
	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2022/2023		
	UNIDAD COMPETENCIA:							

Tema 5: Servicios web RESTFul Web Services

Índice

1. Introducción.....	1
2. RESTful Web Services	1
2.1. Características.....	1
2.2. Recursos	3
2.3. Representaciones	4
3. JAX-RS.....	4
3.1. Configuración inicial	5
3.1.1. Jersey 2.x.....	5
3.1.2. Jersey 3.x.....	11
3.2. Servicio de prueba.....	16
3.3. Anotaciones	20
3.4. Serializado y deserializado XML.....	22
3.5. @PathParam: Parámetros en la URI	22
3.6. @QueryParam: Parámetros en las peticiones	23
3.7. @DefaultValue: Valores por defecto	23
3.8. @FormParam: Extraer información de un formulario.....	24
3.9. @HeaderParam: Parámetros en las cabeceras	25
3.10. @XmlAttribute: Atributos XML.....	25
3.11. @XmlElement: Modificando la estructura de los datos.....	25
3.12. Response y ResponseBuilder.....	26
3.13. Ficheros binarios.....	27
3.14. Gestión de excepciones	28
3.15. Links	28
3.16. Creación de URIs.....	28
4. Apéndice I. Creación de un cliente Rest	30
4.1. Cerrando conexiones.....	33
5. Apéndice II: Control de errores.....	34
6. Bibliografía	35

	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

1. Introducción

Un servicio Web (Web Service) es una tecnología que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en distintos lenguajes de programación y ejecutadas sobre diversas plataformas, pueden utilizar los servicios web para intercambiar datos con el objetivo de ofrecer unos servicios. Los proveedores ofrecen sus servicios y los usuarios los solicitan un servicio a través de la Web.

Dos tipos de servicios web:

- Basados en SOAP¹ (Simple Object Access Protocol).
 - Basados en XML y XML-Schema.
 - Con estructura definida (WSDL²).
 - Más pesados.

- Tipo REST³ (REpresentational State Transfer).

REST nos permite crear servicios y aplicaciones que pueden ser usadas por cualquier dispositivo o cliente que entienda HTTP, por lo que es increíblemente más simple y convencional que otras tecnologías como puede ser SOAP.

2. RESTful Web Services⁴

Un servicio web REST se define como un "estilo arquitectónico" para el desarrollo de aplicaciones distribuidas.

- Forma de construir aplicaciones distribuidas.
- Centrada en el concepto de recurso⁵.
- El resultado de una petición a un servicio REST es la información en un formato determinado (XML, JSON, ...) pero sin información adicional de ningún protocolo (como ocurre en SOAP) .

2.1.Características

- Basado principalmente en HTTP (aprovechando la infraestructura web ya existente) y diversos formatos como pueden ser XML o JSON.
- Menos formalizados y definidos.
- Sin estructurar.
- Ligeros.


¹ http://es.wikipedia.org/wiki/Simple_Object_Access_Protocol

² <http://es.wikipedia.org/wiki/WSDL>

³ http://es.wikipedia.org/wiki/Representational_State_Transfer

⁴ <http://docs.oracle.com/javaee/7/tutorial/jaxrs001.htm>

⁵ http://es.wikipedia.org/wiki/Representational_State_Transfer#Recursos

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos				CURSO:	2º	
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Sin estado (stateless): El servidor no guarda el estado del cliente sino que es el cliente el que mantiene el estado y provee en la petición la información necesaria.
- Cliente/Servidor.
- Manipulación de recursos: Accesibles mediante métodos HTTP → GET, POST, PUT y DELETE.
 - Usa de forma explícita los métodos HTTP para ejecutar acciones CRUD⁶:

Verbo Rest	Acción	Descripción
POST	Create	Crea un nuevo recurso
GET	Read	Define un acceso de lectura a un recurso
PUT	Update	Actualiza un recurso existente o lo crea
DELETE	Delete	Borra un recurso

- Las peticiones y respuestas incluyen metadatos (tipo MIME⁷, cache, validez, ...).
- Están incluidos en las cabeceras HTTP⁸.
- Asíncrono: El servidor puede cambiar el estado real un recurso y el cliente, de forma independiente, puede modificar su copia local del estado del recurso.
- Clientes acceden al estado de recurso mediante representaciones del mismo.
 - El parámetro Content-Type de las cabeceras HTTP especifica el tipo MIME de los datos intercambiados (text/html, application/xml, application/json,...).
 - El cliente puede informar al servidor del tipo de representación que necesita mediante el parámetro Accept en cabecera de las peticiones HTTP (text/html, application/xml, application/json,...).
- Tejido hypermedia como representación del estado de la aplicación.
 - Los datos retornados pueden incluir enlaces a otros recursos.
 - HATEOAS⁹: Hypermedia As The Engine Of Application State.
- Las APIs que publican muchos de los sitios web actualmente no son más que servicios web de tipo REST, aunque en la mayoría de los casos con medidas de seguridad adicionales tales como autenticación OAuth¹⁰ o similares.

⁶ <http://es.wikipedia.org/wiki/CRUD>

http://en.wikipedia.org/wiki/Create,_read,_update_and_delete


⁷ http://es.wikipedia.org/wiki/Multipurpose_Internet_Mail_Extensions

⁸ https://es.wikipedia.org/wiki/Cabeceras_HTTP

http://en.wikipedia.org/wiki/List_of_HTTP_header_fields

⁹ <https://es.wikipedia.org/wiki/HATEOAS> <http://en.wikipedia.org/wiki/HATEOAS>

¹⁰ <http://es.wikipedia.org/wiki/OAuth>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

2.2.Recursos

Los recursos son elementos de información (una persona, una dirección, una imagen, ...) a la que queremos acceder ya sea para consultar, modificar, borrar o crear. Un recurso es independiente de la representación que pueda tener. Esta se puede presentar en distintos formatos: XML, JSON, PDF, JPEG, texto plano, ...

Los recursos están identificados universalmente mediante URIs¹¹ (Uniform Resource Identifier). Estas deben ser intuitivas y que además de identificar de forma única el recurso nos deben permitir localizarlo para poder acceder a él.


Las URIs se definen como una estructura de carpetas o directorios.

- Una persona: <http://example.org/persona/123>.
- Una película: <http://example.org/pelicula/Terminator>.
- Un grupo de personas: <http://example.org/personas?inicio=10&cant=100>.

Existen una serie de reglas para establecer las URIs de un recurso:

- Ocultar la tecnología usada en el servidor (.jsp, .php, .asp, ...), de manera que se puedan usar distintas tecnologías sin cambiar de URI.
- Los nombres de URI no deben implicar una acción, por lo tanto debe evitarse usar verbos en ellos y deben ser únicos. Además no debemos tener más de una URI para identificar un mismo recurso.
 - Por ejemplo, /facturas/234/editar sería incorrecta ya que tenemos el verbo editar en la misma.
 - Para el recurso factura con el identificador 234, la siguiente URI sería la correcta, independientemente de que vayamos a editarla, borrarla, consultarla o leer sólo uno de sus conceptos: /facturas/234.
- Deben ser independiente de formato.
 - Por ejemplo, /facturas/234.pdf no sería una URI correcta, ya que estamos indicando la extensión pdf en la misma.
 - Para el recurso factura con el identificador 234, la siguiente URI sería la correcta, independientemente de que vayamos a consultarla en formato pdf, epub, txt, xml o json: /facturas/234.
- Deben mantener una jerarquía lógica.
 - Por ejemplo, /facturas/234/cliente/007 no sería una URI correcta, ya que no sigue una jerarquía lógica.
 - Para el recurso factura con el identificador 234 del cliente 007, la siguiente URI sería la correcta: /clientes/007/facturas/234.
- Los filtrados de información de un recurso no se hacen en la URI.
 - Para filtrar, ordenar, paginar o buscar información en un recurso, debemos hacer una consulta sobre la URI, utilizando parámetros HTTP en lugar de incluirlos en la misma.

¹¹ http://es.wikipedia.org/wiki/Identificador_de_recursos_uniforme

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Por ejemplo, la URI /facturas/orden/desc/fecha-desde/2007/pagina/2 sería incorrecta ya que el recurso de listado de facturas sería el mismo pero utilizaríamos una URI distinta para filtrarlo, ordenarlo o paginarlo.
- La URI correcta en este caso sería:
/facturas?fecha-desde=2007&orden=DESC&pagina=2.

2.3.Representaciones

REST permite que los recursos sean presentados en diferentes tipos de datos: Text, HTML, XML, JSON,...

Una representación es un formato de datos de un recurso que se envía del servidor al cliente.

