Indice

1. REST vs Web Services

1.1. ¿Qué es un Servicio Web?

El consorcio W3C define los Servicios Web como sistemas software diseñados para soportar una interacción interoperable máquina a máquina sobre una red. Los Servicios Web suelen ser APIs Web que pueden ser accedidas dentro de una red (principalmente Internet) y son ejecutados en el sistema que los aloja.

La definición de Servicios Web propuesta alberga muchos tipos diferentes de sistemas, pero el caso común de uso de refiere a clientes y servidores que se comunican mediante mensajes XML que siguen el estándar SOAP.

En los últimos años se ha popularizado un estilo de arquitectura Software conocido como REST *(Representational State Transfer)*. Este nuevo estilo ha supuesto una nueva opción de estilo de uso de los Servicios Web. A continuación se listan los tres estilos de usos más comunes:

- **RPC: Remote Procedure Calls** (Llamadas a Procedimientos Remotos). Los Servicios Web basados en RPC presentan una interfaz de llamada a procedimientos y funciones distribuidas, lo cual es familiar a muchos desarrolladores. Típicamente, la unidad básica de este tipo de servicios es la operación WSDL (WSDL es un descriptor del Servicio Web, es decir, el homólogo del IDL para COM).

  Las primeras herramientas para Servicios Web estaban centradas en esta visión. Algunos lo llaman la primera generación de Servicios Web. Esta es la razón por la que este estilo está muy extendido. Sin embargo, ha sido algunas veces criticado por no ser débilmente acoplado, ya que suele ser implementado por medio del mapeo de servicios directamente a funciones específicas del lenguaje o llamadas a métodos. Muchos especialistas creen que este estilo debe desaparecer.

- **SOA: Service-Oriented Architecture** (Arquitectura Orientada a Servicios). Los Servicios Web pueden también ser implementados siguiendo los conceptos de la arquitectura SOA, donde la unidad básica de comunicación es el mensaje, más que la operación. Esto es típicamente referenciado como servicios orientados a mensajes.

  Los Servicios Web basados en SOA son soportados por la mayor parte de desarrolladores de software y analistas. Al contrario que los Servicios Web basados en RPC, este estilo es débilmente acoplado, lo cual es preferible ya que se centra en el "contrato" proporcionado por el documento WSDL, más que en los detalles de implementación subyacentes.

- **REST: REpresentation State Transfer.** Los Servicios Web basados en REST intentan emular al protocolo HTTP o protocolos similares mediante la restricción de establecer la interfaz a un conjunto conocido de operaciones estándar (por ejemplo GET, PUT,…). Por tanto, este estilo se centra más en interactuar con recursos con estado, que con mensajes y operaciones.

1.2. ¿Qué es REST realmente?

REST (Representational State Transfer) es un estilo de arquitectura de software para sistemas hipermedias distribuidos tales como la Web. El término fue introducido en la tesis doctoral de Roy Fielding en 2000, quien es uno de los principales autores de la especificación de HTTP.

En realidad, REST se refiere estrictamente a una colección de principios para el diseño de arquitecturas en red. Estos principios resumen cómo los recursos son definidos y diseccionados. El término frecuentemente es utilizado en el sentido de describir a cualquier interfaz que transmite datos específicos de un domino sobre HTTP sin una capa adicional, como hace SOAP. Estos dos significados pueden chocar o incluso solaparse. Es posible diseñar un sistema software de gran tamaño de acuerdo con la arquitectura propuesta por Fielding sin utilizar HTTP o sin interactuar con la Web. Así como también es posible diseñar una simple interfaz XML+HTTP que no sigue los principios REST, y en cambio seguir un modelo RPC.

Cabe destacar que REST no es un estándar, ya que es tan solo un estilo de arquitectura. Aunque REST no es un estándar, está basado en estándares:

- HTTP
- URL
- Representación de los recursos: XML/HTML/GIF/JPEG/…
- Tipos MIME: text/xml, text/html,…

1.3. ¿Cuál es la motivación de REST?

La motivación de REST es la de capturar las características de la Web que la han hecho tan exitosa.

Si pensamos un poco en este éxito, nos daremos cuenta que la Web ha sido la única aplicación distribuida que ha conseguido ser escalable al tamaño de Internet. El éxito lo debe al uso de formatos de mensaje extensibles y estándares, pero además cabe destacar que posee un esquema de direccionamiento global (estándar y extensible a su vez).

En particular, el concepto central de la Web es un espacio de URIs unificado. Las URIs permiten la densa red de enlaces que permiten a la Web que sea tan utilizada. Por tanto, ellos consiguen tejer una mega-aplicación.

Las URIs identifican recursos, los cuales son objetos conceptuales. La representación de tales objetos se distribuye por medio de mensajes a través de la Web. Este sistema es extremadamente desacoplado.

Estas características son las que han motivado para ser utilizadas como guía para la evolución de la Web.

1.4. ¿Cuáles son los principios de REST?

El estilo de arquitectura subyacente a la Web es el modelo REST. Los objetivos de este estilo de arquitectura se listan a continuación:

- Escalabilidad de la interacción con los componentes. La Web ha crecido exponencialmente sin degradar su rendimiento. Una prueba de ellos es la variedad de clientes que pueden acceder a través de la Web: estaciones de trabajo, sistemas industriales, dispositivos móviles,…

- Generalidad de interfaces. Gracias al protocolo HTTP, cualquier cliente puede interactuar con cualquier servidor HTTP sin ninguna configuración especial. Esto no es del todo cierto para otras alternativas, como SOAP para los Servicios Web.

- Puesta en funcionamiento independiente. Este hecho es una realidad que debe tratarse cuando se trabaja en Internet. Los clientes y servidores pueden ser puestas en funcionamiento durante años. Por tanto, los servidores antiguos deben ser capaces de entenderse con clientes actuales y viceversa. Diseñar un protocolo que permita este tipo de características resulta muy complicado. HTTP permite la extensibilidad mediante el uso de las cabeceras, a través de las URIs, a través de la habilidad para crear nuevos métodos y tipos de contenido.

- Compatibilidad con componentes intermedios. Los más populares intermediaros son varios tipos de proxys para Web. Algunos de ellos, las caches, se utilizan para mejorar el rendimiento. Otros permiten reforzar las políticas de seguridad: firewalls. Y por último, otro tipo importante de intermediarios, gateway, permiten encapsular sistemas no propiamente Web. Por tanto, la compatibilidad con intermediarios nos permite reducir la latencia de interacción, reforzar la seguridad y encapsular otros sistemas.

REST logra satisfacer estos objetivos aplicando cuatro restricciones:

- Identificación de recursos y manipulación de ellos a través de representaciones. Esto se consigue mediante el uso de URIs. HTTP es un protocolo centrado en URIs. Los recursos son los objetos lógicos a los que se le envían mensajes. Los recursos no pueden ser directamente accedidos o modificados. Más bien se trabaja con representaciones de ellos. Cuando se utiliza un método PUT para enviar información, se coge como una representación de lo que nos gustaría que el estado del recurso fuera. Internamente el estado del recurso puede ser cualquier cosa desde una base de datos relacional a un fichero de texto.

- Mensajes auto descriptivos. REST dicta que los mensajes HTTP deberían ser tan descriptivos como sea posible. Esto hace posible que los intermediarios interpreten los mensajes y ejecuten servicios en nombre del usuario. Uno de los modos que HTTP logra esto es por medio del uso de varios métodos estándares, muchos encabezamientos y un mecanismo de direccionamiento. Por ejemplo, las cachés Web saben que por defecto el comando GET es cacheable (ya que es *side-effect-free*) en cambio POST no lo es. Además saben cómo consultar las cabeceras para controlar la caducidad de la información. HTTP es un protocolo sin estado y cuando se utiliza adecuadamente, es posible es posible interpretar cada mensaje sin ningún conocimiento de los mensajes precedentes. Por ejemplo, en vez de logarse del modo que lo hace el protocolo FTP, HTTP envía esta información en cada mensaje.

- Hipermedia como un mecanismo del estado de la aplicación. El estado actual de una aplicación Web debería ser capturada en uno o más documentos de hipertexto, residiendo tanto en el cliente como en el servidor. El servidor conoce sobre el estado de sus recursos, aunque no intenta seguirle la pista a las sesiones individuales de los clientes. Esta es la misión

del navegador, él sabe cómo navegar de recurso a recurso, recogiendo información que el necesita o cambiar el estado que el necesita cambiar.

En la actualidad existen millones de aplicaciones Web que implícitamente heredan estas restricciones de HTTP. Hay una disciplina detrás del diseño de sitios Web escalables que puede ser aprendida de los documentos de arquitectura Web o de varios estándares. Por otra parte, también es verdad que muchos sitios Web comprometen uno más de estos principios, como por ejemplo, seguir la pista de los usuarios moviéndose a través de un sitio. Esto es posible dentro de la infraestructura de la Web, pero daña la escalabilidad, volviendo un medio sin conexión en todo lo contrario.

Los defensores de REST han creído que estas ideas son tan aplicables a los problemas de integración de aplicaciones como los problemas de integración de hipertexto. Fielding es bastante claro diciendo que REST no es la cura para todo. Algunas de estas características de diseño no serán apropiadas para otras aplicaciones. Sin embargo, aquellos que han decidido adoptar REST como un modelo de servicio Web sienten que al menos articula una filosofía de diseño con fortaleza, debilidades y áreas de aplicabilidad documentada.

1.5. ¿Cómo sería un ejemplo de diseño basado en REST?

De nuevo tomaremos como ejemplo a la Web. La Web evidentemente es un ejemplo clave de diseño basado en REST, ya que muchos principios son la base de REST. Posteriormente mostraremos un posible ejemplo real aplicado a Servicios Web.

La Web consiste del protocolo HTTP, de tipos de contenido, incluyendo HTML y otras tecnologías tales como el Domain Name System (DNS).

Por otra parte, HTML puede incluir JavaScript y applets, los cuales dan soporte al code-on-demand, y además tiene implícitamente soporte a los vínculos. HTTP posee un interfaz uniforme para acceso a los recursos, el cual consiste de URIs, métodos, códigos de estado, cabeceras y un contenido guiado por tipos MIME.

Los métodos HTTP más importantes son PUT, GET, POST y DELETE. Ellos suelen ser comparados con las operaciones asociadas a la tecnología de base de datos, operaciones CRUD: CREATE, READ, UPDATE, DELETE. Otras analogías pueden también ser hechas como con el concepto de copiar-y-pegar (Copy&Paste). Todas las analogías se representan en la siguiente tabla:

Acción	HTTP	SQL	Copy&Paste	Unix Shell
Crear	PUT	Insert	Pegar	>
Leer	GET	Select	Copiar	<
Actualizar	POST	Update	Pegar después	>>
Eliminar	DELETE	Delete	Cortar	del/rm

Las acciones (comandos) CRUD se diseñaron para operar con datos atómicos dentro del contexto de una transacción con la base de datos. REST se diseña alrededor de transferencias atómicas de un estado más complejo, tal que puede ser visto como la transferencia de un documento estructurado de una aplicación a otra.

El protocolo HTTP separa las nociones de un servidor y un navegador. Esto permite a la implementación cada uno variar uno del otro, basándose en el concepto cliente/servidor. Cuando utilizamos REST, HTTP no tiene estado. Cada mensaje contiene toda la información necesaria para comprender la petición cuando se combina el estado en el recurso. Como resultado, ni el cliente ni el servidor necesita mantener ningún estado en la comunicación. Cualquier estado mantenido por el servidor debe ser modelado como un recurso.

La restricción de no mantener el estado puede ser violada mediante cookies que mantienen las sesiones. Fielding advierte del riesgo a la privacidad y seguridad que frecuentemente surge del uso de cookies, así como la confusión y errores que pueden resultar de las interacciones entre cookies y el uso del botón “Go back” del navegador.

HTTP proporciona mecanismos para el control del *caching* y permite que ocurra una conversación entre el navegador y la caché del mismo modo que se hace entre el navegador y el servidor Web.

1.6. ¿Cómo crear una interfaz basada en REST?

En vez de cubrir esto desde un punto de vista arquitectural, es aconsejable realizarlo a modo de receta. Existen una serie de pasos a tomar y una serie de preguntas a responder.

