Servicios Rest

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

Table of Contents

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 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.

1.1. ¿Qué es un servicio Web?

El diseño del software tiende a ser cada vez más modular. Las aplicaciones se componen de 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 requieren 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).

  # Orientado 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 soprtar 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 `&`.

Un ejemplo de URI podría ser éste:

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

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 condifican como una secuencia de bits siguiendo un esquema de codificación hexadecimal, de forma que cada dos dígitos hexadecimales van precedidos por el carácter `%`.

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/cliente/1"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/2"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/4"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/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. ////ver ejemplo pag 10 en Restful seb services cookbook. Subbu Allamaraju

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 (aunque nosotros nos vamos a centrar solamente en las cuatro anteriores):

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.

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**: Finalmente, y 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.

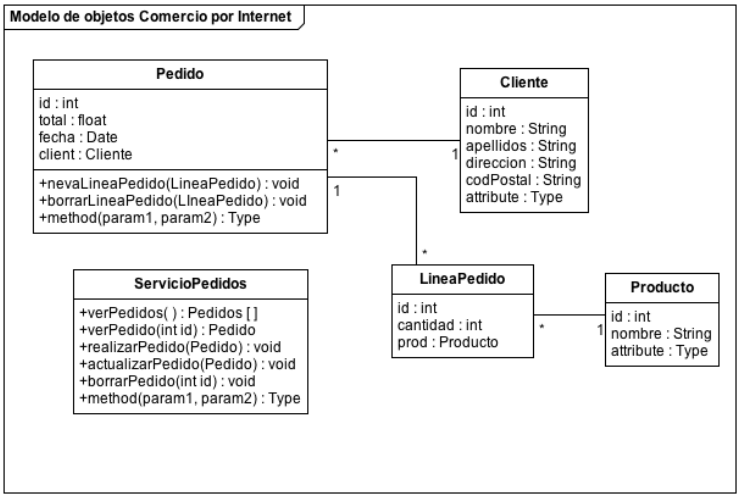
1.4. Un primer servicio JAX-RS

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 nuestros clientes. 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 *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 de un *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 un *Cliente* o un *Producto*, se proporciona la URI necesaria para interactuar con cada uno de dichos recursos.

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:

```
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:

```
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:

```
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.

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.

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.

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 dsatos. La operación de cancelar 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/purga HTTP/1.1
```

Un efecto interesante de lo que acabamos de hacer es que, puesto que ahora `purga` es una URI, podemos hacer que la interfaz de nuestro servicios RESTful evolucionen con el tiempo. Por ejemplo, la orden **GET /pedidos/purga** 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 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.

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 proyeco
- `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

Visualización de artefactos generados por Maven

> Por defecto, los ficheros generados por Maven durante el proceso de construcción del proyecto (contenido del directorio *target*) no se visualizan en la ventana *Project*. Para poder visualizar el contenido del disco duro, hacemos lo siguiente: * Seleccionamos el proyecto * Accedemos a *File#Project Structure*, y seleccionamos *Project Settings#Modules*, y seleccionamos el módulo. Veremos en la parte derecha los directorios que contienen los fuentes del proyecto (en azul), los directorios que contienen los fuentes de tests (en verde), y en rojo estarán indicados los directorios que no se visualizan (*Exlcuded Folders*). Pinchamos sobre la "X" correspondiente al directorio *target*, con lo que dicho directorio ahora será visible en la vista *Projects* (cuando sea generado por el proceso de construcción de Maven)

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>
    <java.min.version>1.7</java.min.version>
    <version.javaee_api>7.0</version.javaee_api>
    <version.wildfly>8.1.0.Final</version.wildfly>
    <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. La etiqueta

Librerías utilizadas para construir el proyecto

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>${version.javaee_api}</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>${java.min.version}</source>
            <target>${java.min.version}</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>2.1.1</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

```
package org.expertojava;

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; }
}
```

Clases de nuestro servicio RESTful: ClienteResource.java

Una vez definido el objeto de nuestro dominio que representará a un *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.

Como veremos en la siguiente sesión, los servicios JAX-RS pueden ser objetos *singletons* o *per-request*. **Per-request** significa que se crea un objeto Java para procesar **cada** petición de entrada, y se "destruye" cuando la petición se ha servido. *Per-request* también implica "sin estado", ya que no se guarda el estado del servicio entre peticiones.

De momento, vamos a asumir que son objetos *per-request*, de forma que delegamos en JAX-RS para que cree las instancias correspondientes (una por petición) en cuanto se realicen invocaciones sobre nuestro servicio RESTful.

Comencemos con la implementación del servicio:

```
package org.expertojava;

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 *ClienteResource es un _singleton* y tendrá peticiones concurrentes de acceso a los clientes. 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. 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. 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")
public Response crearCliente(InputStream is) {
  //leemos los datos del cliente del body del mensaje HTTP
  Cliente cliente = leercliente(is); ❷
  idContador++;
  cliente.setId(idContador);
  clienteDB.put(cliente.getId(), cliente);  ❸
  System.out.println("Cliente creado " + cliente.getId());

  return Response.created(URI.create("/clientes/"
              + cliente.getId())).build();  ❹
   }
```

❶ se recibe una petición POST
❷ se parsea el documento del cuerpo de la petición de entrada: **is**, y creamos un objeto *Cliente* a partir de dicho documento
❸ se añade el nuevo objeto *Cliente* a nuestro "mapa" de clientes (**clienteDB**)
❹ 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 **(3)**

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 *java.io.InputStream*. 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. En este caso, queremos acceder al cuerpo del mensaje en su forma más básica, un *InputStream*.

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.

La implementación del método lee y transforma el mensaje XML de la petición POST en un objeto *Cliente*, y lo almacena en nuestro mapa *clienteDB*. Para ello utiliza el método `leerCliente(is)`. Este método es el responsable de leer un texto XML a partir de un *InputStream*. Para leer el documento XML se puede utilizar la clase *javax.xml.parsers.DocumentBuilder* para analizar ("parsear") el *InputStream* que se pasa como entrada y construir un objeto *Cliente*. Más adelante, veremos que es posible, utilizando anotaciones *JAXB* realizar este análisis del documento XML y conversión a un objejo Java de forma automática.

El método 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 esta clase.

Consulta de clientes

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

```
@GET
@Path("{id}")
@Produces("application/xml")
public StreamingOutput recuperarClienteId(@PathParam("id") int id) {
    final Cliente cli = clienteDB.get(id);
    if (cli == null) {
        throw new WebApplicationException(Response.Status.NOT_FOUND);
    }
    return new StreamingOutput() {
        public void write(OutputStream outputStream)
                  throws IOException, WebApplicationException {

            escribirCliente(outputStream, cli);
        }
```

```
    };
}
```

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.

A continuación escribimos la respuesta para el cliente **manualmente** utilizando un `java.io.OutputStream`. En JAX-RS, cuando queremos enviar nuestra respuesta utilizando *streaming* manualmente, debemos implementar y devolver una instancia de la interfaz **javax.ws.rs.core.StreamingOutput** en nuestro método JAX-RS. La interfaz **StreamingOutput** es una interfaz *callback* con el método *callback* **write()**.

En la última línea del método **recuperarClienteId()**, implementamos y devolvemos una instancia de una clase interna que implementa **StreamingOutput**. Cuando nuestro proveedor de JAX-RS (en la siguiente sesión veremos que se trata de un *servlet*) está listo para enviar un mensaje en el cuerpo de la respuesta HTTP al cliente través de la red, efectuará una llamada al método **write** que hemos implementado para devolver la representación XML de nuestro objeto *Cliente*.

En general, no usaremos la interfaz **StreamingOutput** para devolver la respuesta al cliente. Más adelante veremos que JAX-RS dispone de varios manejadores de contenidos que pueden, de formam automática, convertir objetos Java en el formata que decidamos enviar a través de la red.

Así, por ejemplo, los tipos Java **String** y **char[]**, son convertidos **automáticamente** a **texto** (cabecera `Content-Type: /text/plain`), y viceversa

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,
                             InputStream is) {
    Cliente nuevo = leerCliente(is);
    Cliente actual = clienteDB.get(id);
    if (actual == null)
       throw new WebApplicationException(Response.Status.NOT_FOUND);
    actual.setNombre(nuevo.getNombre());
    actual.setApellidos(nuevo.getApellidos());
```

```
    actual.setDireccion(nuevo.getDireccion());
    actual.setCodPostal(nuevo.getCodPostal());
    actual.setCiudad(nuevo.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 `InputStream` que nos permitirá leer el documento XML que se pasa como entrada en la petición PUT. Recordemos que, al igual que con **recuperarClienteId()**, un parámetro que no esté anotado con una anotación JAX-RS se considera que representa el cuerpo del mensaje de entrada.