Por ejemplo:

- Una representación XML de un libro.
- Una representación JSON de un libro.
- Una imagen png de la portada de un libro.
-

3. JAX-RS¹²

JAX-RS (Java API for RESTful Web Services) es una API de Java que define una infraestructura (clases e interfaces) para la creación de servicios web de acuerdo con el estilo arquitectónico REST (paquete javax.ws.rs) usando para ello usa anotaciones¹³.

A partir de la versión 6 de java JAX-RS es parte oficial de Java EE.

- Jersey¹⁴ es la implementación de referencia, aunque hay otras como CXF (Apache) o RESTEasy (Jboss). Nos permite desarrollar tanto servidores REST como clientes REST.
- Gestiona automáticamente las representaciones de los recursos intercambiados.
 - Es capaz de mapear de XML a JSON en el caso del tipo MIME application/json.
 - La generación y tratamiento de otros tipos de representaciones debe manejarse manualmente por el programador (imágenes, PDF, ...).
- Basados en el concepto de POJO¹⁵ (Plain Old Java Object).
 - Clases sin extends ni implements.
 - Son clases de datos con sus getters y sus setters.

¹² <http://es.wikipedia.org/wiki/JAX-RS>

<https://docs.oracle.com/javaee/7/api/javax/ws/rs/package-summary.html>

¹³ http://es.wikipedia.org/wiki/Anotación_Java

¹⁴ <https://eclipse-ee4j.github.io/jersey/>

¹⁵ http://es.wikipedia.org/wiki/Plain_Old_Java_Object

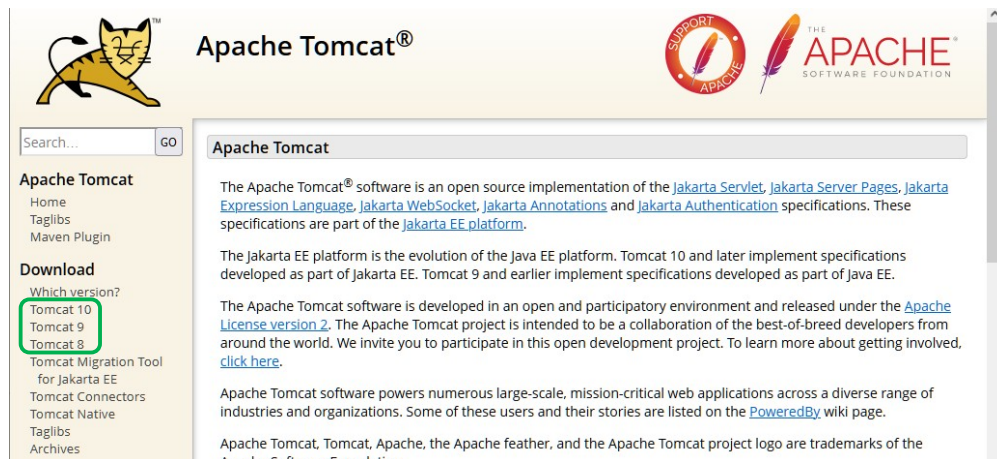
COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Tiene que existir una clase recurso raíz
 - Ya que las URIs son jerárquicas
 - Tiene que tener una anotación @Path en la clase o en un método → Especifica el path inicial.
 - Métodos con anotaciones @GET, @PUT, etc.

3.1. Configuración inicial

Necesitamos los siguientes elementos para comenzar a desarrollar servicios web Rest:

1. Eclipse IDE for Java EE Developers
2. Apache Tomcat¹⁶: Necesitamos un servidor apache Tomcat que se encargara de ejecutar el servicio web. Veremos en puros siguientes que versión descargar.



A partir de este punto necesitamos tener en cuenta los siguientes aspectos dependiendo de la versión:

3.1.1. Jersey 2.x

Donde:

- Funciona con versiones de Apache Tomcat 8 y 9.
- Utiliza JAXB para el mapeo de XML pero debido a que este ha sido deprecado en Java 9 y eliminado de forma completa desde Java 11¹⁷ se necesitará, para su utilización en versiones de Java 9 o superiores, descargar el jar con la especificación de JAXB¹⁸ en su versión 2.x y añadirla al proyecto (ya se verá donde).
- Utiliza los paquetes javax.ws.rs.* y javax.xml.bind.*

¹⁶ <http://es.wikipedia.org/wiki/Tomcat> <http://tomcat.apache.org/>

¹⁷ <https://www.jesperdj.com/2018/09/30/jaxb-on-java-9-10-11-and-beyond/>

¹⁸ <https://repo1.maven.org/maven2/com/sun/xml/bind/jaxb-ri/>
<https://javaee.github.io/jaxb-v2/>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Necesitamos descargar la especificación de JAX-RS. Para ello necesitamos descargar el zip con los JARs de Jersey¹⁹ (debemos añadir todos los jar presentes en todos los subdirectorios del archivo). El nombre del archivo tiene el formato: jaxrs-ri-X.Y.zip donde X es la versión e Y la subversión. Además podemos descargar el archivo con el jaxrs-ri-X.Y-javadoc.jar con la documentación javadoc. Esta se puede consultar online tanto la Api²⁰ como la guía²¹ de desarrollo

Actualmente se puede descargar los archivos necesarios desde las dos direcciones indicadas en el pie de página número 14.

¹⁹ <https://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jaxrs-ri/>

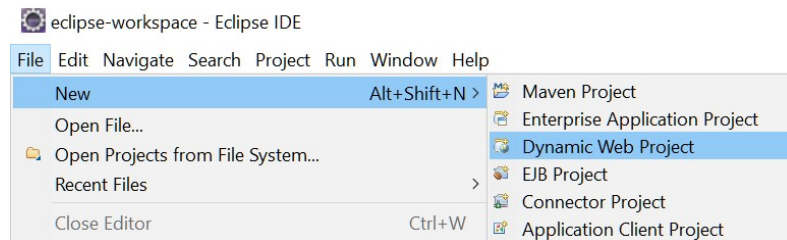
²⁰ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/latest/jersey/index.html>

²¹ <https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/index.html>

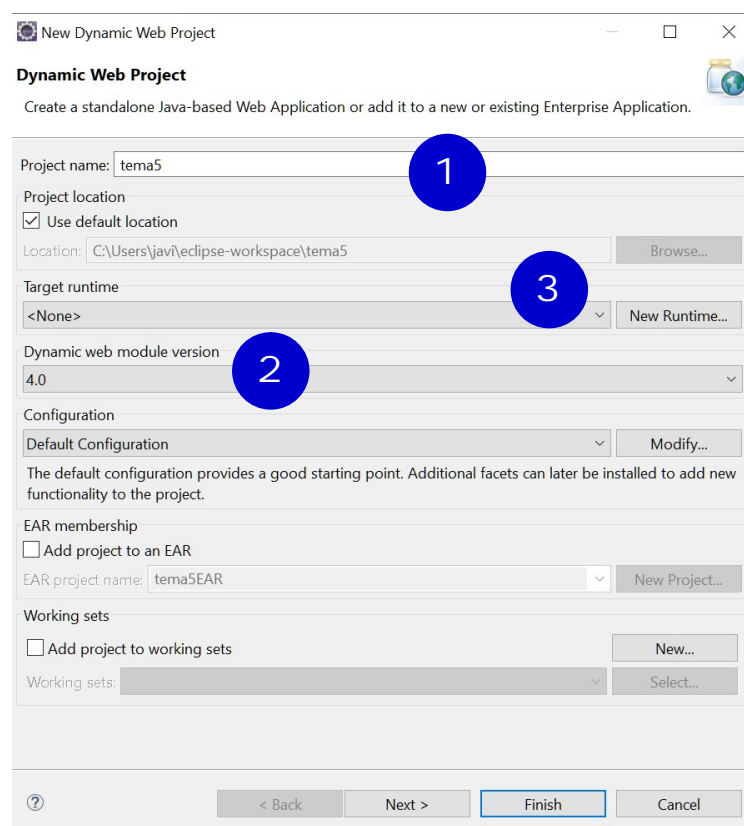
COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Seguiremos los siguientes pasos para construir un servicio web:

- Creamos un proyecto Dynamic Web Project.




