Creamos un nuevo cliente
    Response response = client.target(uri)
                              .request()
                              .post(Entity.entity(customer,
 MediaType.APPLICATION_JSON));
    assertEquals(Response.Status.CREATED, response.getStatusInfo());
    URI referenceURI = response.getLocation();
    response.close(); ❸

    // Obtenemos el recurso que hemos añadido
    response = client.target(referenceURI).request().get();

    Customer retrieved_customer = response.readEntity(Customer.class);
    assertEquals(Response.Status.OK, response.getStatusInfo());
    assertEquals(retreivedRef.getName(), r.getName());
    response.close(); ❹
  }
}
```

❶ Creamos una instancia *Client* ANTES de ejecutar cualquier test

❷ Cerramos el socket asociado a la conexión DESPUÉS de ejecutar TODOS los tests

❸ Liberamos la conexión para poder reutilizarla

❹ Liberamos la conexión para poder reutilizarla

Ahora veamos otro ejemplo en el que utilizamos una instancia *Client* para cada test:

**Ejemplo de Test que utiliza una instancia *Client* para CADA test**

```
public class TestRESTServices {

   private Client client;

   @Before
   public void setUp() {
        this.client = ClientBuilder.newClient(); ❶
   }

   @After
   public void tearDown() {
        this.client.close();  ❷
   }

   @Test
   public void getAllCustomersAsJson() {
      String uriString = "http://localhost:8080/rest/customers";
      JsonArray json_array = client
                 .target(uriString)
                 .request(MediaType.APPLICATION_JSON)
                 .accept(MediaType.APPLICATION_JSON)
                 .get(JsonArray.class);

      Assert.assertEquals(2, json_array.size());
    }

    @Test
    public void getAllCustomers() {
     String uriString = "http://localhost:8080/rest/customers";
     //Consultamos los datos de todos los customers
     List<Customer> lista_usuarios = client.target(uriString)
                 .request("application/json")
                 .get(new GenericType<List<Customer>>() {});
     Assert.assertEquals(2, lista_usuarios.size());
   }
}
```

❶ Creamos una instancia *Client* ANTES de ejecutar CADA test

❷ Cerramos el socket asociado a la conexión DESPUÉS de ejecutar CADA los tests

Podemos ver en este último ejemplo que no es necesario liberar la conexión entre usos sucesivos de la instancia *Client*, si no utilizamos la clase *Response*. En este caso el proceso se realiza de forma automática por el sistema.

Finalmente comentaremos que, debido a un bug en la especificación JAX-RS, el deserializado del de los objetos Link no se realiza, por lo que obendremos una lista de Links vacía (ver http://kingsfleet.blogspot.com.es/2014/05/reading-and-writing-jax-rs-link-objects.html). Podemos comprobar que, si obtenemos la representación en formato texto de la entidad del mensaje, dicha lista de objetos tendrá el valor correcto.

Si no utilizamos la solución propuesta en el enlace anterior, deberemos usar la anotación `@JsonIgnoreProperties(ignoreUnknown = true)`. De esta forma, ignoraremos el

deserializado de los objetos Link, pero tendremos que utilizar la representación en formato de cadena de caracteres del recurso json, en lugar del objeto java *Link* asociado.

Así, por ejemplo, si nuestro recurso *Customer* tiene asociado una lista de objetos *Link*, para poder utilizar el API Cliente y acceder a la lista de enlaces, usaremos la anotación anterior en la implementación de la clase *Customer*:

```
@JsonIgnoreProperties(ignoreUnknown = true)
@XmlRootElement(name="customer")
public class Customer {
  int id;
  String name;
  ...
  List<Link> links;
  ...
}
```

## 5.5. Ejercicios

### Tests utilizando el API cliente y un mapeador de excepciones (1 punto)

Se proporciona como plantilla el MÓDULO IntelliJ "s5-tienda" con una implementación parcial de una tienda de clientes on-line. Este proyecto ya contiene varios tests implementados, a modo de ejemplo.

Los recursos rest implementados lanzan excepciones de tipo `RestException` si, por ejemplo se intenta realizar una consulta sobre un producto y/o usuario que no existe.

Se ha implementado un mapeador de excepciones `RestExceptionMapper` que captura excepciones de tipo `RuntimeException`, y devuelve una respuesta de tipo `ErrorMensajeBean` que será serializada a formato json y/o formato xml (dependiendo del valor de la cabecera *Accept* de la petición), con información sobre el error producido.

Implementa los siguientes dos tests: * `test7recuperarTodosLosUsuarios()`, en el que realizamos una invocación GET sobre "http://localhost:8080/s5-tienda/rest/clientes/". Esta URI podría corresponderse con un método anotado con @GET y que devolviese una lista de todos los clientes de la tienda. Sin embargo, no existe tal método en nuestro recursos rest. Verifica que dicha invocación devuelve el código de estado "500" (Internal Server Error), y que en el cuerpo del mensaje se recibe "Servicio no disponible"

- `test8recuperarClienteQueNoExiste()`, en el que intentamos recuperar la información de un cliente que no exista en nuestra base de datos. En este caso, debemos verificar que el mensaje obtenido en formato json es el siguiente:

```
{
  "status": "Not Found",
  "code": 404,
  "message": "El producto no se encuentra en la base de datos",
  "developerMessage": "error"
}
```

### Tests utilizando el API Json y JUnit (1 punto)

Vamos a seguir usando el proyecto s4-foroAvanzado con el que hemos trabajado en la sesión anterior.

Vamos a implementar algunos tests con JUnit en los que utilizaremos, además del API cliente, el API Json, que nos proporciona jaxrs.

Para ejecutar los tests necesitamos modificar el pom.xml añadiendo las dependencias correspondientes que hemos visto a lo largo de la sesión, y añadiendo las goals para que se ejecuten los tests después de desplegar la aplicación en Wildfly.

Proporcionamos el contenido del *pom.xml* con las librerías y plugins que necesitarás (aunque como ejercicio deberías intentar modificar la configuración tú mismo, y luego puedes comprobar el resultado con el *pom.xml* que se proporciona). El contenido del nuevo pom.xml lo tienes en /src/test/resources/nuevo-pom.mxl.

#### Inicialización de los datos para los tests

Vamos a utilizar DBUnit para inicializar la BD para realizar los tests. Para ello tendrás que añadir en el pom.xml las dependencias necesarias (ya están añadidas en el fichero de

configuración proporcionado). En el fichero *src/test/resources/foro-inicial.xml* encontraréis el conjunto de datos con el que inicializaremos la base de datos para ejecutar nuestros tests.

No es necesario (aunque es una muy buena práctica) que inicialicemos la BD para cada test.

## Implementación de los tests

Vamos a implementar los siguientes tests (que se ejecutarán en en este mismo orden):

- `test1ConsultaTodosUsuarios()`: recuperamos los datos de todos los usuarios del foro. Recuerda que previamente tienes que haber inicializado la BD con los datos del fichero *foro-inicial.xml*. Recupera los datos en forma de JsonObject y comprueba que el número de usuarios es el correcto. También debes comprobar que tanto el login, como los enlaces hatehoas para cada usuario están bien creados. En concreto, para cada usuario, debes verificar que la uri ("uri"), el tipo mime ("type"), y el tipo de enlace ("rel") son los correctos.
- `test2CreamosMensajeDePepe()`: crearemos un nuevo mensaje del usuario con login "pepe". Recuerda que este usuario tiene el rol "registrado". El mensaje tendrá el asunto "cena", y el texto será: "Mejor me voy al cine". En este caso, deberás comprobar el valor de estado (debe ser 201), y debes recuperar (consultar con una petición REST) el mensaje para comprobar que la operación de insertar el mensaje ha tenido éxito.
- `test3CreamosMensajeDeUsuarioNoAutorizado()`: creamos un nuevo mensaje de un usuario que no está autorizado (por ejemplo, de un usuario con login "juan"). En este caso el mensaje tendrá el asunto "cena", y el mensaje puede ser: "Pues yo tampoco voy". El resultado debe ser el código de estado 401 ( Unauthorized)
- `test4ConsultaUsuario()`: Consultamos los datos del usuario "pepe". Recuperaremos los datos como un JsonObject, y comprobaremos que el valor de la "uri" para el tipo de relación "self" del enlace *Link* asociado es el correcto