En la primera parte de la implementación del método, leemos el documento XML de entrada y creamos un objeto *Cliente* a partir de él. A continuación el método intenta encontrar un objeto *Cliente* en nuestro mapa **clienteDB**. Si no existe, provocamos una **WebApplicationException** que enviará una respuesta al cliente 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*. 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 tienda ❶
mvn package ❷
./usr/local/wildfly/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

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.

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 tienda#Plugins#wildfly#wildfly:deploy

Probando nuestro servicio

Podemos probar nuestro servicio de varias formas. Vamos a mostrar como hacerlo directamente desde línea de comandos, o bien utilizando IntelliJ.

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/tienda/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. 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/tienda/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/tienda/rest/clientes/1??
```

Invocación del servicio desde IntelliJ

Abrimos la utilidad 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.

1.5. Ejercicios

Debido a la extensión de las clases de teoría y al poco tiempo que tuvimos para hacer ejercicios en clase, no hay que entregar los ejercicios de la sesión 4 y se modifica la puntuación de los ejercicios de las sesiones 1, 2 y 3

Antes de empezar a crear los proyectos, debes descargarte el repositorio *git* `java_ua/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>/rest-expertojava
```

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

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

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

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

Servicio REST saludo (1,5 puntos)

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

- Creamos un módulo Maven con IntelliJ (desde el directorio `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 "Test RESTFul Web Service" de IntelliJ. Comprobar que la invocación:

```
GET http://localhost:8080/saludo-rest/recursos/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 /recursos/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. Comprobar que la invocación:

```
GET http://localhost:8080/saludo-rest/recursos/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, y posteriores invocaciones a los métodos GET utilizarán el nuevo saludo. La variable que contenga el nuevo saludo debe ser estática. ¿Qué pasa si no lo es? Una vez desplegada la aplicación en el servidor WildFly, prueba el servicio mediante la utilidad "Test RESTFul Web Service" de IntelliJ. Realizar las siguientes invocaciones (en este orden):

```
PUT http://localhost:8080/saludo-rest/recursos/holamundo/Buenos%20dias
GET http://localhost:8080/saludo-rest/recursos/holamundo
GET http://localhost:8080/saludo-rest/recursos/holamundo/pepe
```

La segunda 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 (1,5 puntos)

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 los tres primeros pasos del 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

> 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.

- 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 con 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, por ejemplo un cliente WebDAV que hacen uso de la operación HTTP LOCK, puedan ser "atendidos" por nuestro servicio REST.

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.

> La anotación @Path no es necesario que contenga una ruta que empiece o termine en 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, relativos a 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**

La anotación @Path puede incluir variables 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 **getCliente()** 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+}")
  public String getCustomer(@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 par 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:

```
"([]*)"
```

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 *"/address"*. 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`.

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

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 Jpeg 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 y (*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, Customer>();
  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 StreamingOutput recuperarClienteId(@PathParam("id") int id)
 { ... }
```

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

Si un usuario envía la petición *GET /clientes/norteamerica-db/333*, el proveedor JAX-RS primero intentará "emparejar" del 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 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*.

Carácter dinámico del "dispatching" de peticiones

En el ejemplo anterior hemos ilustrado el concepto de *subresource locator*, aunque no hemos mostrado completamente su carácter dinámico. 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 StreamingOutput 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,
                             InputStream is) {
      ...
    }
}
```

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(InputStream representation) { }
}
```

Si @Consumes se aplica a la clase, por defecto los métodos de respuesta 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 *input stream* cuyo tipo MIME es "application/xml", y que se almacena en la variable *representation*. 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`.

Si no hay métodos de recurso que puedan responder al tipo MIME solicitado, 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")**. Como por ejemplo sucede en el código siguiente:

```
@POST
@Consumes("application/xml")
public void creaPedido(InputStream 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.

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`.

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 StreamingOutput recuperarClienteId(@PathParam("id") int id) {
    ...
  }
}
```

Podemos referenciar más de un parámetro de 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 StreamingOutput 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 String getPath();
  public List<PathSegment> getPathSegments();
  public MultivaluedMap<String, String> getPathParameters();
  ...
}
```

El método `getPath()` permite obtener la ruta relativa de nuestros servicios REST utilizada para realizar el *matching* con nuestra petición de entrada. El método `getPathSegments()` "divide" la ruta relativa de nuestro servicio REST en una serie de objetos *PathSegment* (segmentos de ruta, delimitados por `/`). El método `getPathParameters()` devuelve un objeto de tipo *MultivaluedMap* con todos los parámetros de path definido en todas las expresiones @Path.

Podemos obtener 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 Jpeg getImagen(@Context UriInfo info) {
          String fabricado = info.getPathParameters().getFirst("marca");
          PathSegment modelo = info.getPathSegments().get(2); DUDA ??????
          ...
  }
}
```

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.

@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 Jpeg 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 *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>
```

Podríamos, a partir de este cuerpo de mensaje, inyectar 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.

@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*.

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 especificada, 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, 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, para implementaremos 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 "pre-asignados". 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. Cuando estos objetos sean registrados, JAX-RS inyectará los valores correspondientes a los atributos y métodos *setter* anotados con @Context.

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, ni tampoco indicamos nada para *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 denominada *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:

```
<?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).

2.6. Ejercicios

Debido a la extensión de las clases de teoría y al poco tiempo que tuvimos para hacer ejercicios en clase, no hay que entregar los ejercicios de la sesión 4 y se modifica la puntuación de los ejercicios de las sesiones 1, 2 y 3

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 `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 (1 punto)

Vamos a crear un recurso JAX-RS al que denominaremos `TemasResource`. En el siguiente ejercicio , al configurar la aplicación, haremos que el recurso sea un **singleton**. Vamos a hacer ahora que nuestro recurso gestione 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

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

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

y sus correspondientes *getters* y *setters*:

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

- Nuestro recurso estará accesible en el servidor en la ruta `/temas` (relativa a la raíz del contexto de nuestra aplicación.
- Implementamos un primer método para poder crear un nuevo tema en el foro denominado `creaTema()`. Un tema estará representado por una cadena de caracteres, pero con la restricción de que solamente puede contener letras mayúsculas o minúsculas y como mínimo debe contener dos caracteres. Para ello tendrás que utilizar una expresión regular en la anotación correspondiente. Ejemplos de URIs válidas e inválidas son:

  # URIs válidas: /temas/animales, /temas/plantas, /temas/ab

    como resultado se crearán los temas "animales", "plantas" y "ab", respectivamente

  # URIs inválidas: /temas/a, /temas/problemas1, /temas/123

- Implementamos un segundo método para consultar los temas creados en el foro. El método se denominará `verTemasTodos()`, y devuelve (en formato texto) todos los mensajes 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 mensajes almacenados, entonces devolveremos el mensaje: "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 9:
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"
```

Las URIs indicadas en este ejercicio son relativas a la raíz del contexto de nuestra aplicación. 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". Por lo tanto la URI completa para, por ejemplo, el último apartado sería: http://localhost:8080/s2-foro-nuevo//temas?inicio=7&total=1

Despliegue y pruebas del recurso (1 punto)

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*. 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 el cliente que proporciona IntelliJ, utilizando las URIs indicadas en el ejercicio anterior, de forma que comprobemos que se crean correctamente los temas "animales", "plantas", y "ab", y que no se admiten como nombres de temas "a", "problemas1", o "123". Comprueba también 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 (1 punto)

Implementa dos 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). Por ejemplo:

  # /temas/2

  Debe devolver lo siguiente:

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

/temas/4

Debe devolver lo siguiente:

```
"Ver el tema 4:"
```

/temas/2we

Es una URI incorrecta. 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:"

temas/af

Debe devolver un cuerpo de mensaje vacío y la cabecera de respuesta: **HTTP/1.1 404 Not Found**

Creación de subrecursos (1 punto)

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, devolveremos el valor "null". Por ejemplo:
  # Deberemos poder realizar una petición **POST** a **/temas/1/mensajes**, con el cuerpo de mensaje = "Mensaje numero 1". Fíjate que en este caso el cuerpo del mensaje ya no puede extraerse de la URI como ocurre en los apartados anteriores. 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

Una petición **POST** a la URI: /temas/1/mensajes/pepe, con el cuerpo de mensaje con valor "mensaje de pepe" debería crear el mensaje, 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)

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. Api cliente.

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.

Asímismo, ahora hemos hablado sobre la creación de servicios web RESTful y hemos "probado" nuestros servicios utilizando el cliente que nos proporciona IntelliJ 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.

3.1. Proveedores de entidades

El *runtime* de JAX-RS puede "extenderse" utilizando clases "proveedoras" 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. Concretamente, las interfaces `MessageBodyWriter` y `MessageBodyReader` permiten crear clases denominadas **proveedoras de entidades** (**entity providers**). Una clase *entity provider* proporciona un servicio de mapeado entre representaciones (tipos *mime*) y sus tipos Java asociados. Por ejemplo en el cuerpo del mensaje HTTP de entrada podemos proporcionar los datos de un cliente en formato XML, que necesitaremos mapear a un objeto Java *Cliente* para ser procesado por nuestro servicio RESTful. La clase que realiza esta transformación desde XML a objetos Java es una clase *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.