- Escogemos las opciones del proyecto:

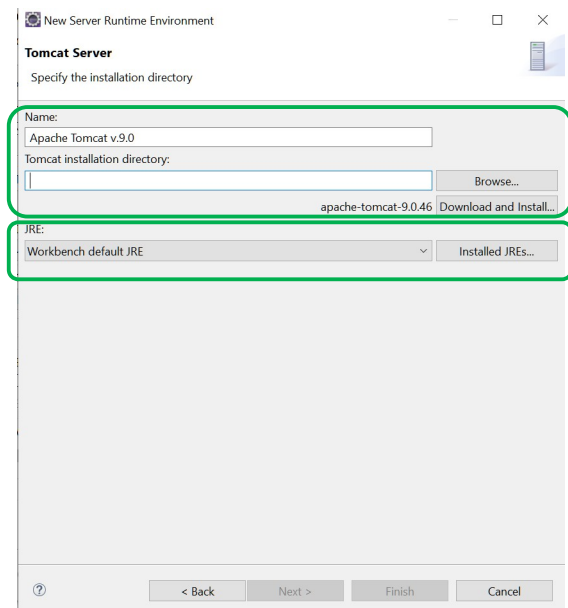
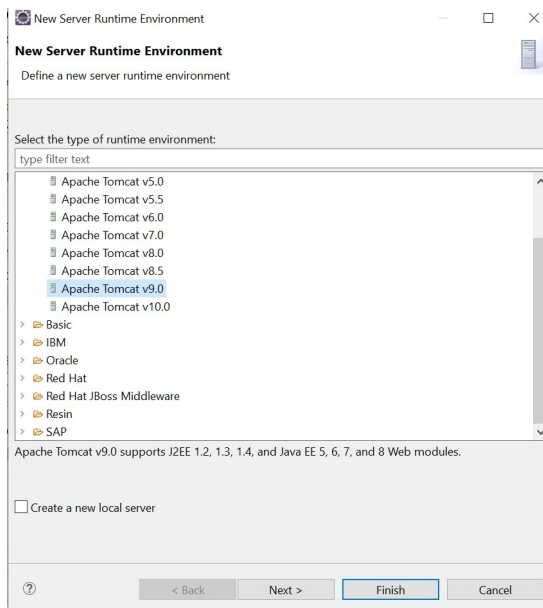


1. Nombre del proyecto: tema5.
2. Versión a utilizar del servlet²² utilizado para crear el contenido de la página web dinámica (Jersey implementa web services en un Java Servlet Container). Dejaremos el valor por defecto.

²² http://es.wikipedia.org/wiki/Java_Servlet http://en.wikipedia.org/wiki/Java_servlet

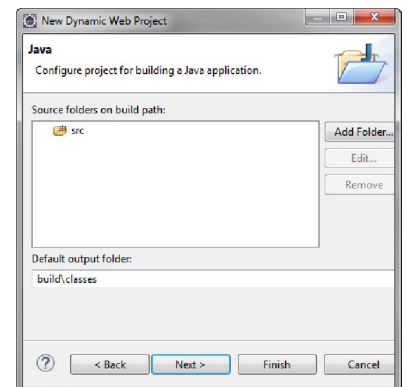
	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

3. Runtime de destino: escogeremos el servidor Apache Tomcat que ejecutará el servicio web. Si no tenemos ninguno creado se puede crear una nueva instancia del servidor (o usamos una existente), mediante el botón: New Runtime. Es lo mismo que:
- o Desde el menú principal de Eclipse escogemos File > New > Other...
 - o Seleccionamos Server > Server.
 - Escogemos la versión y las opciones de Tomcat.

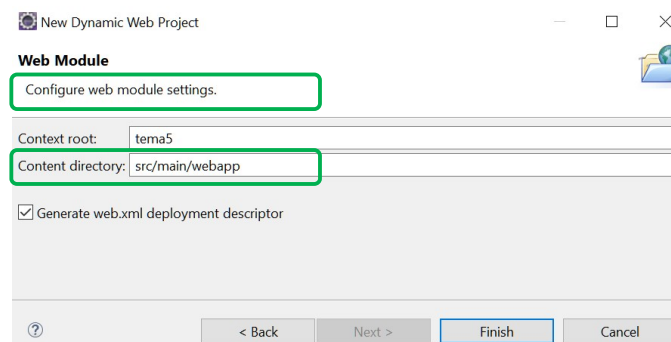



Establecemos el nombre del servidor Tomcat e indicamos el directorio donde se encuentra. Con la opción JRE se puede especificar la versión Java que utilizará Tomcat. Para terminar se pulsará Finish.

- Una vez establecidas todas las configuraciones en la ventana de creación del proyecto web dinámico se pulsa next pasando a la ventana situada a la derecha.

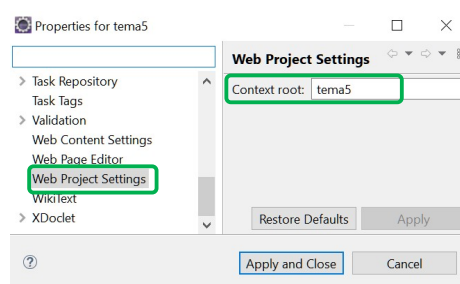


- Pulsando next se pasa a la ventana con configuración del módulo web. Donde:



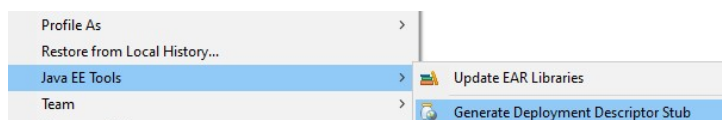
	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD COMPETENCIA:							

Context root el punto de entrada al servicio web. Es la base de la url que tendremos que indicar para conectarnos al servicio web. Su valor por defecto coincide con el nombre del proyecto. Se puede modificar tanto ahora como más adelante en la opción del proyecto Web Project Settings (para que este cambio funcione hay que limpiar el proyecto y volver a publicarlo en el servidor).

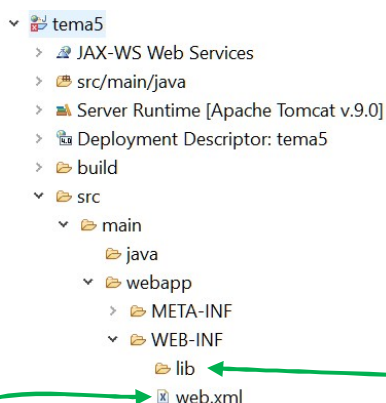


Finalizamos la creación del proyecto marcando la opción Generate web.xml deployment descriptor

Si nos olvidamos de realizar este paso se puede realizar posteriormente pulsando con el botón derecho en el árbol de proyectos (en la parte izquierda) y escogiendo la siguiente opción:



- Una vez finalizado podemos ver que en el explorador de proyectos se ha creado una estructura de directorios que se corresponde con un proyecto web dinámico.



- Debemos pegar en el directorio lib todos los jars que hay dentro del fichero jaxrs-ri-X.Y.zip que hemos descargado al principio del punto.
- Si usamos java 9 o una versión superior deberemos añadir soporte para JAXB añadiendo, a la carpeta lib, los jars presentes en el directorio mod del fichero jaxb-ri-2.x.x descargado al principio del punto.
- Ahora deberemos modificar el fichero web.xml²³ con el motivo de registrar el servlet²⁴ necesario para que la aplicación funcione correctamente en nuestro contenedor de servlets. Para ello añadiremos el siguiente código dentro del elemento <web-app>.

²³ http://es.wikipedia.org/wiki/Descriptor_de_despliegue

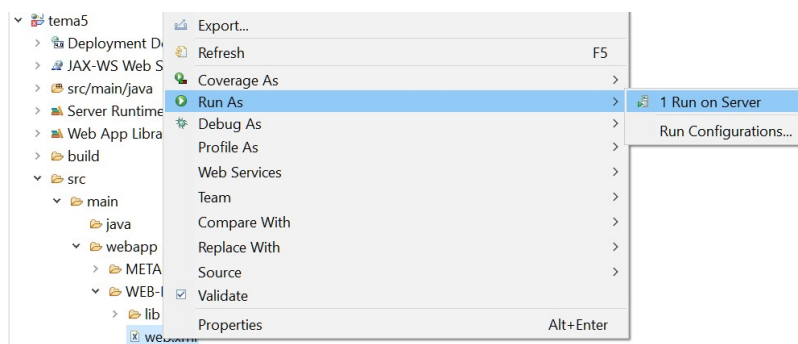
²⁴ https://en.wikipedia.org/wiki/Jakarta_Servlet