Antes de empezar a pensar en el servicio, debemos responder las siguientes preguntas en orden:

- ¿Qué son las URIs? Las “cosas” identificadas por URIs son **recursos**. Aunque es más apropiado decir que **los recursos son identificados mediante URIs.**

  Si solo existe una única URI como punto de acceso, posiblemente se esté creando un protocolo RPC. En tal caso, se debe desmenuzar el problema en tipos de recursos que se quieren manipular. Dos lugares donde se debe considerar cuando se buscan los recursos potenciales son las colecciones y las interfaces de búsqueda. Una colección de recursos es en sí mismo un recurso. Una interfaz de búsqueda también lo es, ya que el resultado de una búsqueda es otro conjunto de recursos.

  A modo de ejemplo, supongamos un sistema de mantenimiento de una lista de contactos de empleados. En tal sistema cada usuario debería tener su propia URI con una apropiada representación. Además, la colección de recursos es otro recurso.

  Por tanto, hemos identificado dos tipos de recursos, por tanto habrá dos tipos de URIs: de Instancia y Colección.

  - *Employee* (Una URI por empleado).
  - *AllEmployee* (Listado de todos los empleados).

- ¿Cuál es el formato? Cuando hablamos informalmente de formato, estamos hablando de la representación. Como ya hemos comentado con anterioridad, no se puede acceder directamente al recurso, hay que obtener una representación. Esta representación puede ser un documento HTML, XML, una imagen,… dependiendo de la situación.

Para cada uno de los recursos, se tiene que decidir cuál va a ser su representación. Si es posible, reutilizar formatos existentes, ayudará a incrementar la oportunidad de que nuestro sistema se componga con otros.

Continuando con nuestro ejemplo, el formato de representación va a ser XML, ya que es el principal formato para intercambio de información. El formato del empleado podría ser el siguiente:

```
<employee xmlns='HTTP://example.org/my-example-ns/'>
      <name>Full name goes here.</name>
      <title>Persons title goes here.</title>
      <phone-number>Phone number goes here.</phone-number>
</employee>
```

Para el listado de empleados podríamos tomar este otro:

```
<employee-list xmlns='HTTP://example.org/my-example-ns/'>
      <employee-ref href="URI of the first employee"/>
      Full name of the first employee goes here.</employee>

<employee-ref href="URI of employee #2"/>Full name</employee>
.
.
<employee-ref href="URI of employee #N"/>Full name</employee>
      </employee-list>
```

- ¿Qué métodos son soportados en cada URI? En este apartado tenemos que definir cómo van a ser referenciados los recursos. Por medio de las URIs y los métodos soportados el acceso va a ser posible. El acceso de puede hacer de muchas formas, recibiendo una representación del recurso (GET o HEAD), añadiendo o modificando una representación (POST o PUT) y eliminando algunas o todas las representaciones (DELETE).

HTTP	CRUD	Descripción
POST	Create	Crear un nuevo recurso
GET	Retrieve	Obtener la representación de un recurso
PUT	Update	Actualizar un recurso
DELETE	Delete	Eliminar un recurso

Continuando con nuestro ejemplo, ahora vamos a combinar recursos, representación y métodos:

Recurso	Método	Representación
Employee	GET	Formato del empleado
Employee	PUT	Formato del empleado
Employee	DELETE	-
All Employees	GET	Formato de la lista de empleados
All Employees	POST	Formato del empleado

- ¿Qué códigos de estado pueden ser devueltos? No solo es necesario conocer que tipo de representación va a ser devuelta, también es necesario enumerar los códigos de estado HTTP típicos que podrían ser devueltos.

  Volvemos a actualizar nuestra tabla para mostrar los códigos de estado:

Recurso	Método	Representación	Códigos de estado
Employee	GET	Formato del empleado	200, 301, 410
Employee	PUT	Formato del empleado	200, 301, 400, 410
Employee	DELETE	-	200, 204
All Employees	GET	Formato de la lista de empleados	200, 301
All Employees	POST	Formato del empleado	201, 400

1.7. ¿Por qué surge el debate entre REST y los Servicios Web?

Esta pregunta es un tanto errónea, aunque es bastante típica en los foros de discusión. REST es un estilo, mientras que los servicios Web son sistemas software. Por tanto, no es posible la comparación de ambos conceptos. Por otra parte, popularmente se generaliza el concepto de servicio Web con el de servicio Web basado en SOAP. Como hemos visto en apartados anteriores, es posible diseñar servicios Web basados en REST, es decir tomando REST como estilo de diseño.

Por tanto, esta pregunta debería haber sido formulada como ha sido hecha en el siguiente apartado.

1.8. ¿Por qué surge el debate entre los Servicios Web basados en REST y SOAP?

Muchos diseñadores de Servicios Web están llegando a la conclusión que SOAP es demasiado complicado. Por tanto, están comenzando a utilizar Servicios Web basados en REST para mostrar cantidades de datos masivos. Este es el caso de grandes empresas como eBay y Google.

El problema principal surge del propósito inicial de SOAP. Esta tecnología fue originalmente pensada para ser una versión, sobre Internet, de DCOM o CORBA. Así lo demuestra su predecesor, el protocolo XML-RPC. Estas tecnologías lograron un éxito limitado antes de ser adaptadas. Esto es debido a que este tipo de tecnologías, las basadas en modelos RPC (*Remote Procedure Call*) son más adecuadas para entornos aislados, es decir, entornos donde se conoce perfectamente el entorno. La evolución en este tipo de sistemas es sencilla, se modifica cada usuario para que cumpla con los nuevos requisitos.

Pero cuando el número de usuarios es muy grande es necesario emplear una estrategia diferente. Se necesita organizar frameworks que permitan evolucionar, tanto por el lado del cliente como del servidor. Se necesita proponer un mecanismo explícito para la interoperabilidad de los sistemas que no poseen la misma API. Protocolos basados en RPC no son adecuados para este tipo de sistemas, ya que cambiar su interfaz resulta complicado.

Por esta razón, se intenta tomar como modelo a la Web. A primera vista se puede pensar que SOAP lo hace, ya que utiliza HTTP como medio de transporte. Pero Fielding argumenta que la Web funciona mejor cuando se utiliza en el estilo que lo hace REST. Utilizar HTTP como medio de transporte para protocolos de aplicación a través de firewalls es una idea equivocada. Esto reduce la efectividad de tener un firewall. Lo cual aumenta las posibilidades de nuevos agujeros de seguridad.

Sin embargo, los partidarios de SOAP argumentan que gracias a la tecnología existente que permite a los diseñadores encapsular la complejidad del sistema, dando lugar a interfaces generadas automáticamente que permiten facilitar el diseño del sistema.

Según argumenta Spolsky lo interesante de SOAP es que permite utilizar lenguajes de alto nivel para llamar y para implementar el servicio. Añade que "Alegar que SOAP es malo porque el formato de conexión es horrible y pesado es como alegar que nadie debería utilizar Pentiums porque su juego de instrucciones es mucho más complicado que el juego de instrucciones del 8086. Eso es cierto, pero por esa razón existen los compiladores."

1.9. ¿Cómo diseñar un Servicio Web basado en REST?

Las pautas a seguir en el diseño de servicios Web basados en REST son las mismas que las descritas en el apartado dedicado a crear una interfaz REST. Por otra parte, hemos ampliado dicha información en el contexto de los Servicios Web:

- Identificar todas las entidades conceptuales que se desean exponer como servicio.

- Crear una URL para cada recurso. Los recursos deberían ser nombres no verbos (acciones). Por ejemplo no utilizar esto:

  ```
  http://www.service.com/entities/getEntity?id=001
  ```

  Como podemos observar, *getEntity* es un verbo. Mejor utilizar el estilo REST, un nombre:

  ```
  http://www.service.com/entities/001
  ```

- Categorizar los recursos de acuerdo con si los clientes pueden obtener un representación del recurso o si pueden modificarlo. Para el primero, debemos hacer los recursos accesibles utilizando un HTTP GET. Para el último, debemos hacer los recursos accesibles mediante HTTP POST, PUT y/o DELETE.

- Todos los recursos accesibles mediante GET no deberían tener efectos secundarios. Es decir, los recursos deberían devolver la representación del recurso. Por tanto, invocar al recurso no debería ser el resultado de modificarlo.

- Ninguna representación debería estar aislada. Es decir, es recomendable poner hipervínculos dentro de la representación de un recurso para permitir a los clientes obtener más información.

- Especificar el formato de los datos de respuesta mediante un esquema (DTD, W3C Schema, …). Para los servicios que requieran un POST o un PUT es aconsejable también proporcionar un esquema para especificar el formato de la respuesta.

- Describir como nuestro servicio ha de ser invocado, mediante un documento WSDL/WADL o simplemente HTML.

1.10. ¿Cuáles son las características de REST y SOAP en definitiva?

Según hemos visto a lo largo del documento, el principal beneficio de SOAP recae en ser fuertemente acoplado, lo que permite poder ser testado y depurado antes de poner en marcha la aplicación. En cambio, las ventajas de la aproximación basada en REST recaen en la potencial escalabilidad de este tipo de sistemas, así como el acceso con escaso consumo de recursos a sus operaciones debido al limitado número de operaciones y el esquema de direccionamiento unificado.

A modo de resumen, veamos las características de ambas aproximaciones en la siguiente tabla:

	REST	**SOAP**
Características	Las operaciones se definen en los mensajes. Una dirección única para cada instancia del proceso. Cada objeto soporta las operaciones estándares definidas. Componentes débilmente acoplados.	Las operaciones son definidas como puertos WSDL. Dirección única para todas las operaciones. Múltiple instancias del proceso comparten la misma operación. Componentes fuertemente acoplados.
Ventajas declaradas	Bajo consumo de recursos. Las instancias del proceso son creadas explícitamente. El cliente no necesita información de enrutamiento a partir de la URI inicial. Los clientes pueden tener una interfaz “listener” (escuchadora) genérica para las notificaciones. Generalmente fácil de construir y adoptar.	Fácil (generalmente) de utilizar. La depuración es posible. Las operaciones complejas pueden ser escondidas detrás de una fachada. Envolver APIs existentes es sencillo Incrementa la privacidad. Herramientas de desarrollo.
Posibles desventajas	Gran número de objetos. Manejar el espacio de nombres (URIs) puede ser engorroso. La descripción sintáctica/semántica muy informal (orientada al usuario). Pocas herramientas de desarrollo.	Los clientes necesitan saber las operaciones y su semántica antes del uso. Los clientes necesitan puertos dedicados para diferentes tipos de notificaciones. Las instancias del proceso son creadas implícitamente.

1.11. ¿Es realmente REST la panacea?

Evidentemente no. Existen una serie de puntos que resaltan los partidarios de SOAP sobre REST:

- **Asincronía:** Esto normalmente tiene diferentes significados, pero el concepto más común es que dos programas en comunicación (cliente y servidor, en nuestro caso) deberían no necesitar estar en comunicación constante mientras uno de ellos está realizando una operación costosa. Deberían existir algún tipo de mecanismo mediante el cual un participante pueda avisar al otro cuando el resultado esté listo. SOAP no soporta directamente esto, sin embargo puede utilizarse de una manera asíncrona por medio del uso de un transporte asíncrono, aunque esto no está todavía estandarizado. Por otra parte, existen una variedad de propuestas para hacer asíncrono a HTTP, ya que existen una variedad de aspectos a asincronizar. Una de tales especificaciones es conocida como "HTTPEvents", la cual está propuesta para su estandarización.

- **Routing (Enrutamiento):** Los mensajes HTTP son enrutados del cliente a los proxys y a los servidores. Esto es una especie de enrutamiento controlado por la red, pero a veces es adecuado poder controlar por el cliente el enrutamiento de los mensajes mediante la definición de una ruta entre los nodos. Los partidarios de REST creen que esto puede ser uno de los pocos puntos en común con SOAP, ya que el modo que SOAP utiliza el "Routing" es compatible con la Web. Por tanto, los investigadores de REST trabajan en un modelo similar al de SOAP.

- **Fiabilidad:** Este concepto al igual que el de asincronía puede tener varias interpretaciones. El más común es el envío de una única vez del mensaje. Esto es relativamente fácil de lograr en HTTP, pero no es una característica integrada. La solución consiste en enviar repetidamente e internamente el mensaje hasta obtener una confirmación. Pero el problema recae en el uso del comando POST, recordemos que no es *free-side-effect*. Por tanto, es necesario incluir en la cabecera algún tipo de identificador del mensaje. En el caso de que el destinatario obtenga un mensaje previamente analizado, el mensaje se desecha.

- **Seguridad:** Los defensores de SOAP argumentan que REST no dispone de mecanismos tan completos de seguridad como es el caso de la especificación WS-Security. Aunque en realidad, esto debería ser estudiado más profundamente.