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*.

A continuación expondremos 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. Todos ellos son una alternativa al uso de `MessageBodyWriter` y/o `MessageBodyReader`.

javax.ws.rs.core.StreamingOutput

Ya hemos introducido la interfaz StreamingOutput en la primera sesión. 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("filepath") 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 tipo especificado en `Content-Type`. 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 que 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) {
        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.

3.3. Introducción a JAXB

JAXB (**J**ava **A**rchitecture for **X**ML **B**inding) es una especificación Java antigua (JSR 222[1]) 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 nombreCompleto;

    public Customer() {}

    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 (etiquetas XML). 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 atributo `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 `@XMLElement` 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.

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

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). Esta anotación tiene como valor `XmlAccessType.FIELD`, lo que significa que por defecto se deben serializar **todos** los campos de la clase (estén anotados o no). 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.

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">
     <nombre>Pablo Martinez</nombre>
</cliente>
```

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 {

    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 = "estado")
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 definidos 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() {};

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

Si accedemos a este servicio, nos devolverá la información sobre el estado de la siguiente forma:

```
<tarea>
   <estado>Idle</estado>
   <tonerRestante>25</tonerRestante>
   <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>
```

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

```
@XmlRootElement(name="estado")
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:

```
<estado 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>
</estado>
```

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

> En caso de que las propiedades no sean públicas, podemos etiquetar los *getters* correspondiente a ellas.

Hemos visto que en 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="estado")
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:

```
<estado 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>
</estado>
```

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:

```
<estado 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>
</estado>
```

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 clase etiquetándola con `@XmlAccessorType`. Por ejemplo:

```
@XmlAccessorType(NONE)
```

```
@XmlRootElement(name="estado")
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, necesitamos 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`. Vamos a mostrar un ejemplo utilizando la clase `Cliente` que hemos definido con anterioridad:

```
@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</descriction>
   <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"
               }
}
```

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, el formato de respuesta por defecto XML, pero se elegirá el formato JSON 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 la siguiente sesión.

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[2] para más información.

3.4. Respuestas del servidor y manejo de excepciones

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.

[2] http://www.vogella.de/articles/JAXB/article.html

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.

Finalmente, comentaremos cómo podemos tratar las excepciones en nuestros servicios RESTful.

Códigos de respuesta por defecto

Los códigos de respuesta por defecto se corresponden con el comportamiento indicado en la especificación[3] 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) {...}

  @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

[3] http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html

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 `clientes`, entonces el servidor no encontrará ningún método del recurso que pueda servir dicha petición. En este caso, al cliente se le enviará 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 al cliente 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 nuestro cliente realiza la siguiente petición de entrada:

```
GET /customers
```

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 con POST se crea un nuevo recurso 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 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 URU getLocation();  ❹
   public abstract MediaType getMediaType();  ❺
   ...
}
```

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

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 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`:

```
public static 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 `Response` pre-inicializada con un código de estado de `200 OK`. El resto de métodos funcionan de forma similar.

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);
```

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
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");
     return builder.cookie(cookie).build();
}
```

❶ Creamos una nueva *cookie* con el nombre de clave `nombre` y le asignamos el valor `pepe`

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"),
  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 *100" 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.

Manejadores de excepciones

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. Esta excepción se puede pre-inicializar con un objeto `Response`, o con un código de estado particular:

```
public class WebApplicationException extends RuntimeException {
  public WebApplicationException() {...}
```

```
  public WebApplicationException(Response response) {...}
  public WebApplicationException(int status) {...}
  public WebApplicationException(Response.Status status) {...}
  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() {...]
}
```

Cuando JAX-RS detecta que se ha lanzado la excepción `WebApplicationException`, la captura y realiza una llamada al método `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. Por ejemplo, podemos cambiar el ejemplo 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`

Table 3. 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
NotSupportedException	415	El cliente ha incluido un *Media type* no soportado
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` 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` 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` se usa cuando queremos comunicar al cliente que el recurso que está solicitando no existe. Esta excepción también se generará de formam 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` 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` 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` 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` 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` 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.

3.5. 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 veremos en los ejemplos, que lo utilizaremos "encadenando" una sucesión de llamadas a métodos del API, permitiéndonos así escribir menos líneas de código.

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();

Cliente cliente = target.queryParam("nombre", "Alvaro Gomez")
                        .request()
                        .get(Cliente.class);  ❺
client.close();
...
```

❶ Obtenemos una instancia de `Client`
❷ Creamos un `WebTarget`
❸ Creamos la petición
❹ Realizamos una invocación POST
❺ 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`:

```
package javax.ws.rs.client;
  import java.net.URL;
  import java.security.KeyStore; import javax.ws.rs.core.Configurable;
  import javax.ws.rs.core.Configuration; import
 javax.net.ssl.HostnameVerifier;
  import javax.net.ssl.SSLContext;

  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` list para ser usada. Si queremos utilizar alguna configuración particular para la construcción de la instancia `Client`, el método `newBuilder()` crea una instancia de tipo `ClientBuilder` que permite registrar componentes y determinar propiedades de configuración (utilizando los métodos de la interfaz `Configurable`):

```
package javax.ws.rs.core;
  public interface Configurable<C extends Configurable> {
    public C property(String name, Object value);

    public C register(Class<?> componentClass);
    public C register(Object component);
    ...
}
```

Veamos un ejemplo de uso de `ClientBuilder.newBuilder()` :

```
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()` 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 *pesadas*. Se deberían reutilizar las instancias de esta interfaz en la medida de lo posible, 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

> 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".

Al igual que `ClientBuider`, la interfaz `Client` implementa `Configurable`. Esto nos permitirá cambiar la configuración y registrar componentes en la parte del cliente en tiempo de ejecución. Sin embargo, el principal propósito de `Client` es crear instancias de `WebTarget`:

```
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);
    ...
}
```

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.

```
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:

```
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
saludo.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.

Realizamos la petición

Una vez que hemos creado y configurado convenientemente el `WebTarget`, que representa la URI que queremos invocar, tenemos que realizar la petición a través de uno de los métodos `WebTarget.request()`:

```
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`. El método `WebTarget.request()` devuelve una instancia de `Invocation.Builder`, una interfaz que proporciona métodos para preparar la respuesta del cliente. Contiene un conjunto de métodos que nos permiten construir diferentes tipos de cabeceras de peticiones.

```
Client cliente = ClientBuilder.newClient();
WebTarget miRecurso = client.target("http://ejemplo/webapi/mensaje");
Invocation.Builder builder = miRecurso.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");
```

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 que el recurso REST, al que va dirigido la petición, espera. Estos métodos son:

- `get()`
- `post()`
- `delete()`
- `put()`
- `head()`
- `options()`

Por ejemplo, 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);
```

Si el recurso REST destinatario de la petición espera una petición de tipo HTTP POST, invocaremos el método `Invocation.Builder.post()`.

```
Client cliente = ClientBuilder.newClient();
PedidoAlmacen pedido = new PedidoAlmacen(...);
WebTarget miRecurso = client.target("http://ejemplo/webapi/escritura");
NumeroSeguimiento numSeg =
                   miRecurso.request(MediaType.APPLICATION_XML)
                            .post(Entity.xml(pedido),
 NumeroSeguimiento.class);
```

En el ejemplo anterior, el tipo de retorno es una clase particular (`NumeroSeguimiento`), y para recuperarla es necesario incluir el tipo de la clase como un parámetro en el método `InvocationBuilder.post(Entity<?> entity, Class<T> responseType)`. El cuerpo del mensaje se crea con la llamada `Entity.xml(pedido)`. La clase `Entity` proporciona métodos estáticos para diferentes tipos de *media types*. El método `xml()` tiene como parámetro un objeto Java, y asigna el valor `application/xml` a la cabecera HTTP `Content-Type`.