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

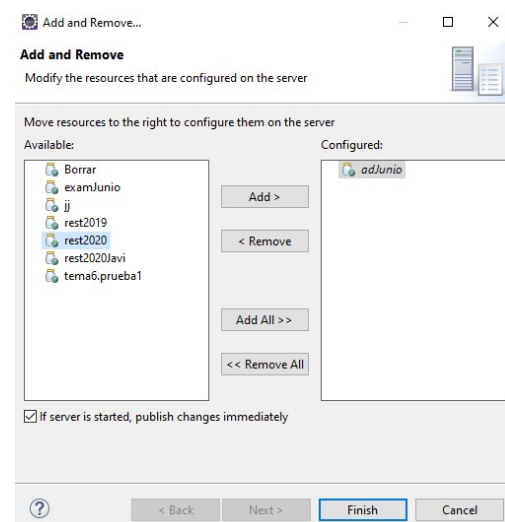
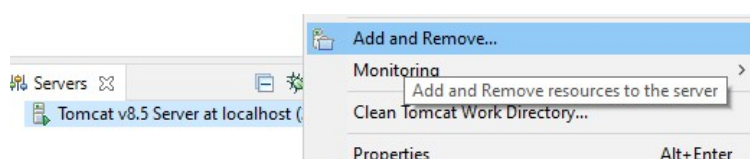
```
<servlet>
  <servlet-name>Jersey REST Service</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>ejem1</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Estamos indicando que existe un servlet (org.glassfish.jersey.servlet.ServletContainer) que se encarga de capturar todas las peticiones que le lleguen al subdirectorio rest del servidor y que las procesará como RESTful: tendrá en cuenta la URI y las operaciones de la cabecera para asignárselas a las clases correspondientes. El valor de url-pattern puede ser modificado por el que nosotros queramos. Este será el valor que hay que poner tras la url base del servicio.

- Mediante jersey.config.server.provider.packages indicamos el nombre del paquete (puede contener mayúsculas) donde están las clases java que implementaran el servicio web y a las que se tiene que mapear las peticiones RESTful. En nuestro ejemplo usamos el valor ejem1.
- Deberemos especificar que el proyecto se ejecute en el servidor Tomcat creado. Para ello pulsando con el botón derecho sobre el fichero web.xml se escoge la opción: Run As > Run on Server.



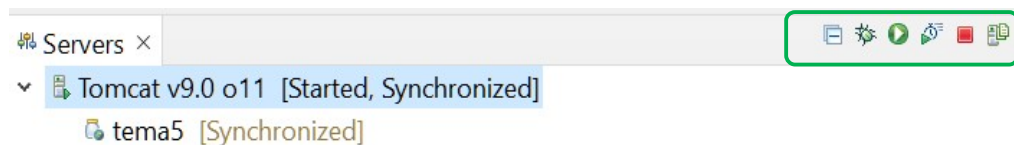
También se pueden añadir, y quitar, proyectos del servidor pulsando con el botón derecho encima de su nombre y escogiendo la opción Add and remove:



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos				CURSO:	2º	
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Una vez ejecutado el proyecto vemos que en la pestaña Servers aparece el servidor creado anteriormente y debajo de su nombre los proyectos que tiene añadidos. En este caso solo existe el proyecto tema5.

El servidor se puede administrar mediante los botones de la parte superior derecha de la ventana o pulsando directamente con el botón derecho encima de su nombre pudiendo arrancarlo, pararlo, reiniciarlo, ...



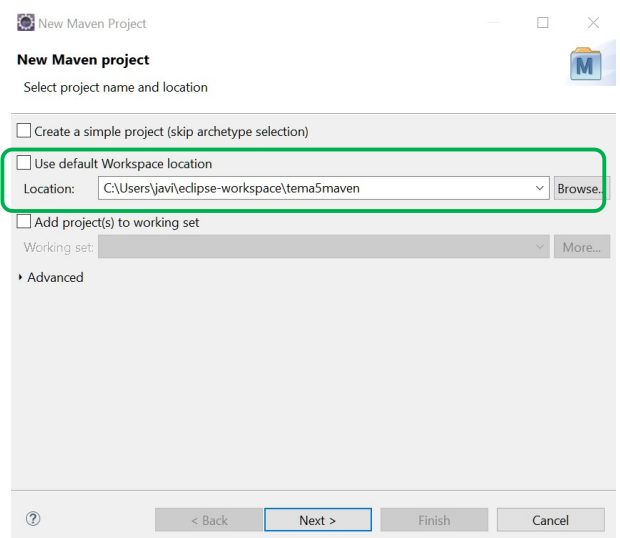
3.1.2. Jersey 3.x

Donde:

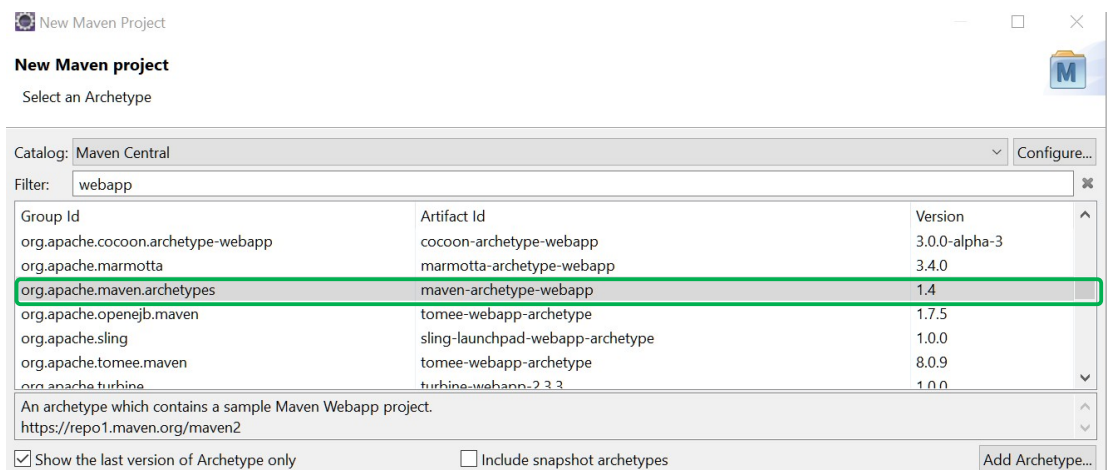
- Funciona con Apache Tomcat 10.
- Utilizaremos Maven para su configuración.
- Utiliza los paquetes jakarta.ws.rs.* y jakarta.xml.bind.*

Seguiremos los siguientes pasos para construir un servicio web usando Maven:

- Creamos un proyecto Maven. En este punto podemos cambiar la carpeta donde crear nuestro proyecto. Pulsamos next para continuar



- Escogemos el arquetipo maven. En nuestro caso su group id es org.apache.maven.archetypes y su id es maven-archetype-webapp.



COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- En la siguiente ventana deberemos cubrir los siguientes campos²⁵:
- Group Id: es un identificador único para cada proyecto. Se suele usar un formato de dominio inverso con el nombre de la organización y con él se crea el paquete donde se codificarán las clases del servicio web. En nuestro caso, al igual que en el caso de Jersey 2.x usaremos un valor sencillo: ejem1.
 - Artifact Id: mediante este campo se define el nombre del proyecto y que coincide con el jar que se crea sin versión. Tiene que estar en minúsculas y sin símbolos extraños. Usaremos el valor de tema5maven.
 - Versión: representa la versión del proyecto. Esta versión puede venir junto con dos sufijos:
 - SNAPSHOT: representa una versión que aún está en desarrollo y que por tanto puede ir cambiando.
 - RELEASE: representa una versión estable no se va a modificar.

Dejaremos el valor por defecto.


- Una vez finalizado la configuración se crea el proyecto con la siguiente estructura.

```

▼ tema5maven
  > Deployment Descriptor: Archetype Created Web Application
  > JRE System Library [JavaSE-1.7]
  > Maven Dependencies
  > Deployed Resources
  ▼ src
    ▼ main
      ▼ webapp
        ▼ WEB-INF
          web.xml
          index.jsp
    > target
      pom.xml
  
```

El error que se muestra se soluciona eliminando el fichero index.jsp que no usaremos.

²⁵ <https://maven.apache.org/guides/mini/guide-naming-conventions.html>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Establecemos en el fichero pon.xml las dependencias necesarias:

```
<!-- the REST API -->
<!-- https://mvnrepository.com/artifact/jakarta.ws.rs/jakarta.ws.rs-api -->
<dependency>
  <groupId>jakarta.ws.rs</groupId>
  <artifactId>jakarta.ws.rs-api</artifactId>
  <version>3.0.0</version>
</dependency>
```

```
<!-- Jersey core client implementation -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.core/jersey-client -->
<dependency>
  <groupId>org.glassfish.jersey.core</groupId>
  <artifactId>jersey-client</artifactId>
  <version>3.0.3</version>
</dependency>
```

```
<!-- Jersey core Servlet 3.x implementation -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.containers/jersey-container-servlet-core -->
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet-core</artifactId>
  <version>3.0.3</version>
</dependency>
```

```
<!-- HK2 InjectionManager implementation -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.inject/jersey-hk2 -->
<dependency>
  <groupId>org.glassfish.jersey.inject</groupId>
  <artifactId>jersey-hk2</artifactId>
  <version>3.0.3</version>
</dependency>
```

```
<!-- Jersey JSON Jackson (2.x) entity providers support module. -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.media/jersey-media-json-jackson -->
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId>
  <version>3.0.3</version>
</dependency>
```