  Una manera muy sencilla de controlar la seguridad consiste en utilizar listas de control de acceso (ACLs). Esta medida de seguridad se puede tomar tanto a nivel global como a nivel local (URIs). Este sistema resulta mucho más difícil de implementar en soluciones basadas en RPC, ya que el sistema de direccionamiento es propietario y expresado en parámetros arbitrarios. Por otra parte, es posible utilizar las características de seguridad implícitas de HTTP: HTTPS y el sistema de autorización y autenticación de HTTP.

- **Modelo extensible:** SOAP tiene un modelo extensible que permite al creador del mensaje SOAP ser muy explícito sobre si comprender una porción del mensaje es opcional u obligatorio. Esto también permite al encabezamiento ser objetivo de algunos intermediarios. Existe una extensión de HTTP con muchas de las mismas ideas (posiblemente la versión SOAP esté basada en esta versión), pero no es tan conocida ni está tan clara sintácticamente.

- **Descripción del servicio:** SOAP tiene WSDL, pero muchos alegan que HTTP no tiene nada similar hasta la fecha. Esto no es del todo cierto, WADL (Web Application Description Language) proporciona una simple alternativa a WSDL para el uso con aplicaciones Web

basadas en XML/HTTP. Hasta el momento tales aplicaciones han sido descritas principalmente por medio de combinaciones de descripciones textuales y esquemas XML. WADL pretende proporcionar una descripción de estas aplicaciones procesable automáticamente.

- **Familiaridad:** REST requiere repensar el problema en términos de manipulación de recursos direccionables en vez de llamadas a métodos. Evidentemente, la parte del servidor puede implementarse como se quiera, pero la interfaz con los clientes debe hacerse en términos de REST.

  Los clientes podrían preferir una interfaz basada en componente a una interfaz REST. Así como los programadores que están más familiarizados con el uso de APIs. Además, las APIs se integran mucho mejor en los lenguajes de programación existentes. Para los programadores por el lado del cliente, REST puede resultar algo novedoso, en cambio para el otro lado, no se notara mucha diferencia con lo que se había desarrollado en los últimos años, sitios Webs

1.12. ¿Qué puede pasar con SOAP en el futuro?

El problema de la escalabilidad ya ha sido analizado a lo largo de este documento. Resulta muy fácil de implantar un protocolo dentro de una compañía, pero cuando hablamos de entornos que atraviesan estas fronteras (miles de sistemas), no existe la oportunidad de actualización uno a uno.

Por tanto, las nuevas aplicaciones basadas en SOAP tendrán un gran obstáculo a superar antes de ser implantadas y tendrán incluso mayores retos adaptando y evolucionando una vez hayan sido implantadas.

Por otra parte, posiblemente tendrá varios años de éxito dentro de las organizaciones. Muchos profesionales ven a SOAP como una versión estandarizada de DCOM y CORBA, y por tanto, tendrá al menos el mismo éxito que tuvieron estas tecnologías en la integración punto a punto de sistemas internos. Sin embargo, si una alternativa basada en REST llega a ser dominante en Internet, será inevitable su filtración a los sistemas corporativos internos como hizo la Web.

1.13. ¿Qué pasará con REST?

Los negocios electrónicos van a necesitar algo más que tecnologías orientadas en RPC. Todos los negocios de cualquier lugar tendrán que estandarizar sus modelos de direccionamiento para exponer las interfaces en común a sus socios. SOAP no permite esto en sí mismo, incluso confunde más que aclara. Para que los negocios interoperen sin programar manualmente de manera explícita enlaces a los socios, se necesitará estandarizar un modelo de direccionamiento, más que invertir en sistemas propietarios. REST proporciona un alto grado de estandarización. Por tanto, si los servicios Web basados en SOAP no consiguen implantar este mecanismo, no sobrevivirán y, por tanto, surgirá la era de los Servicios Web basados en REST.

1.14. ¿Dónde es útil REST?

Tanto los arquitectos como los desarrolladores necesitan decidir cuál es el estilo adecuado para las aplicaciones. En algunos casos es adecuado un diseño basado en REST, se listan a continuación:

- El servicio Web no tiene estado. Una buena comprobación de esto consistiría en considerar si la interacción puede sobrevivir a un reinicio del servidor.

- Una infraestructura de *caching* puede mejorar el rendimiento. Si los datos que el servicio Web devuelve no son dinámicamente generados y pueden ser cacheados, entonces la infraestructura de *caching* que los servidores Web y los intermediarios proporcionan, pueden incrementar el rendimiento.

- Tanto el productor como el consumidor del servicio conocen el contexto y contenido que va a ser comunicado. Ya que REST no posee todavía (aunque hayamos visto una propuesta interesante) un modo estándar y formal de describir la interfaz de los servicios Web, ambas partes deben estar de acuerdo en el modo de intercambiar de información.

- El ancho de banda es importante y necesita ser limitado. REST es particularmente útil en dispositivos con escasos recursos como PDAs o teléfonos móviles, donde la sobrecarga de las cabeceras y capas adicionales de los elementos SOAP debe ser restringida.

- La distribución de Servicios Web o la agregación con sitios Web existentes puede ser fácilmente desarrollada mediante REST. Los desarrolladores pueden utilizar tecnologías como AJAX y toolkits como DWR (Direct Web Remoting) para consumir el servicio en sus aplicaciones Web.

1.15. ¿Dónde es útil SOAP?

Un diseño basado en SOAP es adecuado cuando:

- Se establece un contrato formal para la descripción de la interfaz que el servicio ofrece. El lenguaje de Descripción de Servicios Web (WSDL), como ya sabemos, permite describir con detalles el servicio Web.

- La arquitectura debe abordar requerimientos complejos no funcionales. Muchas especificaciones de servicios Web abordan tales requisitos y establecen un vocabulario común para ellos. Algunos ejemplos incluyen transacciones, seguridad, direccionamiento, … La mayoría de aplicaciones del mundo real se comportan por encima de las operaciones CRUD y requieren mantener información contextual y el estado conversacional. Con la aproximación REST, abordar este tipo de arquitecturas resulta más complicado.

- La arquitectura necesita manejar procesado asíncrono e invocación. En estos casos, la infraestructura proporcionada por estándares como WSRM y APIs como JAX-WS junto con la asincronía por el lado del cliente nos permitirán el soporte de estas características.

1.16. ¿Qué podemos concluir de todo este debate?

Amazon posee ambos estilos de uso de sus servicios Web. Pero el 85% de sus clientes prefieren la interfaz REST. A pesar de la promoción que las empresas han invertido para ensalzar a SOAP, parece

que es evidente que los desarrolladores prefieren, en algunos casos, la aproximación más sencilla: REST.

Todo lo visto a lo largo del documento como en el párrafo anterior, parece que nos da pistas para el futuro éxito de REST. Aunque, todos sabemos que en el mundo de la tecnología no siempre acaba triunfando la tecnología mejor, recuérdese el caso de VHS vs BetaMax.

Lo que está claro es que falta una pieza en la Web. La comunicación Hombre – Máquina parece que funciona, pero la comunicación Máquina – Máquina sigue siendo un reto. Recientemente, se ha presentado una propuesta donde los principios de REST han sido aplicados a los estándar y guías de diseño asociadas con la nueva versión de SOAP, es decir, SOAP podría ser utilizado de tal manera que no violara los principios de REST. Esto parece prometedor. Pero en mi opinión, este tipo de propuestas (incluyendo a REST) no triunfarán si la industria no apuesta realmente por ellas (herramientas y frameworks).

Resumiendo:

#	SOAP	REST
1	Protocolo de mensajes XML	Protocolo de estilo arquitectural
2	Usa WSDL en la comunicación entre el consumidor y el proveedor	Usa XML o JSON para enviar y recibir datos
3	Invoca a los servicios mediante llamadas a métodos RPC	Llamada a un servicio vía URL
4	Información que devuelve no legible para el humano	Resultado es legible por el humano (XML, JSON…)
5	Transferencia sobre HTTP y otros protocolos (SMTP, FTP, etc…)	Transferencia es sólo sobre HTTP
6	JavaScript permite invocar SOAP (implementación compleja)	Fácil de invocar desde JavaScript
7	El rendimiento no es tan bueno comparado a REST	Rendimiento mucho mejor que SOAP – menor consumo CPU, código más pulido, etc.

No obstante, la decisión sobre implementar una tecnología u otra depende en gran medida de las características del proyecto en el que estemos implicados, por lo que es recomendable siempre hacer un análisis concienzudo del proyecto y las tecnologías disponibles para decantarnos por una o por otra. En el siguiente capítulo se describen más detalles sobre REST que te pueden ayudar a elegir esta metodología por encima de otras.

2. Servicios Web RESTful

2.1. ¿Qué es la Arquitectura REST?

REST (*Representational State Transfer*) es un estilo de arquitectura de software basada en estándares web y en el protocolo HTTP cuya finalidad es desarrollar servicios web. En REST cada componente es un recurso y para acceder a éste se hace a través de una interfaz común utilizando operaciones estándar HTTP. REST fue introducido por primera vez por Roy Fielding en el año 2000.

En la arquitectura REST, un servidor REST simplemente proporciona acceso a los recursos y el cliente REST accede y presenta los recursos, identificados mediante IDs globales (normalmente URIs). REST permite usar diferentes tipos de formato para representar un recurso: Texto, JSON and XML. JSON es el más extendido.

2.2. Principios de la arquitectura REST

- **Recursos direccionables:** La abstracción de la información y los datos en REST es un recurso, y cada recurso debe ser direccionable via URI (*Uniform Resource Identifier*).

- **Interfaz uniforme y limitada:** REST usa un conjunto reducido y bien definido de comandos/métodos para gestionar los recursos.

- **Orientada a la representación:** La interacción con los servicios se realiza a través de representaciones de dichos servicios.

- **Comunicación sin estado:** esto significa que nuestro servidor no tiene porqué almacenar datos del cliente para mantener un estado del mismo. Esta limitación es sujeto de mucho debate en la industria, incluso ya empiezan a usarse tecnologías relacionadas que implementan el estado dentro de la arquitectura, como WebSockets. Como sabemos, HTTP también cumple esta norma, por lo que estamos acostumbrados ya a hacer uso de protocolos stateless. Las aplicaciones sin estado son más fáciles de mantener y extender.

- **Hypermedia As The Engine Of Application State (HATEOAS):** Deje que el formato de los datos generen transiciones de estado en sus aplicaciones.

2.3. REST y el renacimiento de HTTP

REST no es un protocolo ni un estándar específico, pero cuando la gente habla de REST, normalmente se refieren a REST sobre HTTP.

Aprender REST implica el redescubrimiento del protocolo HTTP así como aprender un nuevo estilo de desarrollo de aplicaciones distribuidas. Las aplicaciones web basadas en un navegador sólo ven una mínima parte de las características de HTTP. Por otro lado, las tecnologías que no son RESTful como SOAP and WS- *usan HTTP estrictamente como protocolo de transporte y por ello solo utilizan un conjunto muy reducido de su funcionalidad. Muchos dirían que SOAP y WS*- se usan tan solo para

atravesar redes (*tunneling*) mediante cortafuegos. HTTP es realmente un protocolo de aplicación muy rico que provee multitud de funcionalidades interesantes y útiles para los desarrolladores de software. Para crear servicios web RESTful es necesario comprender y tener una buena base del protocolo HTTP.