Si el tipo de retorno 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>>()
 {});
```

También podemos enviar parámetros de formularios en nuestras peticiones PUT y/o POST. Veamos un ejemplo:

```
Form form = new Form().param("nombre", "Pedro")
                      .param("apellido", "Garcia");
...
response = client.target("http://ejemplo/clientes")
                 .request().
                .post(Entity.form(form));
response.close();
```

Manejo de excepciones

Veamos que 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.

3.6. Ejercicios

Debido a la extensión de las clases de teoría y al poco tiempo que tuvimos para hacer ejercicios en clase, no hay que entregar los ejercicios de la sesión 4 y se modifica la puntuación de los ejercicios de las sesiones 1, 2 y 3

Para esta sesión proporcionamos un proyecto como plantilla con el nombre `s3-tienda` que tendrás utilizar como punto de partida para aplicar lo que hemos aprendido en esta sesión.

Se trata de una implementación parcial para gestionar un sistema de ventas on-line.

La estructura lógica del proyecto proporcionado es la siguiente:

- Paquete `org.expertojava.negocio`: es la capa de negocio de nuestra aplicación. Por simplicidad, no usamos una base de datos real, sino que trabajamos con datos en memoria.
  - # sub paquete `modelo`: contiene la defínción de las clases que utiliza la capa de negocio para interactuar con las funcionalidades de la tienda. elemento Cliente
  - # sub paquete `api`: contiene la implementación de los servicios ofertados por nuestra tienda virtual
- Paquete `org.expertojava.rest`: constituye la capa rest de nuestra aplicación. Esta capa es cliente de la capa de negocio
  - # sub paquete `modelo`: contiene la definición de las clases utilizadas por los servicios rest
  - # sub paquete `api`: contiene la implementación de los recursos rest de nuestra tienda.

Vamos a trabajar con JAXB y API cliente de JAX-RS, y también con JUnit para probar nuestra implementación, por lo que hemos incluido en el pom.xml las dependencias correspondientes:

```
...
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>3.0.5.Final</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jaxb-provider</artifactId>
  <version>2.3.3.Final</version>
</dependency>
...
```

Además, hemos "alterado" el orden de ejecución de los test unitarios (por defecto, Maven ejecuta dichos tests "antes" de generar el .war), para poder desplegar nuestro .war en Wilfly, y

a continuación ejecutar los tests Junit. El plugin que se encarga de ejecutar los tests unitarios en Maven es el plugin "surefire":

```
<!-- 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>${wildfly-hostname}</hostname>
    <port>${wildfly-port}</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 install-->
<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>
```

Con estos cambios en el pom.xml, y ejecutando el comando `mvn install` se llevarán a cabo las siguientes acciones, en este orden:

- Después de compilar el proyecto, obtenemos el `.war`
- El `.war` generado se despliega en el servidor de aplicaciones Wilfly

- Se ejecutan los test JUnit directamente en el servidor de aplicaciones
- Se guarda una copia del `.war` en nuestro repositorio local (`$HOME/.m2`)

El proyecto tiene implementado de momento dos recursos:

- `ClienteResource`: acepta peticiones POST con un cuerpo del mensaje en formato XML, con los datos de un cliente. Dichos datos son leidos y convertidos a la entidad java correspondiente a través de un `InputStream` utilizando el método `leercliente()`. También acepta peticiones GET que devuelven los datos de un cliente en formato XML. Igualmente, hemos implementado manualmente la conversión de la entidad Java a formato XML y devolvemos un `StreamingOutput`.
- `ProductoResource`: acepta peticiones POST, y los datos del producto se pasan como *queryParams*. También acepta peticiones GET, de la misma forma que para el recurso `ClienteResource`

Para familiarizarte con el código, puedes realizar alguna prueba con el cliente rest de IntelliJ. Para ello, hemos dejado en el directorio `/src/main/resources` hemos dejado un fichero `cliente.xml` que puedes utilizar para crear nuevos clientes.

También puedes probar la aplicación utilizando la herramienta `curl` desde línea de comandos. Por ejemplo, para probar el recurso `ProductosResource` añadimos dos nuevos productos:

```
curl -i -H "Accept: */*" -X POST  "http://localhost:8080/s3-tienda/rest/
productos?nombre=camara%20sony&precio=49.56"

curl -i -H "Accept: */*" -X POST  "http://localhost:8080/s3-tienda/rest/
productos?nombre=ipad%64Gb&precio=600.99"
```

Y consultamos los productos que hemos añadido:

```
curl -i -H "Accept: */*" -X GET  "http://localhost:8080/s3-tienda/rest/
productos/1"

curl -i -H "Accept: */*" -X GET  "http://localhost:8080/s3-tienda/rest/
productos/2"
```

Uso de manejadores de contenidos y JAXB (1 punto)

Modifica convenientemente el código de nuestros recursos REST para utilizar JAXB, de forma que el serializado (y deserializado) de las entidades java de los recursos a XML, se realice de forma automática por dicha librería.

En el caso de recibir peticiones POST sobre los **clientes**, además, debemos devolver en el objeto `Response` la URI del nuevo recurso cliente creado (por ejemplo: http://localhost:8080/rest/clientes/5, suponiendo que se ha creado el cliente número 5)

En el caso de recibir peticiones POST sobre los **productos**, además, en lugar de recibir como parámetro el cuerpo del mensaje (objeto `ProductBean`), recibirá los datos de entrada desde un formulario html. Se proporciona el fichero `index.html`, con dicho formulario, que puedes utilizar para añadir diferentes productos.

Recuerda que primero tenemos que **poner en marcha** el servidor Wilfdly. Y a continuación tenemos que ejecutar la fase `install` desde la ventana de Maven (s3-tienda# Lifecycle#install), para desplejar en Wildfly el .war generado y posteriormente ejecutar los tests JUnit en el servidor.

Uso del API cliente (1 punto)

Utilizar el API cliente para modificar los métodos `inicializarClientes()` e `inicializarPedidos()` del fichero de test (TiendaTest.java) para probar los cambios introducidos en el apartado anterior.

Añade dos nuevos tests, para probar la recuperación de clientes y productos.

Manejo de excepciones (1 punto)

Modifica los métodos GET para que devuelvan una excepción propia de nuestra aplicación denominada `ClienteNoEncontradoException` y `ProductoNoEncontradoException` de tipo *unchecked*, cuando el cliente consultado no existe (en el primer caso), o el producto no existe (en el segundo).

Crea un *mapper* para capturar dicha excepción, de forma que se devuelva el estado 404, "Not found", en ambos casos.

Implementa dos nuevos tests para probar la consulta de clientes o productos que no existen y comprobar que se lanzan correctamente las excepciones.

4. Procesamiento JSON. HATEOAS. Escalabilidad y Seguridad

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.

Ademas, 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.

Por otro lado, cuando hablamos de servicios REST, las respuestas de los mismos deben definirse, de forma implícita o explícita, como "caheables" o no, para así poder evitar que los clientes reutilicen información obsoleta o datos inadecuados sucesivas peticiones. Una buena gestión de la caché, elimina parcial o totalmente la necesidad de nuevas interacciones entre el cliente y el servidor, mejorando así la escalabilidad y rendimiento de los servicios.

4.1. 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**[4] 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*:

```
{ "nombre": "John",
  "apellidos": "Smith",
  "edad": 25,
  "direccion": { "calle": "21 2nd Street",
                 "ciudad": "New York",
                 "codPostal": "10021"
               },
  "telefonos": [
         { "tipo": "fijo",
           "numero": "212 555-1234"
         },
```

[4] https://jcp.org/aboutJava/communityprocess/final/jsr353/index.html

```
        {
          "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** 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:

```
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.

4.2. Modelo de procesamiento basado en el modelo de objetos

En este caso se crea un árbol en memora 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 mças 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 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 cómo crear un modelo de objetos mediante 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() 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()`.

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)
         navigateTree(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 los dos modelos creados en los dos ejemplos de código anteriores 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, el la 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* * Para escribir datos JSON utilizando un *generador*

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: . Obtener una instancia de un parser invocando el método estático `Json.createParser()` . Iterar sobre los eventos del parser utilizando los métodos `JsonParser.hasNext()` y `JsonParser.next()` . 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 *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", "mobile")
         .write("numero", "111-111-1111")
      .writeEnd()
      .writeStartObject()
         .write("tipo", "home")
         .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.

4.3. ¿Qué es HATEOAS?

Comúnmente se hacer 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 "buscatos". 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.

4.4. 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**[5] 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> indica la forma de obtener los siguientes clientes de la lista.

Un enlace Atom es simplemente un elemento XML 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`

[5] http://www.w3.org/2005/Atom

- 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, 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 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 índice 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>
</order>
```

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 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`. 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 Atomm com el mimso nombre. En el ejemplo no se muestra, pero podríamos especificar el *media type* utilizando el atributo `type`.

4.5. 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 rutas variables).

```
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 con 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:

```
public abstract UriBuilder clone();
public abstract UriBuilder uri(URI uri) throws IllegalArgumentException;
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);
#...
```

Los métodos `build()` crean la URI. Esta 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.

```
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}/customers/{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/customers/333?param=value
```

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, "getCustomer");
```

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 plantillas. Para ello disponemos de métodos `resolveTemplateXXX()`, que nos facilitan el trabajo:

```
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);
```

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 plantilla en forma de `String`:

```
String original = "http://{host}/{id}";
String nuevaPlantilla = UriBuilder.fromUri(original)
                        .resolveTemplate("host", "localhost")
                        .toTemplate();
```

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`.