```
<!-- JAX-RS features based upon JAX-B. XML. -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.media/jersey-media-jaxb -->
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-jaxb</artifactId>
  <version>3.0.3</version>
</dependency>
```

```
<!-- application.wadl -->
<!-- https://mvnrepository.com/artifact/org.glassfish.jaxb/jaxb-runtime -->
<dependency>
  <groupId>org.glassfish.jaxb</groupId>
  <artifactId>jaxb-runtime</artifactId>
  <version>3.0.2</version>
</dependency>
```

```
<!-- the REST API https://mvnrepository.com/artifact/javax.ws.rs/javax.ws.rs-api -->
<!--<dependency> -->
<!--      <groupId>javax.ws.rs</groupId> -->
<!--      <artifactId>javax.ws.rs-api</artifactId> -->
<!--      <version>2.1.1</version> -->
<!--</dependency> -->
```

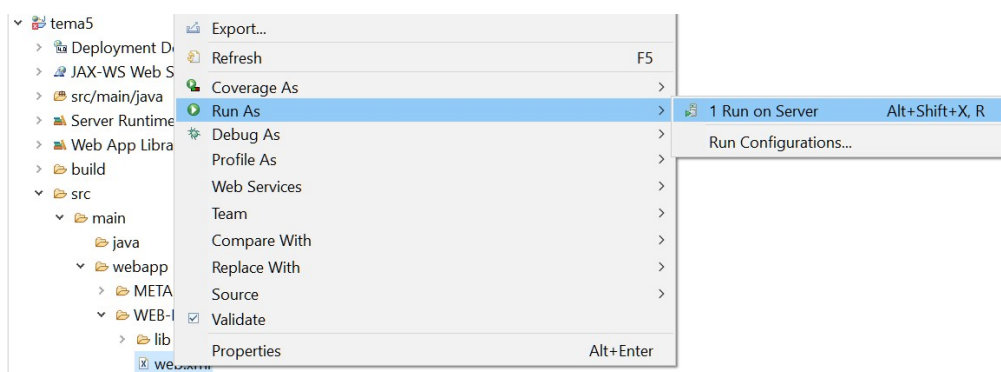

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

- Ahora deberemos modificar el fichero web.xml²⁶ con el motivo de registrar el servlet necesario para que la aplicación funcione correctamente en nuestro contenedor de servlets²⁷. Para ello añadimos el siguiente código dentro del elemento `<web-app>`.

```
<servlet>
  <servlet-name>Jersey REST Service</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>ejem1</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey REST Service</servlet-name>
  <url-pattern>/rest/*</url-pattern>
</servlet-mapping>
```

Estamos indicando que existe un servlet (org.glassfish.jersey.servlet.ServletContainer) que se encarga de capturar todas las peticiones que le lleguen al subdirectorio rest del servidor y que las procesará como RESTful: tendrá en cuenta la URI y las operaciones de la cabecera para asignárselas a las clases correspondientes. El valor de url-pattern puede ser modificado por el que nosotros queramos. Este será el valor que hay que poner tras la url base del servicio.

- Mediante jersey.config.server.provider.packages indicamos el nombre del paquete (puede contener mayúsculas) donde están las clases java que implementaran el servicio web y a las que se tiene que mapear las peticiones RESTful. En nuestro ejemplo usamos el valor ejem1.
- En la carpeta src/main creamos el directorio java y dentro de él el paquete que contendrá las clases java del proyecto. En nuestro ejemplo ejem1.
- Deberemos especificar que el proyecto se ejecute en el servidor Tomcat creado. Para ello pulsando con el botón derecho sobre el fichero web.xmls se escoge la opción: Run As > Run on Server.

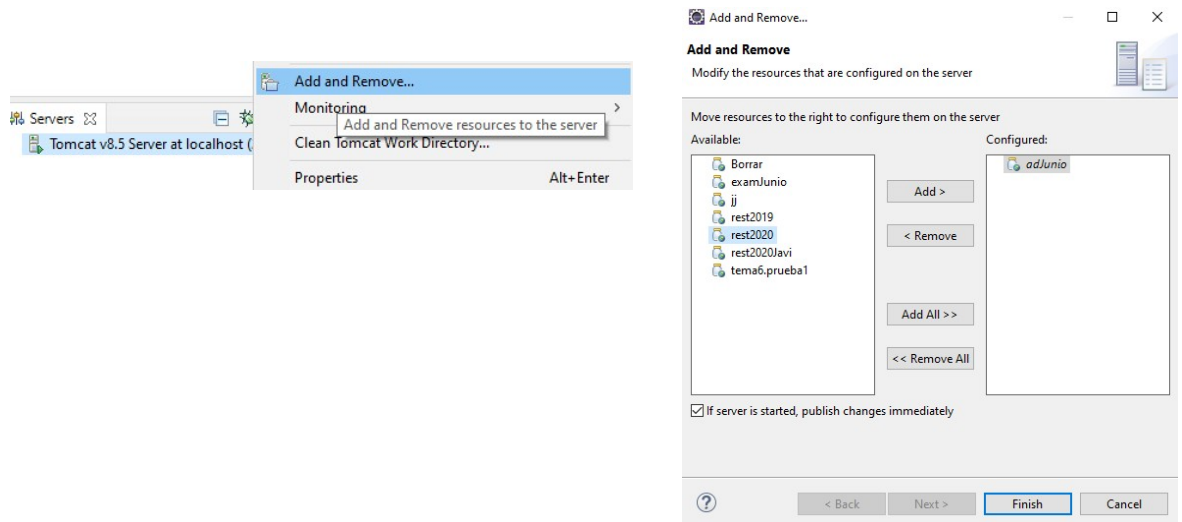


²⁶ http://es.wikipedia.org/wiki/Descriptor_de_despliegue

²⁷ https://en.wikipedia.org/wiki/Jakarta_Servlet

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					


También se pueden añadir, y quitar, proyectos pulsando con el botón derecho encima del nombre del servidor y escogiendo la opción Add and remove:



- Una vez ejecutado el proyecto vemos que en la pestaña Servers aparece el servidor creado anteriormente y debajo de su nombre los proyectos que tiene añadidos. En este caso solo existe el proyecto tema5maven.

El servidor se puede administrar mediante los botones de la parte superior derecha de la ventana o pulsando directamente con el botón derecho encima de su nombre pudiendo arrancarlo, pararlo, reiniciarlo, ...



	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

3.2.Servicio de prueba

Vamos a implementar un servicio para poder probar su funcionamiento.

Dentro del paquete especificado en el fichero web.xml creamos las clases Hola, Car y Coche. En los dos ejemplos es ejem1:

Ejemplo 1

```
package ejem1;

/* En gris se ponen los imports que se usan en la versión 2.x de Jersey */
import jakarta.ws.rs.GET;           // javax.ws.rs.GET;
import jakarta.ws.rs.Path;          // javax.ws.rs.Path;
import jakarta.ws.rs.Produces;      // javax.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType; // javax.ws.rs.core.MediaType;

// Establece la ruta del servicio: URL base + /hola
@Path("/hola")
public class Hola {

    // Se ejecuta este método si se pide un Accept de tipo TEXT_PLAIN
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String textHola() {
        return "Hola Rest Soy un texto";
    }

    // Se ejecuta este método si se pide un Accept de tipo TEXT_HTML
    @GET
    @Produces(MediaType.TEXT_HTML)
    public String htmlHola() {
        return "<html><title> Hola Rest</title><body>"
            + "<h1> Hola Rest</h1>"
            + "</body></html>";
    }

    // En los dos métodos siguientes el contenido se crea "a mano" y el valor
    // devuelto es un String genérico. En ejemplos siguientes veremos como
    // mejorar esto

    // Se ejecuta este método si se pide un Accept de tipo APPLICATION_XML
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public String xmlHola() {
        return "<?xml version='1.0' encoding='UTF-8' standalone='yes'?">"
            + "<hola>Hola Rest. Soy un XML</hola>";
    }

    // Se ejecuta este método si se pide un Accept de tipo APPLICATION_JSON
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public String jsonHola() {
        return "{\"hola\":\"Hola Rest. Soy un JSON\"}";
    }
}
```

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Ejemplo 2A

```
package ejem1;
import jakarta.xml.bind.annotation.XmlRootElement;
// import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement // Utilizado para generar un XML que representa esta clase
public class Car {
    private String marca;
    private String modelo;

    public Car() {
    }

    public Car(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
    }

    public String getMarca() {
        return marca;
    }

    public void setMarca(String marca) {
        this.marca = marca;
    }

    public String getModelo() {
        return modelo;
    }

    public void setModelo(String modelo) {
        this.modelo = modelo;
    }
}
```

Ejemplo 2B