HTTP es un protocolo de red síncrono perteneciente a la capa de aplicación basado en el modelo petición/respuesta (request/response). Se utiliza en sistemas distribuidos y colaborativos basados en documentos. HTTP es el principal protocolo de Internet y particularmente utilizado por los navegadores más extendidos (Firefox, MS Internet Explorer, Safari, …). Se trata de un protocolo muy sencillo: el cliente envia un mensaje de petición formado por el método HTTP que lo invocó, la localización del recurso requerido, un conjunto variable de cabeceras (*headers*) y un cuerpo de mensaje opcional que basicamente puede ser cualquier cosa que queramos, incluyendo HTML, texto plano, XML, JSON, e incluso datos binarios. Ejemplo:

```
GET /resteasy HTTP/1.1
Host: jboss.org
User-Agent: Mozilla/5.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language:      en-us,en;q=0.5
Accept-Encoding:     gzip,deflate
```

Nuestro navegador enviaría la petición (*request*) anterior para acceder a http://jboss.org/resteasy:

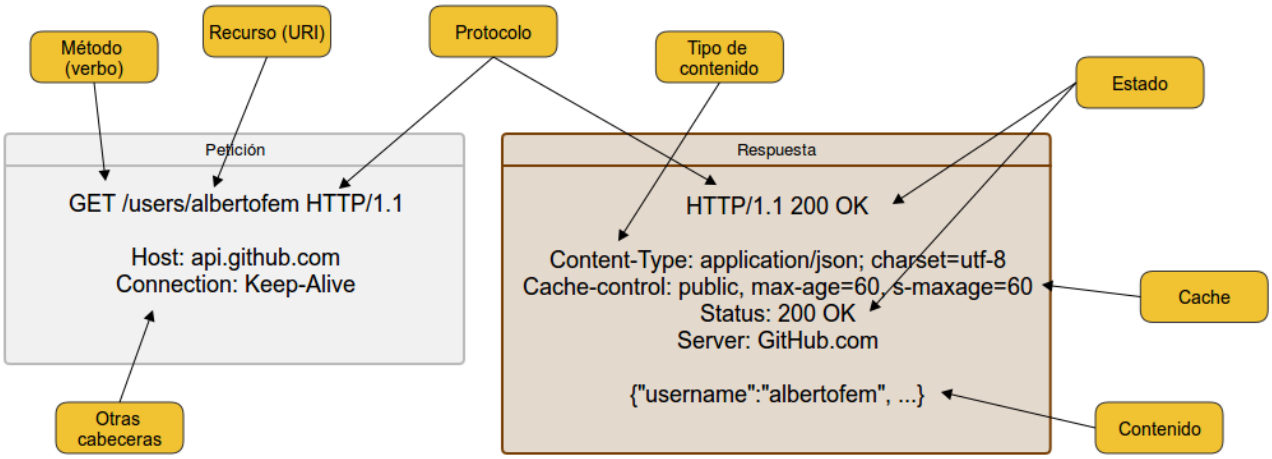
- GET es el método que se invoca en el servidor.
- /resteasy es el objeto al que queremos acceder.
- HTTP/1.1 es la versión del protocolo.
- Host, User-Agent, Accept, Accept-Language y Accept-Encoding son las cabeceras del mensaje.
- En este ejemplo no hay cuerpo de mensaje (request body), ya que se está consultando información desde el servidor.

El mensaje de respuesta del servidor (*response*) es muy similar. Este contiene la versión de HTTP que estamos usando, un código de respuesta, un mensaje corto que describe el código de respuesta, un conjunto variado de cabeceras opcionales, y un cuerpo de mensaje opcional. A continuación se muestra el mensaje de respuesta que el servidor podría enviar para la petición anterior (GET query):

```
HTTP/1.1 200 OK
X-Powered-By: Servlet 2.4; JBoss-4.2.2.GA
Content-Type: text/html

<head>
<title>JBoss RESTEasy Project</title>
</head>
<body>
<h1>JBoss RESTEasy</h1>
<p>JBoss RESTEasy is an open source implementation of the JAX-RS
specification...
```

Como se puede observar, el código de respuesta del mensaje es 200, y el estado del mensaje es “OK”. Este código indica que la petición fue procesada con éxito y que el cliente recibirá la información o el recurso que solicitó. HTTP incluye un amplio conjunto de códigos de respuesta. Estos pueden ser códigos meramente informativos como por ejemplo 200, “OK”, o códigos de error como 500, “Internal Server Error”. Puede consultar el listado completo y detallado de estos códigos en el sitio web oficial de W3C.



2.4. Métodos HTTP

Los cinco métodos HTTP que se utilizan comúnmente en la arquitectura basada en REST son:

- **GET** - Proporciona un acceso de sólo lectura a un recurso.
- **PUT** - Se utiliza para crear un nuevo recurso.
- **DELETE** - Se utiliza para eliminar un recurso.
- **POST** - Se utiliza para actualizar un recurso existente o crear un nuevo recurso.
- **OPTIONS** - Se utiliza para obtener las operaciones apoyadas en un recurso.

2.5. Introducción a los servicios web RESTful

Un servicio web es un conjunto de protocolos y estándares utilizados para el intercambio de datos entre aplicaciones o sistemas. Las aplicaciones de software escritas en diversos lenguajes de programación y que se ejecutan en diferentes plataformas pueden utilizar los servicios web para intercambiar datos a través de redes informáticas como Internet de una manera similar a la comunicación entre procesos en un único equipo. Esta interoperabilidad (por ejemplo entre Java y Python, o Windows y Linux) se debe a la utilización de estándares abiertos.

Los servicios web basados en arquitecturas REST se conocen como servicios web **RESTful.** Estos servicios web utilizan métodos HTTP aplicando así el concepto de arquitectura REST. Un servicio web

RESTful por lo general define una URI *(Uniform Resource Identifier)*, un servicio que ofrece representación de recursos tales como JSON y un conjunto de métodos HTTP.

2.6. Creación de servicios Web REST

La siguiente tabla describe un servicio web de gestión de usuarios con las funcionalidades que se indican en la columna *Operación*:

Método HTTP	URI	Operación	Tipo de operación
GET	/UserService/users	Obtener todos los usuarios	Read Only
GET	/UserService/users/1	Obtener el usuario con Id 1	Read Only
PUT	/UserService/users/2	Insertar usuario con Id 2	Idempotent
POST	/UserService/users/2	Actualizar usuario con Id 2	N/A
DELETE	/UserService/users/1	Borrar usuario con Id 1	Idempotent
OPTIONS	/UserService/users	Lista de operaciones disponibles en el web service	Read Only

3. Configuración del entorno

Este manual le guiará para preparar un entorno de desarrollo para comenzar un proyecto usando el framework Jersey para crear servicios web REST. El framework Jersey implementa JAX-RS 2.0 API que es la especificación estándar para crear servicios web RESTful. Este manual también le enseñará cómo configurar el JDK, Tomcat y Eclipse en su máquina antes de configurar Jersey.

1 - Configuración de Java Development Kit (JDK)

Puede descargar la última versión del SDK de Java desde el sitio de Oracle: Java SE Descargas. Encontrará las instrucciones para la instalación de JDK en los archivos descargados, siga las instrucciones para instalar y configurar la configuración. Finalmente establecer variables de entorno JAVA_HOME PATH que se refieren al directorio que contiene java y javac, típicamente *java_install_dir / bin* y *java_install_dir* respectivamente.

Si está ejecutando Windows e instalado el JDK en C: \ jdk1.7.0_75, tendría que poner la siguiente línea en el archivo C: \ autoexec.bat.

```
set PATH=C:\jdk1.7.0_75\bin;%PATH%
set JAVA_HOME=C:\jdk1.7.0_75
```

Como alternativa, en Windows NT / 2000 / XP, podría también hacer clic derecho en Mi PC, seleccione Propiedades, luego Avanzado, luego variables de entorno. A continuación, le actualizar el valor de PATH y pulse el botón OK.

En Unix (Solaris, Linux, etc.), si el SDK está instalado en /usr/local/jdk1.7.0_75 y utiliza el shell C, que pondría lo siguiente en su archivo .cshrc.

```
setenv PATH /usr/local/jdk1.7.0_75/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.7.0_75
```

Alternativamente, si se utiliza un entorno de desarrollo integrado (IDE) como Borland JBuilder, Eclipse, IntelliJ IDEA, o Sun ONE Studio, para compilar y ejecutar un programa sencillo deberá confirmar que el IDE sabe dónde se ha instalado Java, de lo contrario deberá hacer una configuración adecuada del IDE.

2 - Configuración Eclipse IDE

Todos los ejemplos de este manual se han escrito utilizando el IDE Eclipse. Así que debería tener la última versión de Eclipse instalado en su máquina.

Para instalar Eclipse IDE, descargar los últimos binarios de Eclipse de http://www.eclipse.org/downloads/ . Una vez que ha descargado la instalación, expandir la distribución binaria en una ubicación conveniente. Por ejemplo, en C: \ eclipse en las ventanas, o / usr / local / Eclipse en Linux / Unix y, finalmente, establecer la variable PATH de manera apropiada.

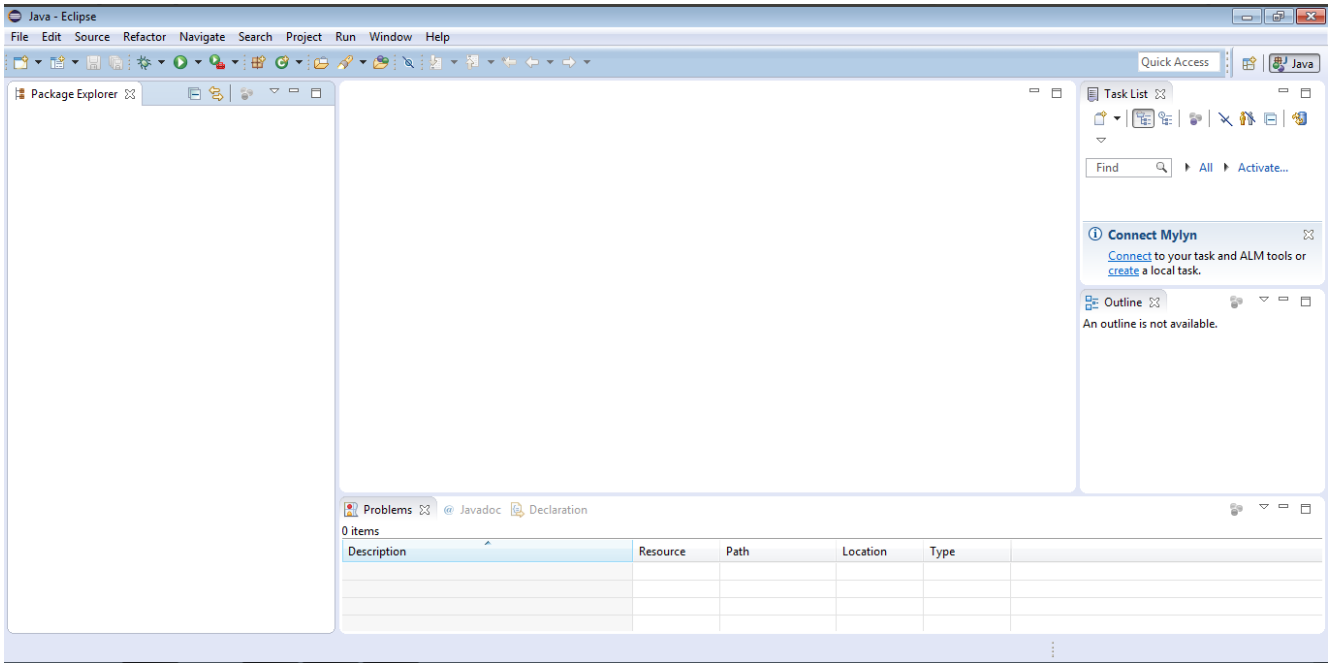
Eclipse se puede iniciar mediante la ejecución de los siguientes comandos desde la consola, o puede simplemente haga doble clic en *eclipse.exe*

```
%C:\eclipse\eclipse.exe
```

Eclipse se puede iniciar mediante la ejecución de los siguientes comandos en Unix (Solaris, Linux, etc.) de la máquina:

```
$/usr/local/eclipse/eclipse
```

Si todo ha ido bien, debería aparecer la siguiente vista del entorno de trabajo de Eclipse:



3 - Configuración de las librerías del framework Jersey

Si todo funciona bien, se puede proceder a la configuración de su marco de trabajo **Jersey**. Los siguientes son los pasos para descargar e instalar el framework en su máquina.

1. Elegir si desea instalar Jersey en Windows o Unix y luego proceder al siguiente paso para descargar el archivo .zip para Windows y el archivo .tz para Unix.
2. Descargar la última versión de los binarios del framework Jersey https://jersey.java.net/download.html .
3. En el momento de escribir este manual, se ha descargado jaxrs-ri-2.17.zip en la máquina Windows y al descomprimir el archivo descargado que creará la estructura de directorios en E: jaxrs-ri-2,17 \ jaxrs-ri \ como se muestra en la imagen:

Name	Size	Type	Date Modified
api		File Folder	3/11/2015 1:49 PM
ext		File Folder	3/11/2015 1:49 PM
lib		File Folder	3/11/2015 1:49 PM
Jersey-LICENSE.txt	36 KB	Text Document	3/11/2015 1:39 PM
third-party-license-readme.txt	23 KB	Text Document	3/11/2015 1:39 PM

Va a encontrar todas las bibliotecas Jersey en los directorios **C:\jaxrs-ri-2.17\jaxrs-ri\lib y dependencias en C:\jaxrs-ri-2.17\jaxrs-ri\ext.** Asegúrese de configurar la variable CLASSPATH en este directorio correctamente de lo contrario se enfrentará a problemas durante la ejecución de la aplicación. Si está utilizando Eclipse, entonces no se requiere establecer CLASSPATH ya que toda la configuración se realiza a través de Eclipse.