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

4.6. *Caching*

La posibilidad de almacenar los datos en memoria caché es una de las características más importantes de la Web. Cuando visitamos un sitio web por primera vez, nuestro navegador almacena imágenes y texto estático en memoria y en el disco duro. Si volvemos a visitar ese sitio en unos minutos, horas, días, o incluso meses, nuestro navegador no tiene que volver a cargar los datos desde la red, y puede acceder a ellos de forma local. Esto acelera notablemente el renderizado de las páginas que ya hemos visitado y hace que nuestra "navegación" sea mucho más fluida. Este almacenamiento en caché de los datos no sólo sirve ayuda a la visualización de las páginas, sino que también reduce la carga del servidor. Si el navegador obtiene las imágenes o el texto de forma local, no está consumiendo ancho de banda del servidor o ciclos de CPU.

También son importantes los *proxies* de memoria caché (*proxy caches*). Estos *proxies* son "pseudo" servidores web que sirven de intermediarios entre los navegadores y los sitios web. Su único propósito es aminorar la carga de los servidores "cacheando" contenido estático y sirviéndolo a los clientes directamente. Las redes de entregas de contenidos (**C**ontent **D**elivery **N**etworks) invierten millones de dólares en este concepto. Las CDNs nos proporcionan una red de *proxies* de memoria caché, que podemos utilizar para publicar sitios web y así escalarlos a cientos de miles de usuarios.

Si nuestros servicios Web son RESTful, podemos aprovechar la posibilidad de "cacheo" que nos proporciona la Web e incluirla en nuestras aplicaciones. Si hemos seguido fielmente las restricciones de la interfaz HTTP, cualquier URI de un servicio que pueda ser accedido con una petición HTTP GET es un cadidato a ser "cachaeado", ya que es, por definición, de sólo lectura e idempotente.

¿Cuándo es conveniente almacenar en caché los datos? Cualquier servicio que proporcione contenido estático que no cambie es un candidato obvio. Aunque tengamos datos dinámicos

que son accedidos de forma concurrente, también podemos considerar el realizar *caching*, incluso aunque los datos sólo sean válidos durante unos pocos segundos o minutos.

Primero tenemos que entender cómo funciona el *caching* de datos en la Web. El protocolo HTTP permite controlar la forma en la que se "cachean" los datos a través de varias cabeceras de petición y respuesta. Vamos a comentar algunas de ellas:

Cabecera Expires

Esta cabecera nos permite establecer y conocer cuando cachear los datos. Es una cabecera de respuesta definida en HTTP 1.0, que indica al navegador que éste puede almacenar los datos en caché y por cuánto tiempo. El valor de esta cabecera es una fecha del futuro en la que los datos ya no serán válidos. Cuando se "alcanza" este fecha, el cliente no debería usar los datos en caché y debería volver a recuperar los datos del servidor. Por ejemplo, si un cliente envía una petición `GET /clientes/123`, un ejemlo de respuesta utilizando la cabecera `Expires` podría ser ésta:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Expires: Tue, 15 May 2014 16:00 GMT

<cliente id="123">...</cliente>
```

Este dato puede ser almacenado en caché, y será válido hasta el jueves, 15 de mayo de 2014.

Podemos implementar esta respuesta en JAX-RS utilizando un objeto `javax.ws.rs.core.Response`. Por ejemplo:

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

  @Path("{id}")
  @GET
  @Produces("application/xml")
  public Response getCliente(@PathParam("id") int id) {
    Customer cli = buscarCliente(id);
    ResponseBuilder builder = Response.ok(cli, "application/xml");
    Date fecha = Calendar.getInstance(TimeZone.getTimeZone("GMT"))
                    .set(2010, 5, 15, 16, 0);
    builder.expires(date); ❶
    return builder.build();
  }
}
```

❶ El método `ResponseBuilder.expires()` asigna a la cabecera `Expires` la fecha límite de validez de los datos de respuesta.

Cabecera Cache-Control

La especificación de HTTP 1.1 incluye un mayor control sobre el almacenamiento en caché, definiendo la cabecera `Cache-Control`. Ésta permite utilizar un conjunto de directivas para definir quién puede realizar el *caching* de los datos, cómo, y por cuánto tiempo. Dichas directivas son las siguientes:

private
Indica que no se permite que ningún intermediario (*proxy* o CDN) *cachee* la respuesta. Esta es una forma de asegurar que el cliente, y sólo el cliente, puede *cachear* los datos.

public
Indica lo contrario que la anterior: que la respuesta puede *cachearse* por cualquier entidad dentro de la cadena de petición/respuesta.

no-cache
Indica que la respuesta no debe cachearse. Si se cachea, no debe utilizarse sin realizar un proceso de "revalidación" con el servidor (hablaremos de esto un poco más adelante)

no-store
Ya hemos indicado que un navegador puede almacenar las respuestas en memoria o en disco. Con esta directiva estamos diciéndole al navegador o al *proxy* que no almacene los datos en disco.

no-transform
En ocasiones, las entidades que realizan *caching* de los datos tienen la opción de transformarlos para reducir el espacio en memoria o el disco, o simplemente para reducir el tráfico de la red (un ejemplo es la compresión de imágenes). Para algunas aplicaciones, podríamos querer deshabilitar estas acciones con la directiva `no-transform`.

max-age
Indica durante cuánto tiempo (en segundos) es válido el dato en caché. Si se especifica también la cabecera `Expires` en la misma respuesta, la directiva `max-age` tiene precedencia.

s-maxage
Es equivalente a la directiva anterior, pero especifica el máximo tiempo de validez del dato "cacheado" por una endidad intermediaria (como un *proxy*). Este valor podría ser diferente del tiempo de validez especificado para el cliente.

Veamos un ejemplo de respuesta utilizando la cabecera `Cache-Control`:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: private, no-store, max-age=300

<clientes>...</clientes>
```

En este ejemplo, la respuesta indica que solamente el cliente puede almacenar la respuesta en caché. Esta respuesta es válida durante 300 segundos y no debe ser almacenada en el disco duro.

La especificación JAX-RS proporciona `javax.ws.rs.core.CacheControl`, que es una clase que representa la cabecera `Cache-Control`:

```
public class CacheControl {
  public CacheControl() {...}
  public static CacheControl valueOf(String value)
                              throws IllegalArgumentException {...}
  public boolean isMustRevalidate() {...}
  ...
  public boolean isProxyRevalidate() {...}
  ...
  public int getMaxAge() {...}
```

```
  public void setMaxAge(int maxAge) {...}
  public int getSMaxAge() {...}
  public void setSMaxAge(int sMaxAge) {...}
  ...
  public void setNoCache(boolean noCache) {...}
  public boolean isNoCache() {...}
  public boolean isPrivate() {...}
  public List<String> getPrivateFields() {...}
  public void setPrivate(boolean private) {...}
  public boolean isNoTransform() {...}
  public void setNoTransform(boolean noTransform) {...}
  public boolean isNoStore() {...}
  public void setNoStore(boolean noStore) {...}
  ...
}
```

Veamos un ejemplo del uso de esta clase:

```
@Path("/clientes")
public class ClienteRecurso {
@Path("{id}")
@GET
@Produces("application/xml")
public Response getCliente(@PathParam("id") int id) {
    Cliente cli = buscarCliente(id);
    CacheControl cc = new CacheControl(); ❶
    cc.setMaxAge(300);
    cc.setPrivate(true);
    cc.setNoStore(true);
    ResponseBuilder builder = Response.ok(cli, "application/xml");
    builder.cacheControl(cc); ❷
    return builder.build();
}
```

❶ Inicializamos un objeto `CacheControl`

❷ Pasamos la instancia de `CacheControl` al método `ResponseBuilder.cacheControl()` para asignar el valor de la cabecera de respuesta `Cache-Control`

Revalidation y *GETS* condicionales

Un aspecto intersante del protocolo de *caching* es que cuando la caché "ha vencido", la entidad que realiza el caché de los datos, puede preguntar al servidor si el dato que ha almacenado todavía sigue siendo válido. Este proceso se denomina **revalidación**. Para poder realizar la *revalidación*, el cliente necesita alguna información adicional del servidor que proporciona el recurso que vamos a almacenar en caché. El servidor enviará de vuelta al cliente una cabecera `Last-Modified` y/o una cabecera `ETag` con su respuesta inicial.

Cabecera Last-Modified

Esta cabecera representa la fecha del dato enviado por el servidor. Mostramos un ejemplo:

```
HTTP/1.1 200 OK
```

```
Content-Type: application/xml
Cache-Control: max-age=1000
Last-Modified: Tue, 15 May 2013 09:56 EST

<cliente id="123">...</cliente>
```

La respuesta inicial del servidor nos dice que el XML de la respuesta es válido durante 1000 segundos y fué enviado el 15 de mayo de 2013. Si el cliente soporta revalidación, almacenará esta fecha junto con la respuesta. Después de 1000 segundos, el cliente puede optar por revalidar los datos almacenados. Para ello, realizará una petición GET **condicional** pasando la cabecera `If-Modified-Since` con el valor de la cabecera `Last-Modified` recibida.

Por ejemplo:

```
GET /customers/123 HTTP/1.1
If-Modified-Since: Tue, 15 May 2013 09:56 EST
```

Cuando un servicio recibe esta petición GET, comprueba si el recurso ha sido modificado desde la fecha indicada en la cabecera `If-Modified-Since`. Si se ha cambiado desde esa fecha, el servidor enviará una respuesta "200 OK", con la nueva representación del recurso. Si no ha cambiado, el servidor responderá con "304 Not Modified", y no devolverá ninguna representación. En ambos casos, el servidor debería enviar las cabeceras `Cache-Control` y `Last-Modified` actualizadas, si procede.

Cabecera ETag

La cabecera `ETag` es un identificador *pseudo único* que representa la versión del dato enviado de vuelta. Es una cadena de caracteres arbitraria y normalmente suele ser un valor hash MD5 (MD5 es un algoritmo criptográfico muy utilizado para proporcionar la seguridad de que un archivo descargado de Internet no se ha alterado). Vamos a mostrar un ejemplo de respuesta con esta cabecera:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Cache-Control: max-age=1000
ETag: "314127134255432234 3200"

<cliente id="123">...</cliente>
```

Al igual que la cabecera `Last-Modified`, cuando el cliente cachea la respuesta, debería también almacenar en caché el valor `ETag`. Cuando la caché expira después de 1000 segundos, el cliente realiza una petición de revalidación con la cabecera `If-None-Match` que contiene el valor de ETag que se había cacheado. Por ejemplo:

```
GET /clientes/123 HTTP/1.1
If-None-Match: "314127134255432234 3200"
```

Cuando un servicio recibe esta petición GET, intenta emparejar el hash `ETag` actual del recurso con el proporcionado por la cabecera `if-None-Match`. Si las etiquetas no coinciden, el servidor enviará de vuelta una respuesta "200 OK", con la nueva representación del recurso.

Si éste no ha cambiado, el servicor responderá con "304 Not Modified", y no devolverá ninguna representación. En ambos casos, el servidor debería enviar las cabeceras `Cache-Control` y `ETag` actualizadas, si procede.

JAX-RS proporciona la clase `javax.ws.rs.core.EntityTag` que representa la cabecera `ETag`:

```
public class EntityTag {
  public EntityTag(String value) {...}
  public EntityTag(String value, boolean weak) {...}
  public static EntityTag valueOf(String value)
                      throws IllegalArgumentException {...}
  public String getValue() {...}
}
```

JAX-RS y GETs condicionales

Para poder realizar invocaciones GET condicionales, JAX-RS proporciona la clase inyectable `javax.ws.res.core.Request`:

```
public interface Request { ...
  ResponseBuilder evaluatePreconditions(EntityTag eTag);
  ResponseBuilder evaluatePreconditions(Date lastModified);
  ResponseBuilder evaluatePreconditions(Date lastModified, EntityTag
 eTag);
}
```

Los métodos `evaluatePreconditions()` aceptan un parámetro de tipo `javax.ws.rs.core.EntityTag`, un `java.util.Date`, que representa la última fecha de modificación, o ambos. Estos valores serán comparados con los valores de las cabeceras `If-Mofified-Since`, `If-unmodified-Since`, o `If-None-Match` enviadas con la petición. Si estas cabeceras no existen o si los valores de las cabeceras de la petición no pasan la revalidación, estos métodos devuelven `null` y deberíamos enviar una respuesta "200 OK" con la nueva representación del recurso. Si el método no devuelve `null`, entonces devuelve una instancia preinicializada de `ResponseBuilder` con el código de respuesta prefijada a 304. Por ejemplo:

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

  @Path("{id}")
  @GET
  @Produces("application/xml")
  public Response getCliente(@PathParam("id") int id,     ❶
                             @Context Request request) {    ❷
    Cliente cli = buscarCliente(id);
    EntityTag tag = new EntityTag(Integer.toString(cli.hashCode())); ❸
    CacheControl cc = new CacheControl();
    cc.setMaxAge(1000);
    ResponseBuilder builder = request.evaluatePreconditions(tag); ❹
    if (builder != null) {   ❺
```

```
      builder.cacheControl(cc);
      return builder.build();
    }

    // No se satisfacen las precondiciones!  ❻
    builder = Response.ok(cli, "application/xml");
    builder.cacheControl(cc);
    builder.tag(tag);
    return builder.build();
  }
}
```

❶ El método `getCliente()` procesa peticiones para las URI que satisfacen la plantilla `/clientes/{id}`.
❷ Inyectamos una instancia de `javax.ws.rs.core.Request` en el método utilizando la anotación `@Context`.
❸ Creamos el `ETag` actual para la instancia de `Cliente` a partir de código hash del objeto
❹ Realizamos la revalidación invocando al método `Request.evaluatePreconditions`
❺ Si las etiquetas coinciden, reseteamos la expiración de la caché del cliente enviando una nueva cabecera `Cache-Control`
❻ Si las etiquetas no coinciden, construimos una `Response` con un nuevo valor actual de `ETag` y `Cliente`

4.7. Filtros e interceptores

Tanto los filtros como los interceptores son objetos que se "interponen" entre el procesamiento de las peticiones, tanto del servidor como del cliente. Permiten encapsular comportamientos comunes que trascienden a buena parte de nuestra aplicación. Este comportamiento está relacionado con código referente a la infraestructura o acciones de protocolo que no queremos "entremezclar" con la lógica de negocio. Aunque la mayoría de características del API de JAX-RS se usan por los desarrolladores de aplicaciones, los filtros e interceptores suelen utilizarse más por los desarrolladores de *middelware*.

Filtros en el servidor

En la parte del servidor, podemos hablar de dos tipos de filtros: filtros de **petición** y filtros de **respuesta**. Los filtros de petición se ejecutan antes de que se invoque a un método JAX-RS. Los filtros de respuesta se ejecutan después de que el método haya terminado. Por defecto se ejecutan para todas las peticiones, pero también pueden asociarse a métodos JAX-RS específicos. Internamente, el algoritmo para ejecutar una petición HTTP en el servidor se asemeja al siguiente:

```
for (filter : preMatchFilters) {
   filter.filter(request);
}

jaxrs_method = match(request);

for (filter : postMatchFilters) {
   filter.filter(request);
}

response = jaxrs_method.invoke();
```

```
for (filter : responseFilters) {
  filter.filter(request, response);
}
```

Los filtros JAX-RS tienen interfaces diferentes para peticiones y respuestas. Debido a que JAX-RS dispone de un API asíncrono, los filtros JAX-RS no se ejecutan en la misma pila de llamadas Java. Cada filtro de petición se ejecuta de principio a fin antes de invocar al método JAX-RS. Cada filtro de respuesta se ejecuta solamente después de que se disponga de una respuesta para enviar de vuelta al cliente. En el caso asíncrono, los filtros de respuesta se ejecutan después de una llamada a `resume()`, `cancel()` o de que venza un temporizador.

Filtros de petición en el servidor

Los filtros de petición son implementaciones de la interfaz `ContainerRequestFilter`:

```
package javax.ws.rs.container;

  public interface ContainerRequestFilter {
      public void filter(ContainerRequestContext requestContext)
                        throws IOException;
}
```

Esta interfaz puede utilizarse antes de realizar el *matching* del método jax-rs de la petición de entrada o después. La anotación `@PreMatching` se ejecutará antes de que el método del recurso JAX-RS correspondiente sea el designado como receptor de la petición HTTP de entrada. Los filtros *prematching* se utilizan a menudo para modificar atributos de la petición para cambiar la forma en la que se realiza el matching sobre un recurso específico. Por ejemplo, algunos *cortafuegos* no permiten invocaciones PUT y/o DELETE. Para poder superar esta limitación, muchas aplicaciones aplican al método HTTP la cabecera `X-Http-Method-Override`:

```
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ContainerRequestContext;
@Provider
@PreMatching
public class HttpMethodOverride implements ContainerRequestFilter {
  public void filter(ContainerRequestContext ctx) throws IOException {
      String methodOverride = ctx.getHeaderString("X-Http-Method-
Override");
      if (methodOverride != null) ctx.setMethod(methodOverride);
  }
}
```

Este filtro `HttpMethodOverride` se ejecutará antes de que la petición HTTP sea derivada a un método JAX-RS específico. El parámetro `ContainterRequestContext` del método `filter()` proporciona información sobre la petición como por ejemplo las cabeceras, y la URI, entre otros. El método `filter()` utiliza el parámetro `ContainerRequestContext` para chequear el valor de la cabecera `X-Http-Method-Override`. Si la cabecera ya tiene un valor asignado en la petición, el filtro sobreescribe el método HTTP realizando una llamada a `ContainerRequestFilter.setMethod()`. Los filtros pueden modificar cualquier información de la petición de entrada a través de los métodos de

`ContainerRequestContext`, pero una vez que la petición se asocia con un método JAX-RS, ningún filtro puede modificar la URI de la petición o el método HTTP.

Otro uso de los filtros de petición es la implementación de protocolos particulares de autorización. Por ejemplo, OAuth 2.0 tiene un protocolo basado en *tokens* que son transmitidos a través de la cabecera HTTP `Autorization`. Un ejemplo de implementación de este tipo de filtros podría ser la siguiente:

```
import javax.ws.rs.container.ContainerRequestFilter;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.NotAuthorizedException;

@Provider
@PreMatching
public class BearerTokenFilter implements ContainerRequestFilter {
  public void filter(ContainerRequestContext ctx) throws IOException {
    String authHeader =
 request.getHeaderString(HttpHeaders.AUTHORIZATION);
    if (authHeader == null) throw new NotAuthorizedException("Bearer");
    String token = parseToken(authHeader);
    if (verifyToken(token) == false) {
      throw new NotAuthorizedException("Bearer error=\"invalid_token\"");
    }
  }
  private String parseToken(String header) {...}
  private boolean verifyToken(String token) {...} }
}
```

En este ejemplo, si no hay una cabecera `Autorization` o si ésta es inválida, la petición es abortada con una `NotAutorizedException`. El cliente recibe una respuesta 401 con una cabecera `WWW-Autenticate` con el valor que se pasa como parámetro al constructor de la clase `notAuthorizedException`.

Filtros de respuesta del servidor

Los filtros de respuesta son implementaciones de la interfaz `ContainerResponseFilter`:

```
package javax.ws.rs.container;
public interface ContainerResponseFilter {
  public void filter(ContainerRequestContext requestContext,
                     ContainerResponseContext responseContext)
                     throws IOException;
}
```

Generalmete, usaremos estos filtros para añadir o modificar cabeceras de respuesta. Un ejemplo es queremos enviar un valor por defecto de cabecera de respuesta `Cache-Control` para cada respuesta a una petición GET, podríamos implementar lo siguiente:

```
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.core.CacheControl;
```

```
@Provider
public class CacheControlFilter implements ContainerResponseFilter {
  public void filter(ContainerRequestContext req,
                     ContainerResponseContext res)
                  throws IOException {
     if (req.getMethod().equals("GET")) {
        CacheControl cc = new CacheControl();
        cc.setMaxAge(100);
        req.getHeaders().add("Cache-Control", cc);
     }
  }
}
```

El método `ContainerResponseFilter.filter()` tiene dos parámetros. El parámetro `ContainerRequestContext` nos permite acceder a la información sobre la peticiión. En este caso, estamos comprobando si la petición es `GET`. El parámetro `ContainerResponseContext` nos permite ver, añadir y modificar la respuesta antes de que sea serializada y devuelta al cliente. En el ejemplo usamos el objeto `ContainerResponseContext` para asignar un valor a la cabecera de respuesta `Cache-Control`.

Interceptores de lectura y escritura

Los filtros modifican las cabeceras de petición o de respuesta, los interceptores de lectura y escritura modifican los cuerpos de los mensajes. Trabajan conjuntamente con un `MessageBodyReader` o `MessageBodyWriter`, y se pueden utilizar tanto en el cliente como en el servidor. Los interceptores de lectura implementan la interfaz `ReaderInterceptor`. Los interceptores de escritura implementan la interfaz `WriterInterceptor`.

```
package javax.ws.rs.ext;

public interface ReaderInterceptor {
   public Object aroundReadFrom(ReaderInterceptorContext context)
                               throws java.io.IOException,
                               javax.ws.rs.WebApplicationException;
}

public interface WriterInterceptor {
   void aroundWriteTo(WriterInterceptorContext context)
                      throws java.io.IOException,
                        javax.ws.rs.WebApplicationException;
}
```

Estos interceptores sólo son ejecutados cuando un `MessageBodyReader` o un `MessageBodyWriter` es necesario para deserializar o serializar un objeto Java a y desde un cuerpo de mensaje HTTP. También son invocados dentro de la misma pila de llamadas Java. Dicho de otra forma, un `ReaderInterceptor` "envuelve" (es un *wrapper* de) la invocación de un `MessageBodyReader.readFrom()` y un `WriterInterceptor` es un *wrapper* de la invocación del método `messageBodyWriter.writeTO()`.

Un ejemplo que ilustra el uso de estas interfaces es por ejemplo el añadir compresión a los *streams* de entrada y salida a través del *encoding* de contenidos. Aunque la mayoría de

implementaciones JAX-RS proporcionan el encoding GZIP, veamos cómo podríamos añadir este soporte utilizando un `ReaderInterceptor` y un `WriterInterceptor`:

```
@Provider
public class GZIPEncoder implements WriterInterceptor {
   public void aroundWriteTo(WriterInterceptorContext ctx)
                  throws IOException, WebApplicationException {
     GZIPOutputStream os = new GZIPOutputStream(ctx.getOutputStream());
     ctx.getHeaders().putSingle("Content-Encoding", "gzip");
     ctx.setOutputStream(os);
     ctx.proceed();
     return;
  }
}
```

El parámetro `WriterInterceptorContext` nos permite ver y modificar las cabeceras HTTP asociadas con esta invocación. Ya que los interceptores pueden utilizarse tanto en la parte del cliente como en la del servidor, estas cabeceras representan tanto una petición del cliente como una respuesta del servidor. En el ejemplo, nuestro método `aroundWriteTo()` utiliza el `WriterInterceptorContext` para obtener y reemplazar el `OutputStream` del cuerpo del mensaje HTTP con un `GZipOutputStream`. También podemos utilizarlo para añadir una cabecera `Content-Encoding`. La llamada a `WriterInterceptorContext.proceed()` invoca al siguiente `WriterInterceptor` registrado, o si no hay ninguno invocan al método `MessageBodyWriter.writeTo()` subyacente.

Veamos cómo se implementaría el correspondiente `ReaderInterceptor` para este ejemplo de *encoding*:

```
@Provider
public class GZIPDecoder implements ReaderInterceptor {
   public Object aroundReadFrom(ReaderInterceptorContext ctx)
                           throws IOException {
      String encoding = ctx.getHeaders().getFirst("Content-Encoding");
      if (!"gzip".equalsIgnoreCase(encoding)) {
         return ctx.proceed();
      }
      GZipInputStream is = new GZipInputStream(ctx.getInputStream());
      ctx.setInputStream(is);
      return ctx.proceed(is);
   }
}
```

En este ejemplo, el parámetro `ReaderInterceptorContext` nos permite ver y modificar las cabeceras HTTP asociadas con esta invocación, tanto en la parte del cliente, como en la del servidor. En este ejemplo, nuestro método `aroundReadFrom()` utiliza un objeto `ReaderInterceptorContext` para comprobar primero si el cuerpo del mensaje está codificado como GZIP. Si no lo está, devuelve una llamada a `ReaderInterceptorContext.proceed()`. El objeto `ReaderInterceptorContext` también se utiliza para obtener y reemplazar el `InputStream` del cuerpo del mensaje HTTP con un `GZipInputStream`. La llamada a `ReaderInterceptorContext.proceed()` invocará al siguiente `ReaderInterceptor` registrado, o si no hubiese ninguno, al métood `MessageBodyReader.readFrom()` subyacente. El valor devuelto por

el método `proceed()` puede ser de cualquier tipo devuelto por el método `MessageBodyReader.readFrom()`.

Otros ejemplos de interceptores podrían ser para firmar digitalmente o encriptar el cuerpo del mensaje.

Filtros en el cliente

El API del cliente JAX-RS tiene sus propias interfaces de filtros de petición y respuesta:

```
package javax.ws.rs.client;

public interface ClientRequestFilter {
  public void filter(ClientRequestContext requestContext) throws
 IOException;
}

public interface ClientResponseFilter {
  public void filter(ClientRequestContext requestContext,
                     ClientResponseContext responseContext) throws
 IOException;
}
```

Vamos a ver un ejemplo de uso de estas dos interfaces implementando un sistema de caché en el cliente. En nuestro ejemplo, queremos que las entradas de la caché tengan una duración basada en los metadatos de las cabeceras de respuesta `Cache-Control`. Queremos realizar GETs condicionales en el cliente si se produce una petición a una entrada de la caché que ha expirado. Veamos como implementar primero nuestro `ClientRequestFilter`:

```
import javax.ws.rs.client.ClientRequestFilter;
import javax.ws.rs.client.ClientRequestContext;

public class ClientCacheRequestFilter implements ClientRequestFilter {
  private Cache cache;
  public ClientCacheRequestFilter(Cache cache) {
    this.cache = cache;
  }
  public void filter(ClientRequestContext ctx) throws IOException {
    if (!ctx.getMethod().equalsIgnoreCase("GET")) return;
    CacheEntry entry = cache.getEntry(request.getUri());
    if (entry == null) return;
    if (!entry.isExpired()) {
       ByteArrayInputStream is = new
 ByteArrayInputStream(entry.getContent());
       Response response = Response.ok(is)
                           .type(entry.getContentType()).build();
       ctx.abortWith(response);
       return;
    }