```
package ejem1;

@Path("/coches")
public class Coche {

    static ArrayList<Car> coches = new ArrayList<Car>();
    @DefaultValue("valor por defecto") @QueryParam("valor") String text;

    @POST
    @Consumes(MediaType.APPLICATION_XML)
    @Produces(MediaType.APPLICATION_JSON)
    public Response getCar(Car coche) {
        this.coches.add(coche); // Se añade el coche a la lista
        return Response.ok(coche).build(); // Se devuelve el coche
    }

    @GET
    @Produces({ MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML })
    public ArrayList<Car> getXML() {
        Car c = new Car(); // Se crea un coche y se inicializan sus param.
        c.setMarca("Ford");
        c.setModelo("Focus");
        this.coches.add(c); // Se añade el coche a la lista
        return this.coches; // Se devuelve la lista de coches
    }
}
```

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

```
}
}
```

Vemos en los ejemplos anteriores que:

- El servicio REST no implementa ninguna interfaz ni extiende ninguna clase.
- @Path indica la URI relativa donde el servicio está disponible. Cuando accedemos al recurso raíz (en nuestro ejemplo /coches) seleccionará uno de los métodos no anotados con @Path.
- Un recurso Get es indicado con la anotación @GET en un método publico de Java (en los ejemplos textHola, htmlHola, xmlHola, jsonHola y getXML) el cual queda asociado al método HTTP GET produciendo un contenido del tipo que indique la anotación @Produces, la cual indica el tipo de representación devolver.
- Un recurso Post es indicado con la anotación @POST en un método java (en el ejemplo getCar) el cual queda asociado al método HTTP POST produciendo un contenido del tipo que indique la notación @Produces y consumiendo datos del tipo que indique la notación @Consumes. Un recurso Post puede o no devolver algún valor.
- Un ejemplo de datos validos XML para Post en el ejemplo anterior seria:

```
<car>
  <marca>Ford</marca>
  <modelo>Focus</modelo>
</car>
```


Ya que el método esta anotado con la anotación consumes y esta tiene el valor: MediaType.APPLICATION_XML.

Una vez arrancado el servidor podemos acceder a los recursos suministrados por el servicio web mediante diferentes clientes Rest:

- Un navegador web. Accediendo a las URLs que según el ejemplo desarrollado son:

Versión 2.x Jersey
http://localhost:8080/tema5/rest/hola http://localhost:8080/tema5/rest/coches
Versión 3.x Jersey
http://localhost:8080/tema5maven/rest/hola http://localhost:8080/tema5maven/rest/coches

Desde ahora solo se indicarán los datos del ejemplo de la versión 3.x de jersey

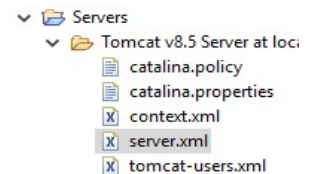
	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Se puede observar que las URIs se componen de las siguientes partes:

- o **Protocolo**: protocolo usado en el servicio web. Rest está basado en el protocolo http.
- o **Servidor**: en este caso es nuestra máquina local → localhost
- o **Puerto**: puerto que utiliza, por defecto, el servidor tomcat es: 8080.

Este se puede cambiar en el fichero servers.xml, que se puede encontrar en el directorio servers del Project Explorer, en la línea:

```
<Connector connectionTimeout="20000" port="8080"
            protocol="HTTP/1.1" redirectPort="8443"/>
```



- o **Nombre del proyecto**: tema5maven (puede contener mayúsculas).
 - o **Base del servicio**: /rest/ (se puede cambiar en el descriptor de despliegue: web.xml).
 - o **El @path indicado en el fichero Java**: en los ejemplos anteriores /hola y /coches.
- Para ver la definición de todos los recursos disponibles se puede usar la siguiente dirección:

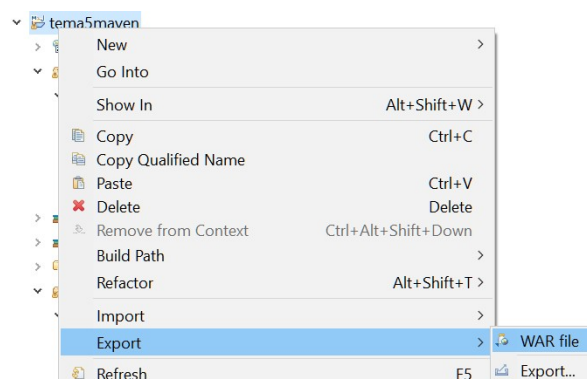
<http://servidor:puerto/proyecto/baseServicio/application.wadl>

Que en nuestro ejemplo seria:

<http://localhost:8080/tema5maven/rest/application.wadl>

- Un addon para navegador web que permita personalizar las peticiones HTTP. Por ejemplo Rester: <https://addons.mozilla.org/es/firefox/addon/rester/> (también tiene versión para Chrome)
- Programando un cliente REST (que veremos más adelante).

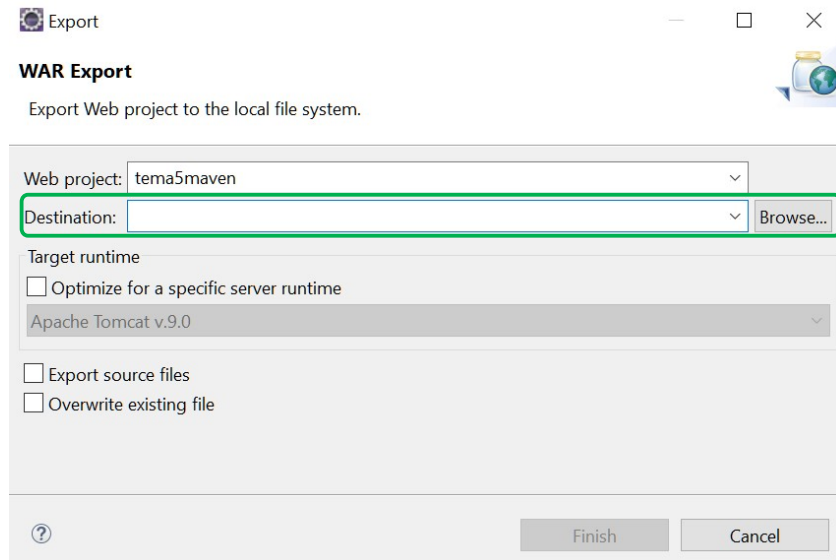
Si queremos establecer el servicio como permanente en Tomcat deberemos generar un fichero WAR²⁸ y desplegarlo en Tomcat. Eclipse lo puede realizar de forma automática por nosotros escogiendo la opción export sobre un proyecto:



²⁸ [http://es.wikipedia.org/wiki/WAR_\(archivo\)](http://es.wikipedia.org/wiki/WAR_(archivo))

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					


En la ventana emergente tendremos que indicar el destino de la exportación, que será la carpeta webapps del directorio donde tengamos instalado Tomcat.



3.3.Anotaciones

JAX-RS incluye un conjunto de anotaciones para especificar el mapeo entre las URIs de los recursos y los métodos HTTP con los métodos Java de una clase. Además también proporciona anotaciones adicionales para extraer información de una solicitud.

Anotación	Descripción
@Path	<ul style="list-style-type: none"> Esta anotación se encarga de definir una URI relativa de una clase o el método que sirve una petición. Puede usarse tanto a nivel de clase como a nivel de método. Las URIs de los métodos son relativas a la URI de la clase. Ejemplo de @Path: /holaMundo
Métodos HTTP	
@GET	<ul style="list-style-type: none"> El método java anotado con esta anotación va a procesar peticiones HTTP GET devolviendo una representación del recurso apuntado por la URI (es una operación de lectura). Es idempotente y seguro (no modifica el estado del recurso).
@POST	<ul style="list-style-type: none"> El método java anotado con esta anotación va a procesar peticiones POST. Crear, o modifica, un recurso a partir de los datos enviados en la petición HTTP ubicándolo en la URI indicada en la petición. No es idempotente (repetir un método POST contra la misma URI no es lo mismo que hacerlo una única vez) y no es seguro.

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

@PUT	<ul style="list-style-type: none"> • El método java anotado con esta anotación va a procesar peticiones PUT. • Esta anotación se encargar de crear o actualizar un recurso (del servidor) apuntado por el URI empleado los datos del cuerpo de la petición HTTP. • Es idempotente (repetir varias veces un PUT sobre la misma URI con los mismos parámetros y cuerpo de mensaje, tiene el mismo efecto que hacerlo una única vez) y no es seguro.
@DELETE	<ul style="list-style-type: none"> • El método java anotado con esta anotación va a procesar peticiones DELETE. • Elimina un recurso (del servidor) referenciado por el URI. • Es idempotente y no seguro.
@HEAD	<ul style="list-style-type: none"> • Implementado automáticamente llamando al método etiquetado con @GET e ignorando la respuesta.
Especificación del formato de las representaciones	
@Produces	<ul style="list-style-type: none"> • Se utiliza para especificar los tipos de contenido MIME de un recurso que pueden producirse y enviarse de vuelta a un cliente por un método anotado con @GET. Será el cliente quien especifique sus preferencias con el parámetro de cabecera Accept. • Ejemplos: "application/xml", "application/json".
@Consumes	<ul style="list-style-type: none"> • Indica los tipos MIME de un recurso que se consume y que son enviadas por un cliente. • Ejemplos: "application/xml", "application/json".
Mapeo de parámetros de URIs y peticiones HTTP	
@PathParam	<ul style="list-style-type: none"> • Se usa para asignar a los parámetros de los métodos valores de parámetros que forman parte de la URI. • Por ejemplo en la URI /persona/nombre/xabi la variable nombre tiene asignado el valor de xabi.
@QueryParam	<ul style="list-style-type: none"> • Permite extraer parámetros incluidos en las queries de las URIs que vengan a través de una petición GET y asociarlos con un método java. • Por ejemplo http://www.example.org?nombre=pablo la variable nombre tiene asignado el valor de pablo.
@HeaderParam	<ul style="list-style-type: none"> • Permite extraer valores enviados dentro de una cabecera HTTP y asociarlos al parámetro de un método.
@FormParam	<ul style="list-style-type: none"> • Permite extraer valores de los datos enviados en el cuerpo de peticiones POST como pares atributos-valor. Si se usa con formularios deberemos usar el tipo mime: "application/x-www-form-urlencoded"

Los métodos con anotaciones:

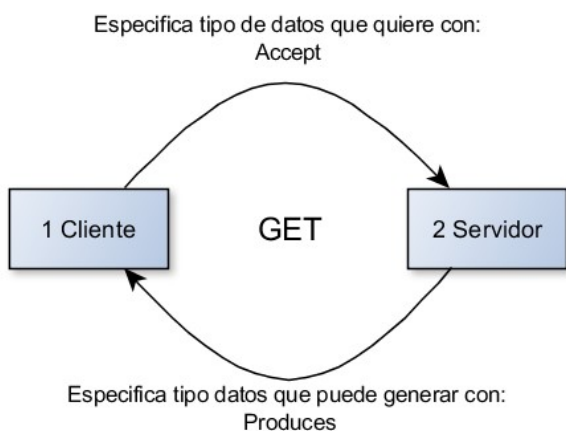
- Pueden retornar void, un tipo Java, un objeto o un objeto de tipo `jakarta.ws.rs.core.Response`.
- `Response` es usado cuando es necesario devolver mas información, metadatos, en la respuesta.

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	Desenvolvemiento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

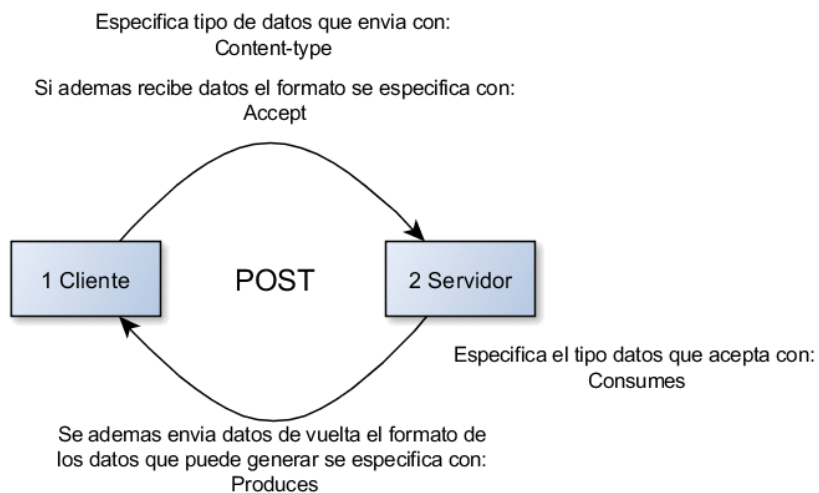
Cliente Realiza conexión con servidor usando métodos http			Servidor Recibe una conexión. Usa anotaciones		
	Acción	Parámetro cabecera		Acción	Anotación
Get	Recibe datos	Accept	@GET	Recibe datos	@Produce
	Envía datos*	Content-type		Envía datos*	@Consumes
Post	Recibe datos*	Accept	@POST	Recibe datos	@Produce
				Envía datos*	@Consumes

* En el caso de que el método retorne un valor

El cliente pide datos al servidor



El cliente envia datos al servidor



3.4.Serializado y deserializado XML

JAX-RS soporta el serializado y deserializado a XML y JSON mediante JAXB²⁹ para los tipos MIME text/xml, application/xml y application/json:

- La clase que queremos que se pueda serializar como un recurso XML o JSON se tiene que anotar al inicio de la declaración de la misma mediante `@XmlElement`.
- Si se retorna algún objeto de esa clase y se etiqueta con los tipos `@Produces({MediaType.TEXT_XML, MediaType.APPLICATION_JSON})` será serializado de forma automática.

3.5.@PathParam: Parámetros en la URI


@PathParam nos permite extraer parámetros de una URI.