4 - Configuración de Apache Tomcat

Puede descargar la última versión de Tomcat desde http://tomcat.apache.org/. Una vez que ha descargado la instalación, expandir la distribución binaria en la ubicación deseada. Por ejemplo, en C: \ apache-tomcat-7.0.59 en las ventanas, o /usr/local/apache-tomcat-7.0.59 en Linux / Unix y establecer entorno CATALINA_HOME variable apuntando a la ruta de instalación.

Tomcat puede iniciarse mediante la ejecución de los siguientes comandos en la consola de comandos, o puede simplemente hacer doble clic en *startup.bat*

```
%CATALINA_HOME%\bin\startup.bat
```

o

```
C:\apache-tomcat-7.0.59\bin\startup.bat
```

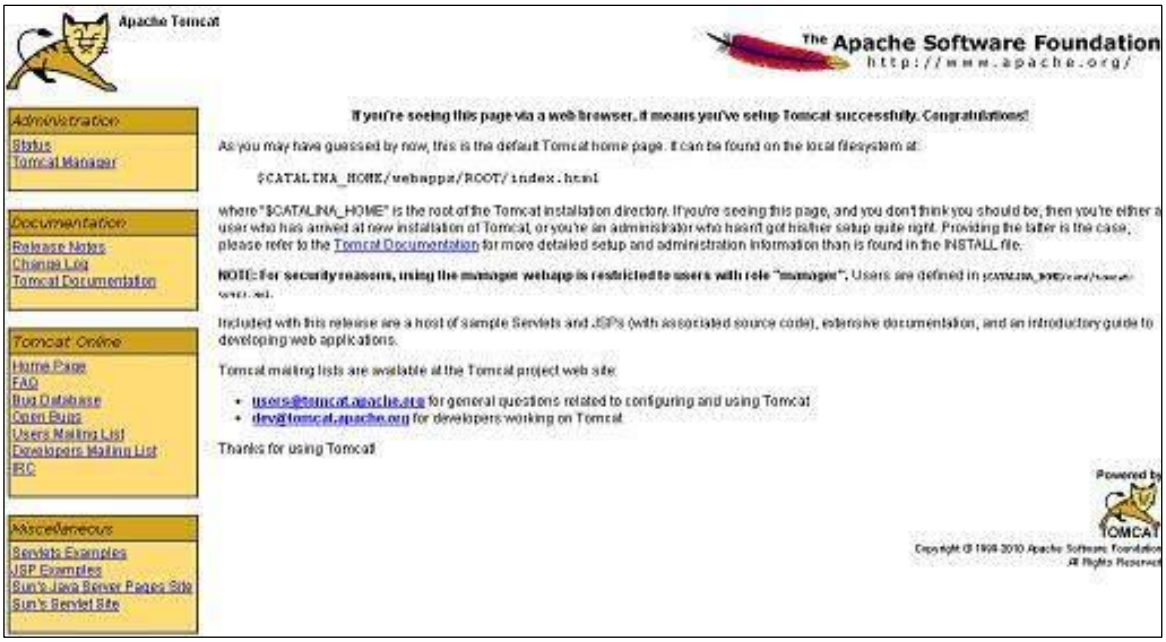
Tomcat puede iniciarse mediante la ejecución de los siguientes comandos en Unix (Solaris, Linux, etc.) de la máquina:

```
$CATALINA_HOME/bin/startup.sh
```

o

```
/usr/local/apache-tomcat-7.0.59/bin/startup.sh
```

Después de una exitosa puesta en marcha, las aplicaciones web predeterminadas incluidas en Tomcat estarán disponibles visitando **http://localhost:8080/.** Si todo ha ido bien, entonces debería mostrar lo siguiente:



Más información sobre la configuración y ejecución de Tomcat se pueden encontrar en la documentación que se incluye aquí, así como en el sitio Web de Tomcat: http://tomcat.apache.org

Tomcat se puede detener mediante la ejecución de los siguientes comandos desde la consola:

```
%CATALINA_HOME%\bin\shutdown
```

o

```
C:\apache-tomcat-7.0.59\bin\shutdown
```

Tomcat se puede detener mediante la ejecución de los siguientes comandos en Unix (Solaris, Linux, etc.) de la máquina:

```
$CATALINA_HOME/bin/shutdown.sh
```

o

```
/usr/local/apache-tomcat-7.0.59/bin/shutdown.sh
```

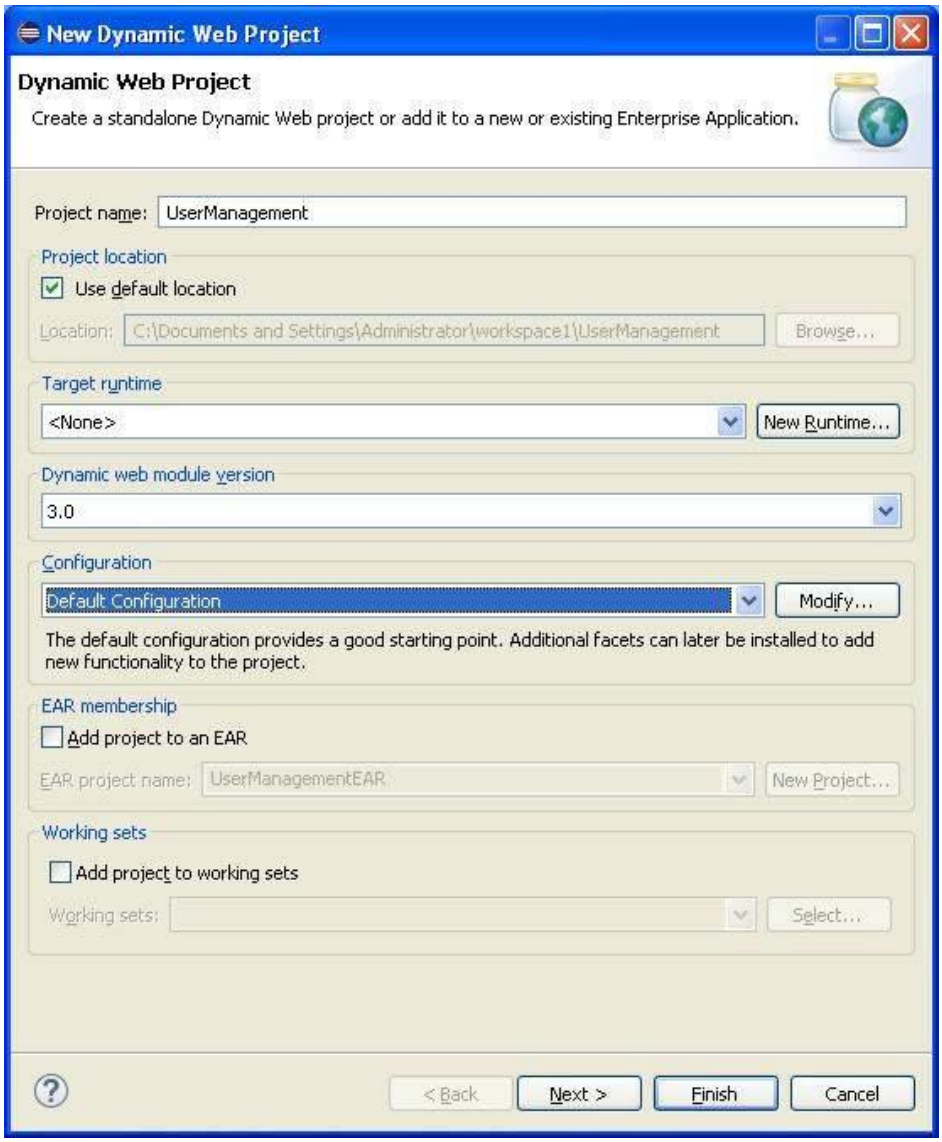
Una vez que haya terminado con este último paso, ya estará listo para crear su primera aplicación Jersey, que se verá en el siguiente apartado.

4. Primera aplicación

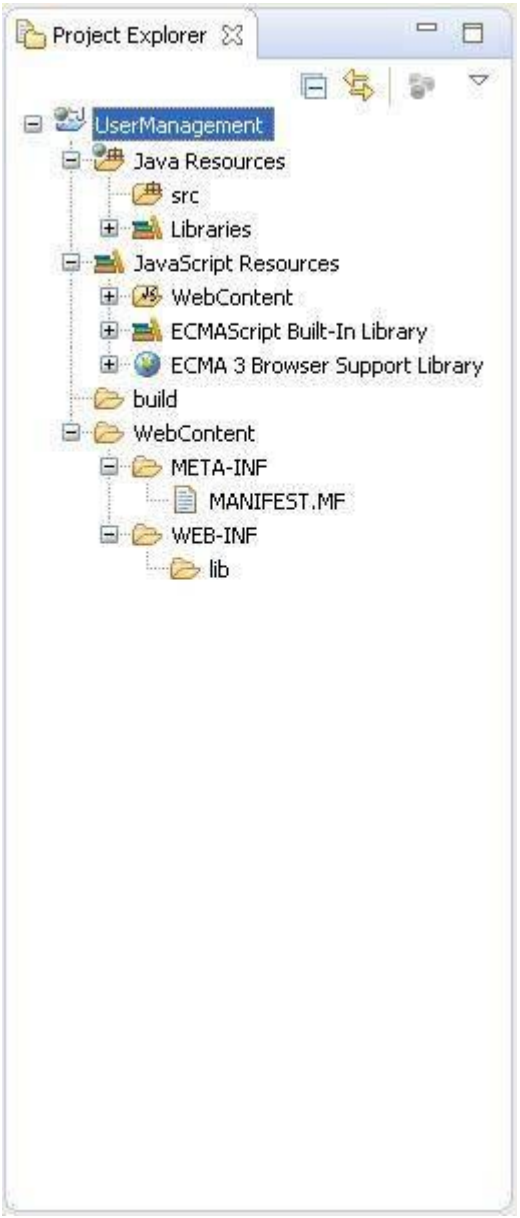
Vamos a empezar a escribir servicios web RESTful reales con Jersey. Antes de empezar a escribir su primer ejemplo usando Jersey, tiene que asegurarse de que ha configurado su entorno Jersey adecuadamente. Por lo tanto vamos a proceder a escribir una aplicación sencilla Jersey que expondrá un método de servicio web a la lista de usuarios a visualizar.

1 - Crear un proyecto Java

El primer paso es crear un proyecto Web dinámico usando Eclipse IDE. Seleccione la opción **File -> New -> Project** y finalmente seleccione **Dynamic Web Project** asistente de la lista. Ahora el nombre de su proyecto como *UserManagement* usando la ventana del asistente de la siguiente manera:



Una vez que se ha creado correctamente su proyecto, tendrá la siguiente estructura de directorios en su **Project Explorer:**



2 - Añadir librerías necesarias

Como segundo paso añadiremos Jersey y sus dependencias (*libraries*) en nuestro proyecto. Copiar todos los .JAR de la carpeta zip de descarga de Jersey en el directorio WEB-INF / lib del proyecto.

- \ Jaxrs-ri-2.17 \ jaxrs-ri \ api
- \ Jaxrs-ri-2.17 \ jaxrs-ri \ ext
- \ Jaxrs-ri-2.17 \ jaxrs-ri \ lib

Ahora, haga clic derecho sobre el nombre del proyecto **UserManagement** y seleccione la siguiente opción disponible en el menú contextual: **Build Path -> Configure Build Path** para visualizar la ventana de Java Build Path.

Ahora usa **Add JARs** debajo de **Libraries** para añadir los archivos **JAR** presentes en **WEB-INF / lib.**

3 - Crear los archivos fuente

Ahora vamos a crear los archivos fuente del proyecto **UserManagement.** En primer lugar tenemos que crear un paquete llamado **com.dominio-organizacion.** Para ello, haga clic en **src** en la sección explorador de paquetes y elija la opción: **New -> Package.**

A continuación vamos a crear las clases **UserService.java , User.java , UserDao.java** en el paquete *com.dominio-organizacion*.