    String etag = entry.getETagHeader();
    String lastModified = entry.getLastModified();

    if (etag != null) {
```

```
      ctx.getHeaders.putSingle("If-None-Match", etag);
    }
    if (lastModified != null) {
      ctx.getHeaders.putSingle("If-Modified-Since", lastModified);
    }
  }
}
```

Nuestro filtro de petición deberá registrarse como un *singleton* y requiere crearse con una instancia de la clase `Cache`. El método `ClientCacheRequestFilter.filter()` realiza varias acciones basadas en el estado de la correspondiente caché. Primero comprueba la instancia de `ClientRequestContext` para ver si se trata de una petición HTTP GET. Si no es así, no hacemos nada y terminamos. Si se trata de una petición GET, buscamos la URI de la petición en la caché. Si no hay ninguna entrada, terminamos. En caso contrario, debemos ver si ésta a expirado o no. Si no ha expirado, creamos un objeto `Response` que devuelva "200 OK". A continuación añadimos al objeto `Response` el contenido y el valor de la cabecera `Content-Header` almacenados y abortamos la invocación llamando al método `ClientRequestContext.abortWith()`. Dependiendo de cómo se ha iniciado la invocación del cliente, el objeto `Response` abortado se devolverá directamente a la aplicación, o será deserializado en el correspondiente tipo Java. Si la entrada de la caché ha expirado, realizamos un GET condicional incluyendo las cabeceras de respuesta `If-None-Match` y/o `If-Modified-Since` con los valores almacenados en la entrada de la caché.

Veamos a continuación como sería la implementación del filtro de respuesta:

```
public class CacheResponseFilter implements ClientResponseFilter {
  private Cache cache;
  public CacheResponseFilter(Cache cache) {
    this.cache = cache;
  }
  public void filter(ClientRequestContext request,
                     ClientResponseContext response)
                        throws IOException {
    if (!request.getMethod().equalsIgnoreCase("GET")) return;
    if (response.getStatus() == 200) {
      cache.cacheResponse(response, request.getUri());
    } else if (response.getStatus() == 304) {
        CacheEntry entry = cache.getEntry(request.getUri());
        entry.updateCacheHeaders(response);
        response.getHeaders().clear();
        response.setStatus(200);
        response.getHeaders().putSingle("Content-Type",
 entry.getContentType());
        ByteArrayInputStream is = new
 ByteArrayInputStream(entry.getContent());
        response.setInputStream(is);
    }
  }
}
```

El método `CacheResponseFileter.filter()` comienza comprobando si la petición invocada es HTTP GET. Si no es así, simplemente termina. Si el estado de la respuesta es "200 OK", entonces pedimos al objeto `Cache` que almacene la respuesta para la URI especificada en la petición. El método `Cache.cacheResponse()` se encarga de almacenar

la respuesta (tanto las cabeceras de respuesta relevantes como el cuerpo del mensaje). Si la respuesta fuese "304, Not Modified", esto significaría que hemos realizado un GET condicional con éxito. En este caso, actualizamos la entrada de la caché con las cabeceras de respuesta `ETag` o `Last-Modified`. Además, como la respuesta no tendrá ningún cuerpo de mensaje, debemos reconstruir la respuesta basada en la entrada de la caché. Borramos todas las cabeceras de `ClientResponseContext` y asignamos el contenido adecuado de `Content-Type`. Finalmente sobreescribimos el `InputStream` de respuesta con el *buffer* almacenado en la entrada de la caché.

Despliegue de filtros e interceptores

En la parte del servidor, para desplegar tanto los filtros como los interceptores podemos utilizar la anotación @Provider, y dejar que el *runtime* de JAX-RS los detecte y registre automáticamente, o bien podemos añadir los filtros o interceptores a las clases `Application`, bien como *singletons* o *per-request*.

En la parte del cliente, podemos hacerlo de la misma forma. Aunque en este caso, hay unos cuantos componentes en al API cliente que implementan la interfaz `Configurable`. Esta interfaz tiene un método `register()` que nos permite pasar a dicho componente nuestra clase filtro o interceptor, o bien el singleton correspondiente. `ClientBuilder`, `Client` y `WebTarget` implementan la interfaz `Configurable`.

Orden de ejecución de filtros en interceptores

Cuando tenemos registrados más de un filtro o interceptor, puede que nos interese establecer un orden de ejecución entre ellos. Por ejemplo, normalmente no querremos que usuarios no autenticados ejecuten ninguno de nuestros componentes JAX-RS. Así, si tenemos un filtro para autenticar a un cliente, probablemente queramos que éste se ejecute el primero. Otro ejemplo podría ser si disponemos de un filtro para codificar en formato GZIP, y otro para encriptar el cuerpo del mensaje. Probablemente no queramos encriptar la representación en formato GZIP. Por lo tanto, el orden es importante.

JAX-RS permite asignar una prioridad numérica a los filtros en interceptores, bien utilizando la anotación @Priority, o bien a través de la interfaz `Configurable`. El *runtime* de JAX-RS ordena los filtros e interceptores basándose en su prioridad numérica, de forma que primero van los que tengan asignado un número más pequeño.

```
package javax.annotation;

public @interface Priority {
  nt value();
}
```

Vamos a mostrar cómo utilizar la anotación @Priority utilizando uno de los ejemplos de los apartados anteriores:

```
import javax.annotation.Priority;
import javax.ws.rs.Priorities;

@Provider
@PreMatching
@Priority(Priorities.AUTHENTICATION)
public class BearerTokenFilter implements ContainerRequestFilter {
```

```
...
}
```

La anotación @Priority puede tomar cualquier valor numérico que queramos. La clase `Priorities` especifica algunas constantes que podemos utilizar cuando utilicemos la antación @Priority:

```
package javax.ws.rs;
public final class Priorities {
  private Priorities() {
  // prevents construction
  }

  // Security authentication filter/interceptor priority
  public static final int AUTHENTICATION = 1000;

  // Security authorization filter/interceptor priority
  public static final int AUTHORIZATION = 2000;

  // Header decorator filter/interceptor priority
  public static final int HEADER_DECORATOR = 3000;

  //Message encoder or decoder filter/interceptor priority
  public static final int ENTITY_CODER = 4000;

  //User-level filter/interceptor priority
  public static final int USER = 5000;
}
```

Si no especificamos ninguna prioridad, la prioridad por defecto es `USER,5000`.

4.8. 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 enfiar 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` condificada 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 passwork 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.

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/addUser.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 el 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 depende de 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 localizados 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>/services/customers</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>jaxrs</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>/services/customers</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, @Permi tAll, 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 autentificada. 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 log.

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.

4.9. Ejercicios

Debido a la extensión de las clases de teoría y al poco tiempo que tuvimos para hacer ejercicios en clase, no hay que entregar los ejercicios de la sesión 4 y se modifica la puntuación de los ejercicios de las sesiones 1, 2 y 3

Para los ejercicios de esta sesión proporcionamos el proyecto s4-foroAvanzado.

La estructura lógica del proyecto es la misma que la de la sesión anterior. Hemos implementado dos recursos (`UsuariosResource` y `MensajesResource`), y dos subrecursos (`UsuarioResource` y `MensajeResource`).

Hemos incluido un test inicial en el que hemos añadido un usuario nuevo, así como algún mensaje.

Uso de Hateoas y Json (NO HAY QUE ENTREGARLO)

Vamos a añadir a los servicios enlaces a las operaciones que podemos realizar con cada recurso, siguiendo el estilo Hateoas.

- 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).
- 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).
- Implementa tres nuevos tests para probar las modificaciones realizadas. En todos los casos recuperaremos la respuesta formato Json (JsonArray, o bien JsonObject, en el caso de recuperar una lista de objetos Json, o uno sólo). Los nombres de los tests serán los siguientes:
  - # `test2ConsultaUsuario()`

    Comprobaremos que el valor de la uri asociada a la relación "self" es la correcta
  - # `test3ConsultaTodosUsuarios()`

    Será suficiente con comprobar que el número de usuarios es el correcto
  - # `test4VerMensaje()`

    Comprobaremos que el número de "Links" para cada mensaje de la lista es el correcto

Ejercicio seguridad (NO HAY QUE ENTREGARLO)

Vamos ahora a restringir el acceso al servicio para que sólo usuarios registrados puedan realizar modificaciones. Se pide:

- Añadir al usuario "pepe" en el "ApplicationRealm" de wildfly, con la contraseña "pepe", y perteneciente al grupo (rol) "registrado"
- Configurar, mediante seguridad declarativa, que las operaciones de modificación (POST, PUT y DELETE) sólo la podrán realizar los usuarios con rol registrado. Utilizar autentificación de tipo BASIC.

b) Ahora vamos a hacer que la modificación o borrado de usuarios sólo pueda realizarlas el mismo usuario que va a modificarse. 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)`

c) 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.