```

@GET
@Path("/modelos/{modelo}")
public String quienEres(@PathParam("modelo") String name) {
    return "Hola, soy un " + name;
}

```

²⁹ <https://es.wikipedia.org/wiki/JAXB>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Donde

- El patrón {modelo} implica un parámetro de la URI.
- A través de la anotación `@PathParam` asociamos el patrón con un parámetro del método.
- Si accedemos a <http://localhost:8080/tema5maven/rest/coches/modelos/saxo> deberíamos obtener:

Hola, soy un saxo

3.6. @QueryParam: Parámetros en las peticiones

La anotación `@QueryParam` es muy similar al anterior y permite extraer información de la parte de la pregunta de una URI.

```
@GET
@Path("coche")
public String marcaYModelo(@QueryParam("marca") String marca,
                           @QueryParam("modelo") String mo) {
    return "La marca es " + marca + " y el modelo " + mo;
}
```

Donde:

- Mediante la anotación `@QueryParam` los parámetros marca y modelo de la parte de la query de la URI se asocian respectivamente con los atributos marca y modelo del método (marcaYModelo en el ejemplo).
- En <http://localhost:8080/tema5maven/rest/coches/coche?marca=Citroen&modelo=Saxo> deberíamos obtener:

La marca es Citroen y el modelo Saxo

3.7. @DefaultValue: Valores por defecto

Se puede usar la anotación `@DefaultValue` para establecer un valor por defecto para los parámetros en el caso de que estos no se encuentren presentes.

```
@GET
@Path("cocheDefecto")
public String marcaYModeloD(
    @DefaultValue("Citroen") @QueryParam("marca") String marca,
    @DefaultValue("Saxo") @QueryParam("modelo") String mo ) {
    return "La marca es " + marca + " y el modelo " + mo;
}
```

Donde:

- `@DefaultValue` establece el valor por defecto
- Accediendo a: <http://localhost:8080/tema5maven/rest/coches/cocheDefecto?marca=Citroen> obtendremos:

La marca es Citroen y el modelo Saxo

Al no estar presente el valor del modelo se le asigna su valor por defecto.

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

En el siguiente ejemplo combinaremos los tres tipos de parámetros anteriores:

```
@GET
@Path("/todo/{marca}")
public String todo (@PathParam("marca") String marca,
                    @DefaultValue("Saxo") @QueryParam("modelo") String mo) {
    return "La marca es " + marca + " y el modelo " + mo;
}
```

Accediendo a:

<http://localhost:8080/tema5maven/rest/coches/todo/citroen?modelo=c3>

obtendremos:

La marca es citroen y el modelo c3

También podemos usar @DefaultValue para establecer valores por defecto a atributos de clase:

```
@DefaultValue("valor por defecto") @QueryParam("valor") String text;
```

3.8.@FormParam: Extraer información de un formulario

@FormParam permite extraer información de un formulario HTML. Tenemos el siguiente formulario:

```
<html>
  <body>
    <form method="POST" action="http://localhost:8080/tema5maven/rest/coches">
      <h2>Formulario</h2>
      <label>Marca: </label> <input type="text" name="marca" /><br><br>
      <label>Modelo: </label> <input type="text" name="modelo" /><br><br>
      <input type="submit" value="Enviar consulta" />
    </form>
  </body>
</html>
```

La petición generada por el formulario anterior será procesada por el método:

```
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces( MediaType.APPLICATION_JSON )
public Car getCarText(@FormParam("marca") String marca,
                      @FormParam("modelo") String model) {
    Car coche=new Car();
    coche.setMarca(marca);
    coche.setModelo(model);
    return coche;
}
```

Produciendo una salida JSON. Imaginemos que hemos introducido, en los campos del formulario, la marca Seat y el modelo Ibiza. La salida producida será:

```
{"marca":"Seat","modelo":"Ibiza"}
```

<div>COLEXIO</div> <div>VIVAS S.L.</div>	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

3.9. @HeaderParam: Parámetros en las cabeceras

Se puede usar la anotación @HeaderParam para extraer información de las cabeceras HTTP.

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String getHeader(@HeaderParam("Accept") String tipoMime) {
    return "Tipo mime: "+tipoMime;
}
```

Donde @HeaderParam indica el parámetro de la cabecera HTTP que se asocia con el parámetro que le sigue. En el ejemplo anterior se asigna el parámetro de la cabecera HTTP Accept a la variable tipoMime.