User.java

```
package com.dominio-organizacion;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name = "user")
public class User implements Serializable {

   private static final long serialVersionUID = 1L;
   private int id;
   private String name;
   private String profession;

   public User() {}

   public User(int id, String name, String profession) {
      this.id = id;
      this.name = name;
      this.profession = profession;
   }

   public int getId() {
      return id;
   }

   @XmlElement
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   @XmlElement
   public void setName(String name) {
      this.name = name;
   }
   public String getProfession() {
      return profession;
   }
   @XmlElement
   public void setProfession(String profession) {
      this.profession = profession;
   }
}
```

UserDao.java

```
package com.dominio-organizacion;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class UserDao {
   public List<User> getAllUsers() {
      List<User> userList = null;
      try {
         File file = new File("Users.dat") ;
         if (!file.exists() ) {
            User user = new User(1, "Mahesh" , "Teacher") ;
            userList = new ArrayList<User>() ;
            userList.add(user) ;
            saveUserList(userList) ;
         }
         else{
            FileInputStream fis = new FileInputStream(file) ;
            ObjectInputStream ois = new ObjectInputStream(fis) ;
            userList = (List<User>) ois.readObject() ;
            ois.close() ;
         }
      } catch (IOException e) {
         e.printStackTrace() ;
      } catch (ClassNotFoundException e) {
         e.printStackTrace() ;
      }
      return userList;
   }
}
```

UserService.java

```
package com.dominio-organizacion;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/UserService")
public class UserService {

   UserDao userDao = new UserDao() ;

   @GET
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)
   public List<User> getUsers() {
      return userDao.getAllUsers() ;
```

```
    }
}
```

Debe tener en cuenta dos cuestiones importantes en el programa principal, *UserService.java*:

1. El primer paso es especificar una ruta para el servicio web utilizando la anotación @Path en *UserService*.

2. El segundo paso consiste en especificar una ruta para el método de servicio web, en particular mediante la anotación @Path en el método de *UserService*.

4 - Crear archivo de configuración web.xml

Es necesario crear un archivo de configuración web XML, que se utiliza para especificar el servlet Jersey para nuestra aplicación.

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
   <display-name>User Management</display-name>
   <servlet>
      <servlet-name>Jersey RESTful Application</servlet-name>
      <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
         <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>com.dominio-organizacion</param-value>
         </init-param>
      </servlet>
   <servlet-mapping>
   <servlet-name>Jersey RESTful Application</servlet-name>
      <url-pattern>/rest/*</url-pattern>
   </servlet-mapping>
</web-app>
```

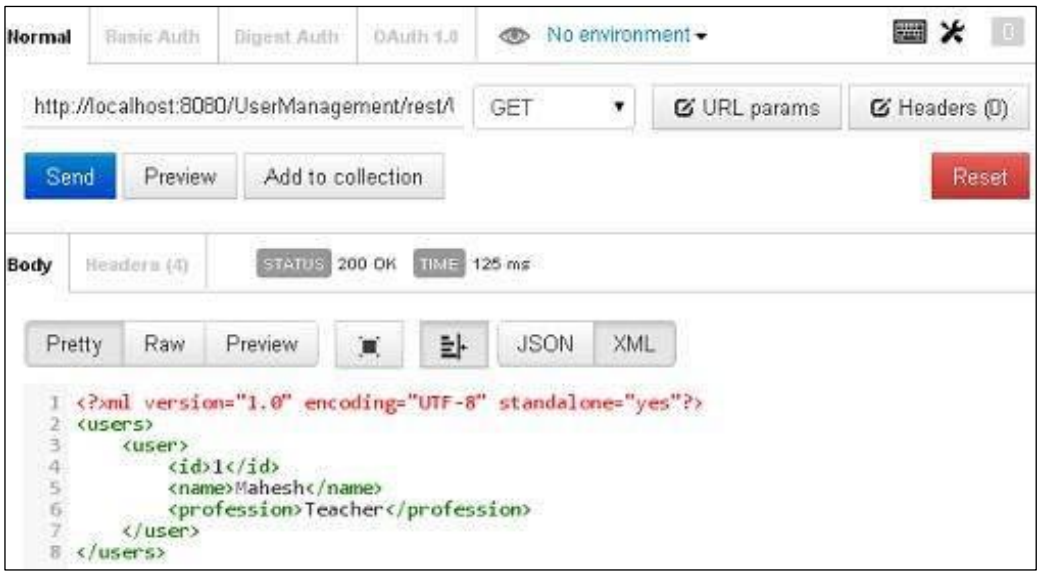
5 - Despliegue del programa

Una vez que haya terminado con la creación de los archivos de configuración web y fuente, estará listo para la compilación y ejecución del programa. Para ello, haga uso de Eclipse, exportar su aplicación como un archivo .WAR y desplegar los mismos en Tomcat. Para crear el archivo WAR utilizando Eclipse, seleccione la opción **File -> export -> Web > War File** carpeta de destino y, finalmente, seleccione el proyecto **UserManagemeNT.** Para implementar el archivo WAR en Tomcat, guarde el archivo *UserManagement.war* en **Tomcat Installation Directory > directorio de aplicaciones web** e iniciar Tomcat.

6 - Ejecución del programa

En este manual hemos utilizado Postman, una extensión de Chrome, para probar nuestros servicios web. Postman ofrece una plataforma GUI para hacer el desarrollo de APIs más rápido y fácil incluyendo la creación de peticiones API a través de testing y documentación.

Probemos una petición (Request) a *UserManagemeNT* y obtener la lista de todos los usuarios. Escriba `http://localhost:8080/UserManagemeNT/rest/UserService/usuarios` en Postman con petición GET y ver el resultado a continuación.



Enhorabuena, hemos creado nuestra primera aplicación RESTful con éxito. Se puede ver la flexibilidad de la aplicación anterior, cambiando el valor de la propiedad *message* y manteniendo los archivos fuente sin cambios. Más adelante haremos algo más interesante.

5. Recursos

5.1. ¿Qué es un recurso?

En la arquitectura REST, todo es un recurso. Estos recursos pueden ser archivos de texto, páginas HTML, imágenes, videos o datos empresariales dinámicos. REST servidor simplemente proporciona acceso a los recursos y el cliente accede a REST y modifica los recursos. Aquí cada recurso es identificado por una ID global (URI). REST utiliza varias tipos de representaciones para especificar un recurso como texto, JSON, XML, siendo estas dos últimas las representaciones más extendidas.

5.2. Representación de Recursos

Un recurso REST es un objeto similar al de la programación orientada a objetos o incluso a la entidad en una base de datos. Una vez que se identifica un recurso entonces su representación no utiliza un formato estándar, de modo que el servidor puede enviar el recurso encima de dicho formato y de esta forma el cliente puede entender el mismo formato.

Por ejemplo, en el siguiente ejemplo, un usuario es un recurso que se representa mediante el formato XML:

```
<user>
   <id>1</id>
   <name>Mahesh</name>
   <profession>Teacher</profession>
</user>
```

El mismo recurso puede representarse en formato JSON:

```
{
   "id":1,
   "name":"Mahesh",
   "profession":"Teacher"
}
```

5.3. Características de una buena representación

En REST, no hay ninguna restricción respecto al formato de la representación de recursos. Un cliente puede pedir la representación JSON en tanto que otro cliente puede solicitar representación XML de un mismo recurso al servidor y así sucesivamente. Es responsabilidad del servidor REST pasar al cliente el recurso en el formato que entiende cliente. Los siguientes, son cuestiones importantes a tener en cuenta a la hora de diseñar un formato de representación de un recurso para en los servicios web RESTful.

- **Understandability:** el servidor y el cliente deben ser capaces de entender y utilizar el formato de representación del recurso.
- **Completeness:** el formato debe ser capaz de representar el recurso completo. Por ejemplo, un recurso puede contener otro recurso. El formato debe ser capaz de representar tanto estructuras de recursos simples como complejas.

- **Linkability:** un recurso puede tener un enlace a otro recurso, el formato debe ser capaz de manejar este tipo de situaciones.

6. Mensajes

Los servicios web RESTful hacen uso del protocolo HTTP como medio de comunicación entre el cliente y el servidor. Un cliente envía un mensaje en forma de una petición HTTP y el servidor responde en forma de una respuesta HTTP. Esta técnica es similar al sistema de mensajería. Estos mensajes contienen datos de los mensajes y metadatos que contiene información sobre el mensaje en sí. Vamos a echar un vistazo a una petición y una respuesta HTTP para mensajes de la versión HTTP 1.1.

6.1 Petición (Request) HTTP

HTTP Request

Una petición (Request) HTTP consta de cinco partes principales:

- **Verb**: Indica los métodos HTTP como GET, POST, etc.
- **URI**: Contiene el URI, Uniform Resource Identifier para identificar el recurso en el servidor
- **HTTP Version**: Indica la versión de HTTP, por ejemplo HTTP v1.1.
- **Request Header**: Contiene metadatos para el mensaje de petición HTTP como pares de valor y clave. Por ejemplo, tipo de cliente (o *browser*), formato soportado por el cliente, el formato del cuerpo del mensaje, la configuración de caché, etc.
- **Request Body**: Contenido del mensaje o representación del recurso.

6.2. Respuesta (Response) HTTP

HTTP Response

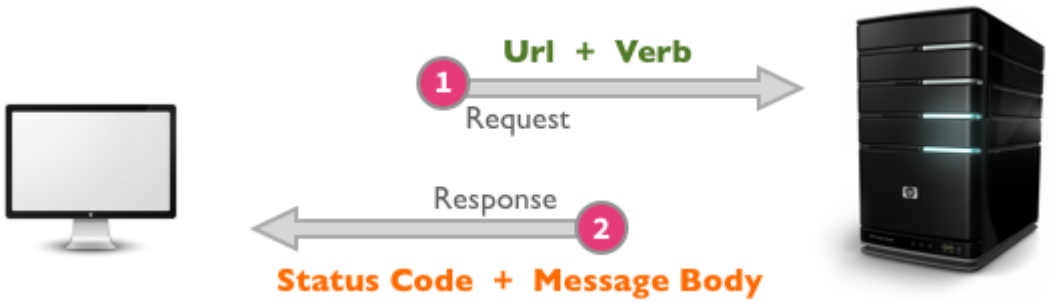
Una respuesta HTTP consta de cuatro partes principales:

- **Status/Response Code**: Indica el estado del servidor para el recurso solicitado. Por ejemplo 404 significa *Recurso no encontrado* y 200 indica que la respuesta es correcta.
- **HTTP Version**: Indica la versión de HTTP, por ejemplo HTTP v1.1.
- **Response Header:** Contiene metadatos para el mensaje de respuesta HTTP como pares de valores clave. Por ejemplo, longitud del contenido, tipo de contenido, fecha de respuesta, el tipo de servidor etc.
- **Response Body**: Contenido del mensaje de respuesta o la representación del recurso.

Ejemplo:

Escriba `http://localhost:8080/UserManagement/rest/UserService/users` en Postman con petición GET. Si hace clic en el botón de vista previa que está cerca del botón de envío de Postman y luego hace clic en Enviar, debería aparecer algo similar a lo siguiente:

Se puede ver arriba, cómo el navegador envía una petición (Request) GET y después recibe el cuerpo de la respuesta como XML.



7. Direccionamiento

El direccionamiento (*addressing*) se refiere a la localización de uno o múltiples recursos situados en el servidor. Es análogo a localizar una dirección postal de una persona.

Cada recurso en la arquitectura REST se identifica por su URI (*Uniform Resource Identifier*). Una URI tiene el siguiente formato:

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

El propósito de una URI es localizar un recurso(s) en el servidor que aloja el servicio web. Otro atributo importante de una petición es el comando que identifica la operación que se realiza en el recurso. Por ejemplo, la URI **http://localhost:8080/UserManagement/rest/UserService/users** y el comando **GET.**

7.1. La construcción de una URI estándar

Las siguientes, son convenciones y buenas prácticas a la hora de diseñar una URI:

- **Usar sustantivo en plural**: utilizar sustantivo plural para definir recursos. Por ejemplo, hemos usado *users* para identificar a los usuarios como un recurso.
- **Evitar usar espacios**: Usar guión bajo (_) o guión (-) cuando se utiliza un nombre de recurso largo, por ejemplo, usar *authorized_users* en lugar de *authorized%20users*.
- **Usar letras en minúsculas:** Aunque la URI es sensible a las mayúsculas, es una buena práctica escribir las URL sólo con letras minúsculas.
- **Mantener la compatibilidad hacia detrás:** Como el servicio web es un servicio público, una vez que la URI se ha hecho pública, debe estar siempre disponible. En el caso, de que la URI se actualice, redirigir la URI antigua a la nueva URI utilizando el código de estado HTTP, 300.
- **Usar comandos HTTP:** Siempre utilice comandos HTTP como GET, PUT y DELETE para realizar las operaciones necesarias en el recurso. No es bueno utilizar nombres de operaciones en la URI. Por ejemplo, la URI siguiente es pobre para buscar un usuario:

  ```
  http://localhost:8080/UserManagement/rest/UserService/getUser/1
  ```

  A continuación se presenta un ejemplo de URI apropiada para buscar un usuario:

  ```
  http://localhost:8080/UserManagement/rest/UserService/users/1
  ```

  Deje que sea el servidor el que decida la operación basándose en el comando HTTP.

8. Métodos

Como hemos visto hasta ahora un servicio web REST utiliza verbos (métodos) HTTP para determinar la operación que se lleva a cabo en el recurso especificado. La siguiente tabla muestra ejemplos típicos de comandos HTTP.

Método HTTP	URI	Operación	Tipo de operación
GET	http://localhost:8080/UserManagement/rest/UserService/users	Obtener lista de usuarios	Read Only
GET	http://localhost:8080/UserManagement/rest/UserService/users/1	Obtener usuario con Id 1	Read Only
PUT	http://localhost:8080/UserManagement/rest/UserService/users/2	Insertar usuario con Id 2	Idempotent
POST	http://localhost:8080/UserManagement/rest/UserService/users/2	Actualizar usuario con Id 2	N/A
DELETE	http://localhost:8080/UserManagement/rest/UserService/users/1	Borrar usuario con Id 1	Idempotent
OPTIONS	http://localhost:8080/UserManagement/rest/UserService/users	Listar operaciones disponibles en el servicio web	Read Only
HEAD	http://localhost:8080/UserManagement/rest/UserService/users	Devuelve solo la cabecera HTTP, no el Body.	Read Only

Debemos tener en cuenta que:

- Las operaciones GET son sólo de lectura y son seguros.
- PUT y DELETE son operaciones idempotentes lo cual significa que su resultado será siempre igual no importando cuántas veces se invocan estas operaciones.
- PUT y POST son similares diferenciándose sólo en el resultado, donde la operación PUT es idempotente y POST puede dar un resultado diferente.

Ejemplo:

Vamos a crear un servicio web que pueda realizar las operaciones CRUD (*Create, Read, Update, Delete*). Para simplificar, hemos utilizado un archivo de E/S para sustituir las operaciones de base de datos.

Actualizar las clases **UserService.java, User.java, UserDao.java** del paquete **com.dominio-organizacion.**

User.java

```
package com.dominio-organizacion;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name = "user")
public class User implements Serializable {

   private static final long serialVersionUID = 1L;
   private int id;
   private String name;
   private String profession;

   public User() {}

   public User(int id, String name, String profession) {
      this.id = id;
      this.name = name;
      this.profession = profession;
   }

   public int getId() {
      return id;
   }
   @XmlElement
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   @XmlElement
      public void setName(String name) {
      this.name = name;
   }
   public String getProfession() {
      return profession;
   }
   @XmlElement
   public void setProfession(String profession) {
      this.profession = profession;
   }

   @Override
   public boolean equals(Object object) {
      if(object == null) {
         return false;
      }else if(!(object instanceof User) ){
         return false;
      }else {
         User user = (User) object;
         if(id == user.getId()
            && name.equals(user.getName() )
            && profession.equals(user.getProfession() )
         ){
            return true;
         }
      }
      return false;
```

```
   }
}
```

UserDao.java

```
package com.dominio-organizacion;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class UserDao {
   public List<User> getAllUsers() {
      List<User> userList = null;
      try {
         File file = new File("Users.dat") ;
         if (!file.exists() ) {
            User user = new User(1, "Mahesh" , "Teacher") ;
            userList = new ArrayList<User>() ;
            userList.add(user) ;
            saveUserList(userList) ;
         }
         else{
            FileInputStream fis = new FileInputStream(file) ;
            ObjectInputStream ois = new ObjectInputStream(fis) ;
            userList = (List<User>) ois.readObject() ;
            ois.close() ;
         }
      } catch (IOException e) {
         e.printStackTrace() ;
      } catch (ClassNotFoundException e) {
         e.printStackTrace() ;
      }
      return userList;
   }

   public User getUser(int id) {
      List<User> users = getAllUsers() ;

      for(User user: users) {
         if(user.getId() == id){
            return user;
         }
      }
      return null;
   }

   public int addUser(User pUser) {
      List<User> userList = getAllUsers() ;
      boolean userExists = false;
      for(User user: userList) {
         if(user.getId() == pUser.getId() ){
            userExists = true;
            break;
         }
```

```
      }
      if(!userExists) {
         userList.add(pUser) ;
         saveUserList(userList) ;
         return 1;
      }
      return 0;
   }

   public int updateUser(User pUser) {
      List<User> userList = getAllUsers() ;

      for(User user: userList) {
         if(user.getId() == pUser.getId() ){
            int index = userList.indexOf(user) ;
            userList.set(index, pUser) ;
            saveUserList(userList) ;
            return 1;
         }
      }
      return 0;
   }

   public int deleteUser(int id) {
      List<User> userList = getAllUsers() ;

      for(User user: userList) {
         if(user.getId() == id){
            int index = userList.indexOf(user) ;
            userList.remove(index) ;
            saveUserList(userList) ;
            return 1;
         }
      }
      return 0;
   }

   private void saveUserList(List<User> userList) {
      try {
         File file = new File("Users.dat") ;
         FileOutputStream fos;

         fos = new FileOutputStream(file) ;

         ObjectOutputStream oos = new ObjectOutputStream(fos) ;
         oos.writeObject(userList) ;
         oos.close() ;
      } catch (FileNotFoundException e) {
         e.printStackTrace() ;
      } catch (IOException e) {
         e.printStackTrace() ;
      }
   }
}
```

UserService.java

```
package com.dominio-organizacion;
```

```
import java.io.IOException;
import java.util.List;

import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.OPTIONS;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;

@Path("/UserService")
public class UserService {

   UserDao userDao = new UserDao() ;
   private static final String SUCCESS_RESULT="<result>success</result>";
   private static final String FAILURE_RESULT="<result>failure</result>";


   @GET
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)
   public List<User> getUsers() {
      return userDao.getAllUsers() ;
   }

   @GET
   @Path("/users/{userid}")
   @Produces(MediaType.APPLICATION_XML)
   public User getUser(@PathParam("userid") int userid){
      return userDao.getUser(userid) ;
   }

   @PUT
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)
   @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public String createUser(@FormParam("id") int id,
      @FormParam("name") String name,
      @FormParam("profession") String profession,
      @Context HttpServletResponse servletResponse) throws IOException{
      User user = new User(id, name, profession) ;
      int result = userDao.addUser(user) ;
      if(result == 1) {
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }

   @POST
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)
   @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public String updateUser(@FormParam("id") int id,
      @FormParam("name") String name,
      @FormParam("profession") String profession,
```

```
      @Context HttpServletResponse servletResponse) throws IOException{
      User user = new User(id, name, profession) ;
      int result = userDao.updateUser(user) ;
      if(result == 1) {
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }

   @DELETE
   @Path("/users/{userid}")
   @Produces(MediaType.APPLICATION_XML)
   public String deleteUser(@PathParam("userid") int userid){
      int result = userDao.deleteUser(userid) ;
      if(result == 1) {
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }

   @OPTIONS
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)
   public String getSupportedOperations() {
      return "<operations>GET, PUT, POST, DELETE</operations>";
   }
}
```

Ahora, utilizando Eclipse, vamos a exportar nuestra aplicación como un archivo WAR y desplegar los mismos en Tomcat. Para crear el archivo WAR utilizando Eclipse, seleccione la opción **File -> export -> Web > War File -> carpeta de destino** y, finalmente, seleccione el proyecto **UserManagemeNT.** Para hacer el despliegue del archivo .WAR en Tomcat, guarde el *UserManagement.war* en **Tomcat Installation Directory > webapps** (directorio de aplicaciones web**)** e iniciar Tomcat.

8.1. Probar el servicio Web

Jersey proporciona las APIs necesarias para crear un cliente para probar los servicios web. Hemos creado una clase de prueba de muestra **WebServiceTester.java** bajo el paquete *com.dominio-organizacion* en el mismo proyecto.

WebServiceTester.java

```
package com.dominio-organizacion;

import java.util.List;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.Form;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;

public class WebServiceTester  {

   private Client client;
   private String REST_SERVICE_URL =
"http://localhost:8080/UserManagement/rest/UserService/users";
```

```
private static final String SUCCESS_RESULT="<result>success</result>";
private static final String PASS = "pass";
private static final String FAIL = "fail";

private void init() {
   this.client = ClientBuilder.newClient() ;
}

public static void main(String[] args) {
   WebServiceTester tester = new WebServiceTester() ;
   //initialize the tester
   tester.init() ;
   //test get all users Web Service Method
   tester.testGetAllUsers() ;
   //test get user Web Service Method
   tester.testGetUser() ;
   //test update user Web Service Method
   tester.testUpdateUser() ;
   //test add user Web Service Method
   tester.testAddUser() ;
   //test delete user Web Service Method
   tester.testDeleteUser() ;
}
//Test: Get list of all users
//Test: Check if list is not empty
private void testGetAllUsers() {
   GenericType<List<User>> list = new GenericType<List<User>>() {};
   List<User> users = client
      .target(REST_SERVICE_URL)
      .request(MediaType.APPLICATION_XML)
      .get(list) ;
   String result = PASS;
   if(users.isEmpty() ){
      result = FAIL;
   }
   System.out.println("Test case name: testGetAllUsers, Result: " + result ) ;
}
//Test: Get User of id 1
//Test: Check if user is same as sample user
private void testGetUser() {
   User sampleUser = new User() ;
   sampleUser.setId(1) ;

   User user = client
      .target(REST_SERVICE_URL)
      .path("/{userid}")
      .resolveTemplate("userid", 1)
      .request(MediaType.APPLICATION_XML)
      .get(User.class) ;
   String result = FAIL;
   if(sampleUser != null && sampleUser.getId() == user.getId() ){
      result = PASS;
   }
   System.out.println("Test case name: testGetUser, Result: " + result ) ;
}
//Test: Update User of id 1
//Test: Check if result is success XML.
private void testUpdateUser() {
   Form form = new Form() ;
   form.param("id", "1") ;
   form.param("name", "suresh") ;
   form.param("profession", "clerk") ;
```

```
      String callResult = client
         .target(REST_SERVICE_URL)
         .request(MediaType.APPLICATION_XML)
         .post(Entity.entity(form,
            MediaType.APPLICATION_FORM_URLENCODED_TYPE),
            String.class);
      String result = PASS;
      if(!SUCCESS_RESULT.equals(callResult) ){
         result = FAIL;
      }

      System.out.println("Test case name: testUpdateUser, Result: " + result ) ;
   }
   //Test: Add User of id 2
   //Test: Check if result is success XML.
   private void testAddUser() {
      Form form = new Form() ;
      form.param("id", "2") ;
      form.param("name", "naresh") ;
      form.param("profession", "clerk") ;

      String callResult = client
         .target(REST_SERVICE_URL)
         .request(MediaType.APPLICATION_XML)
         .put(Entity.entity(form,
            MediaType.APPLICATION_FORM_URLENCODED_TYPE),
            String.class);

      String result = PASS;
      if(!SUCCESS_RESULT.equals(callResult) ){
         result = FAIL;
      }

      System.out.println("Test case name: testAddUser, Result: " + result ) ;
   }
   //Test: Delete User of id 2
   //Test: Check if result is success XML.
   private void testDeleteUser() {
      String callResult = client
         .target(REST_SERVICE_URL)
         .path("/{userid}")
         .resolveTemplate("userid", 2)
         .request(MediaType.APPLICATION_XML)
         .delete(String.class) ;

      String result = PASS;
      if(!SUCCESS_RESULT.equals(callResult) ){
         result = FAIL;
      }

      System.out.println("Test case name: testDeleteUser, Result: " + result ) ;
   }
}
```

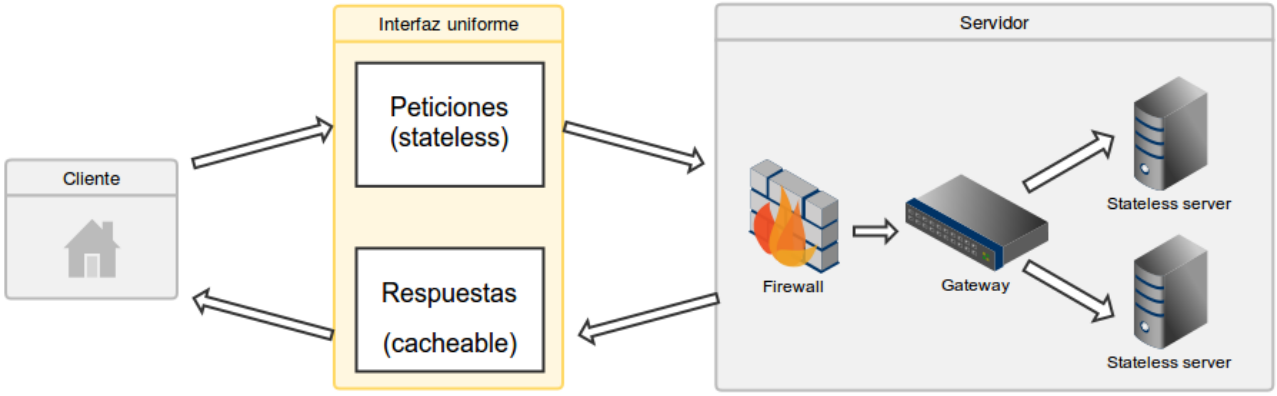
Ahora ejecute la clase de prueba con Eclipse. Haga clic derecho sobre el archivo y seleccione la opción **Run as -> Java Application.** Verá el siguiente resultado en la consola de Eclipse:

```
Test case name: testGetAllUsers, Result: pass
Test case name: testGetUser, Result: pass
Test case name: testUpdateUser, Result: pass
```

```
Test case name: testAddUser, Result: pass
Test case name: testDeleteUser, Result: pass
```

9. Stateless

De acuerdo con la arquitectura REST, un servicio web RESTful no debe mantener el estado del cliente en el servidor. Esta restricción se conoce como "sin estado" (*stateless*). Es responsabilidad del cliente el pasar su contexto al servidor para que éste pueda guardar este contexto para procesar futuras peticiones del cliente. Por ejemplo, la sesión mantenida por el servidor es identificada por el identificador de sesión enviada por el cliente.



RESTful Web services debería cumplir la restricción anterior. Como se ha comentado anteriormente los métodos de un servicio web no guardan ninguna información del cliente que los invoca.

Considere la siguiente URL:

```
http://localhost:8080/UserManagement/rest/UserService/users/1
```

Si se escribe la URL anterior en el navegador, usando un cliente basado en Java o a través de Postman, el resultado siempre será el usuarios XML cuyo Id es 1 ya que el servidor no guarda ninguna información acerca del cliente.

```
<user>
   <id>1</id>
   <name>mahesh</name>
   <profession>1</profession>
</user>
```

9.1. Ventajas de Statelessness

A continuación se enumeran los beneficios de los servicios web RESTful *stateless*:

- Los servicios web pueden tratar cada método de la petición de forma independiente.
- Los servicios web no necesitan mantener interacciones anteriores de un cliente. Esto simplifica el diseño de la aplicación.
- Así como HTTP es un protocolo sin estado en sí mismo, los servicios web RESTful funcionan perfectamente con el protocolo HTTP.

9.2. Desventajas de Statelessness

Los siguientes serían los inconvenientes de los servicios web RESTful *stateless*:

- Los servicios web necesitan obtener información extra en cada petición para después interpretarla y así conseguir el estado del cliente en caso de que la interacción con el este deba ser atendida.

10. Caching

El almacenamiento en memoria caché (*caching*) permite guardar la respuesta del servidor en el propio cliente para que el cliente no necesite hacer otra petición del mismo recurso al servidor una y otra vez. La respuesta de un servidor debería guardar información acerca de cómo se hace el *caching*. De esta forma un cliente puede cachear la respuesta durante un periodo de tiempo o que no "cachee" la respuesta del servidor nunca.

En la tabla siguiente se detallan las cabeceras que la respuesta de un servidor puede tener para configurar el *caching* del cliente:

Header	Descripción
Date	Fecha y Hora de creación del recurso.
Last Modified	Fecha y Hora de la última modificación del recurso.
Cache-Control	Cabecera primaria para controlar el *caching*. Primary header to control caching.
Expires	Fecha y hora de expiración del *caching.*
Age	Duración en segundos desde que el recurso fue encontrado en el servidor.

10.1. Cabecera Cache-Control

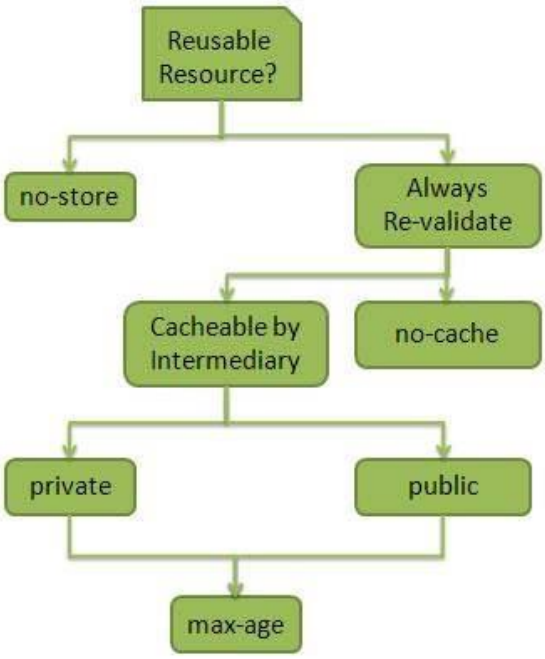
A continuación se describen la cabecera Cache-Control:

Directiva	Descripción
Public	Indica que el recurso es "cacheable" en cualquier momento.
Private	Indica que el recurso es "cacheable" solo por el cliente y el servidor, ningún intermediario puede "cachear" el recurso.
no-cache/no-store	Indica que el recurso no es "cacheable".
max-age	Indica que el *caching* es válido hasta *max-age* en segundos. Cuando se sobrepase este tiempo, el cliente tiene que volver a hacer otra petición.
must-revalidate	Le dice al servidor que revalide el recurso si se ha superado *max-age*.

10.2. Buenas prácticas

- Conserve siempre el contenido estático (imágenes, CSS, JavaScript) "cacheable" con fecha de expiración de 2 o 3 días.

- Nunca mantenga fechas de expiración demasiado largas.
- El contenido dinámico debería ser siempre “cacheado” sólo durante pocas horas.



11. Seguridad

Ya que los servicios web RESTful trabajan con rutas HTTP (URLs), es muy importante salvaguardar un servicio web RESTful de la misma forma que se asegura un sitio web. A continuación se presentan una serie de buenas prácticas que se deben seguir al diseñar un servicio web RESTful:

- **Validación** – Validar todas las entradas en el servidor. Proteger su servidor contra ataques de inyección SQL o NoSQL.
- **Autenticación basada en sesión** - Autenticación basada en sesión para autenticar a un usuario cada vez que se hace una petición a un método del servicio web.
- **Evitar datos personales en URL** – Nunca usar el nombre de usuario, contraseña o "token" de sesión en la URL, estos valores deben pasarse al servicio web a través del método POST.
- **Ejecución de métodos restringido** – Uso restringido de los métodos GET, POST, DELETE. El método GET no debería permitir borrar datos.
- **Validación de XML/JSON** - Comprobar que las entradas que se envíen al método del servicio web estén bien formadas.
- **Uso de mensajes de error genéricos** – El método de un web service debería usar mensajes de error HTTP como 403 para indicar acceso prohibido, etc.

11.1. HTTP Status Codes

Usar siempre códigos HTTP estándar para devolver la respuesta HTTP al cliente.

Código HTTP	Descripción
200	**OK**, resultado de la operación correcto.
201	**CREATED**, cuando un recurso se ha creado satisfactoriamente con una petición PUT o POST. Devuelve un enlace al recurso recién creado usando la cabecera de localización.
204	**NO CONTENT**, cuando el cuerpo de la respuesta está vacío por ejemplo, una petición DELETE.
304	**NOT MODIFIED**, se utiliza para reducir el uso de ancho de banda de la red en caso de peticiones GET condicionales. El cuerpo del mensaje de respuesta debería estar vacío. Las cabeceras deberían tener fecha, ubicación, etc.
400	**BAD REQUEST**, indica que se ha hecho una petición no válida, por ejemplo, error de validación, datos no encontrados.
401	**UNAUTHORIZED**, indica autenticación del usuario no válida.

403	**FORBIDDEN**, advierte de que el usuario no tiene permisos para realizar una operación concreta, por ejemplo, acceso de borrado sin privilegios de administrador.
404	**NOT FOUND**, método o recurso no disponible.
409	**CONFLICT**, advierte de una situación conflictiva cuando se ejecuta un método, por ejemplo, al intentar insertar una entrada duplicada.
500	**INTERNAL SERVER ERROR**, indica que el servidor ha lanzado una excepción al ejecutarse un método/operación.

12. Java (JAX-RS)

JAX-RS es el acrónimo de **JAVA API for RESTful Web Services**. JAX-RS es un API basado en el lenguaje de programación JAVA y la especificación para dar soporte a servicios web RESTful. Su versión 2.0 se distribuyó el 24 de mayo de 2013. JAX-RS hace bastante uso de las anotaciones disponibles en Java SE 5 para simplificar el desarrollo, creación y despliegue de servicios web basados en Java of JAVA. Asimismo permite la creación de clientes para RESTful web services.

12.1. Especificación

A continuación se detallan las anotaciones más comúnmente usadas para mapear un recurso como un servicio web.

Nº	Anotación	Descripción
1	**@Path**	Ruta relativa de la clase/método del recurso.
2	**@GET**	Petición HTTP GET, para obtener un recurso.
3	**@PUT**	Petición HTTP PUT, para modificar/reemplazar un recurso.
4	**@POST**	Petición HTTP POST, para crear/actualizar un recurso.
5	**@DELETE**	Petición HTTP DELETE, para eliminar un recurso.
6	**@HEAD**	Petición HTTP HEAD, para obtener el estado de la disponibilidad del método.
7	**@Produces**	Establece la respuesta HTTP generada por el servicio web, por ejemplo application/xml, text/html, application/json etc.
8	**@Consumes**	Establece el tipo de petición HTTP, por ejemplo, *application/x-www-form-urlencoded* para aceptar formulario de datos del cuerpo HTTP durante una petición POST.
9	**@PathParam**	Asocia el parámetro pasado al método a un valor de la ruta.
10	**@QueryParam**	Asocia el parámetro pasado al método a un parámetro de consulta de la ruta.
11	**@MatrixParam**	Asocia el parámetro pasado al método a un parámetro HTTP matriz de la ruta.
12	**@HeaderParam**	Asocia el parámetro pasado al método a una cabecera HTTP.
13	**@CookieParam**	Asocia el parámetro pasado al método a una Cookie.

14	**@FormParam**	Asocia el parámetro pasado al método a un valor de formulario.
15	**@DefaultValue**	Asigna un valor por defecto a un parámetro pasado al método.
16	**@Context**	Contexto del recurso, por ejemplo petición HTTP como contexto.
17	**@BeanParam**	Permite inyectar una clase de aplicación específica cuyos métodos o campos están anotados con alguno de los parámetros de inyección existentes.
18	**@Encoded**	Los parámetros de los métodos JAX-RS por defecto decodifican los valores obtenidos de la petición. @Encoded permite mantener los valores cifrados si queremos descifrarlos por nuestra cuenta. Pueden estar anotados a nivel de clase, a nivel de método y a nivel de parámetros.

13. Bibliografía

- RESTful Java with JAX-RS 2.0. Designing and developing distributed web services. Bill Burke. O'Reilly.
  https://www.gitbook.com/book/dennis-xlc/restful-java-with-jax-rs-2-0-2rd-edition/details
- Diseño e Implementación de Servicios Web. Rafael Navarro Marset.
- Jersey: RESTful Web Services in Java.
  https://jersey.github.io/

NOTA: En este manual se ha trabajado con Jersey, la implementación de referencia de JAX-RS 2.0 de Oracle.