3.10. @XmlAttribute: Atributos XML

Si queremos que durante el serializado de una clase a XML un atributo de la clase se convierta en un atributo XML usaremos la anotación @XmlAttribute antes del método get que devuelve ese dato:

```
@XmlAttribute
public String getMarca() {
    return marca;
}
```

Siendo el xml devuelto:

```
<car marca="Ford" >
    <modelo>Focus</modelo>
</car>
```

Si la representación devuelta es JSON @XmlAttribute es ignorada.

Si se necesita cambiar el nombre del atributo, el nuevo nombre se puede especificar mediante la clave name. En este caso el nombre de la marca se cambiara a mimarca tanto en XML como en JSON:

```
@XmlAttribute(name = "mimarca")
```

3.11. @XmlElement: Modificando la estructura de los datos


Si una vez creada la estructura de datos, si tenemos que cambiar el nombre de un elemento devuelto se usará la anotación @XmlElement al método get que devuelve ese dato.

Por ejemplo, el método getMarca de la clase Car retorna una cadena cuyo nombre es marca. Si deseamos que en vez de marca devuelva un atributo que se llame nuevaMarca deberemos añadir la anotación:

```
@XmlElement(name = "nuevaMarca") // nombre del valor que devuelve
public String getMarca() {
    return marca;
}
```

Donde name indica el nuevo valor que tendrá atributo devuelto. El resultado retornando seria:

```
[{"modelo": "Saxo", "nuevaMarca": "Citroen"}, {"modelo": "Focus", "nuevaMarca": "Ford"}]
```


	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

3.12. Response³⁰ y ResponseBuilder³¹

Toda petición que se hace a un recurso debe devolver algo, como puede ser un objeto. Sin embargo a veces es necesario devolver información adicional en respuesta a una petición HTTP. Dicha información puede ser construida y devuelta utilizando Response y Response.ResponseBuilder.

Si vemos el método getCar del ejemplo:

```
@POST
@Consumes(MediaType.APPLICATION_XML)
@Produces( MediaType.APPLICATION_JSON)
public Response getCar(Car coche) {
    this.coches.add(coche);
    return Response.ok(coche).build();
    // return Response.status(Status.BAD_REQUEST).build();
}
```

Devolvemos un objeto jakarta.ws.rs.core.Response que contiene el estado de la petición: si está OK, si hay errores, si no responde, etc. Todos los errores disponibles para HTTP están contenidos en ese objeto.

Métodos principales de Response:

- ok: Crea un nuevo ResponseBuilder con el estado Ok (respuesta correcta).
- created: Crea un nuevo ResponseBuilder para un recurso creado, poniendo la localización en la cabecera.
- status: Crea un nuevo ResponseBuilder con el estado³² suministrado.
- ...

Operaciones principales de ResponseBuilder:

- build: Construye un Response a partir de un ResponseBuilder.
- location: Establece la localización.
- header: Añade una cabecera explícitamente.
- status: Establece el estado de un ResponseBuilder.
- type: Añade el tipo de medio:
- ...

³⁰ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/latest/jersey/com/sun/research/ws/wadl/Response.html>

³¹ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/core/Response.ResponseBuilder.html>

³² Ver códigos de estado HTTP: <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/core/Response.Status.html>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Para personalizar el mensaje de un Response podemos usar su método entity:

- Mediante un objeto:

```
Response.status(Status.BAD_REQUEST).entity(coche).type(MediaType.TEXT_XML).build();
```

- Mediante una cadena de texto:

```
Response.status(Status.BAD_REQUEST).entity("Error").type(MediaType.TEXT_PLAIN).build();
Response.status(Status.NO_CONTENT).entity("Sin datos").type("text/plain").build();
Response.status(Status.NOT_FOUND).entity("No se ha encontrado el registro").build();
```

El formato XML presenta problemas ante la devolución en un Response de listas de valores (como puede ser un ArrayList). Para solucionar esto usamos la clase GenericEntity<T>³³.

```
List<String> list = new ArrayList<String>();
GenericEntity<List<String>> entity = new
GenericEntity<List<String>>(list) {};
Response response = Response.ok(entity).build();
```

3.13. Ficheros binarios

A parte de devolver recursos en texto plano: ficheros en texto plano, XML, JSON, HTML, ... JAX-RS puede devolver datos binarios:

Para ello debe especificar el mediatype adecuado en la anotación @Produces.

Los mediatype principales³⁴ de objetos binarios son:

- application/pdf
- application/zip
- audio/mpeg
- image/gif
- image/jpeg
- image/png
- video/avi
- video/mpeg
- video/mp4
- video/webm
- video/x-flv
- video/x-ms-wmv


La clave es devolver en el response un flujo FileInputStream al fichero deseado.

Un ejemplo de utilización:

```
@GET
@Produces("image/jpg")
public Response img() {
    File file=new File("c:/imagen.jpg");
    try {
        return Response.ok(new FileInputStream(file)).build();
    } catch (FileNotFoundException e) {
        return Response.status(Status.BAD_REQUEST).entity("Error").
            type(MediaType.TEXT_PLAIN).build();
    }
}
```

³³ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/index.html?javax/ws/rs/core/package-summary.html>

³⁴ http://en.wikipedia.org/wiki/Internet_media_type

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

3.14. Gestión de excepciones

Jersey nos proporciona una interfaz muy sencilla que nos permite capturar las excepciones que definamos y devolver un mensaje encapsulándolas de forma que podamos construir una respuesta con la propia traza o lo que creamos oportuno.

```
@POST
@Consumes("application/x-www-form-urlencoded")
@Produces(MediaType.APPLICATION_JSON)
public Car getCarText(@FormParam("marca") String marca,
                      @FormParam("modelo") String model) {
    if (marca.length()==0 || model.length()==0)
        throw new WebApplicationException(Response.Status.NO_CONTENT);
    Car coche=new Car();
    coche.setMarca(marca);
    coche.setModelo(model);
    this.coches.add(coche);
    return coche;
}
```

3.15. Links

La clase Link³⁵ añade soporte para añadir enlaces como parámetros en las cabeceras HTTP, por ejemplo, si queremos agregar información adicional sobre algún recurso podemos insertar un enlace a esa información adicional. Por ejemplo imaginemos que estamos consumiendo la descripción de un libro y nos interesa insertar enlaces a recursos adicionales del libro como puede ser el autor, el género del libro, la colección, ...

```
Response response = Response.ok().
    link("http://example.com/Niven", "autor").
    link("http://example.com/CienciaFiccion", "genero").
    build();
return response;
```

Donde en las cabeceras HTTP tendríamos los parámetros:

Link: <http://example.com/Niven>; rel="autor", <http://example.com/CienciaFiccion>; rel="genero"


3.16. Creación de URIs

Un aspecto muy importante de REST son los hipervínculos. En las representaciones de un recurso los clientes pueden usar los hipervínculos para navegar entre las representaciones. Esto se conoce también como Hypermedia As The Engine Of Application State (HATEOAS).

UriInfo³⁶ se puede utilizar para acceder a la información de la URI de una petición. UriInfo se obtiene usando la anotación @Context: @Context UriInfo uriInfo.

³⁵ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/core/Link.html>

³⁶ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/core/UriInfo.html>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Imaginemos realizamos la siguiente petición:

<http://localhost:8080/tema5maven/rest/coches?id=5>

Los siguientes métodos de `uriInfo` devolverán:

- `getBaseUri`: Devuelve el path base de la petición.
<http://localhost:8080/tema5maven/rest/>
- `getAbsolutePath`: Devuelve el path absoluto de la petición.
<http://localhost:8080/tema5maven/rest/coches>
- `getPath`: Consigue el path de la petición actual relativo a la URI base.
coches
- `getQueryParameters`: Obtiene la parte de la consulta de la URI actual.
{ id=[5]}
- `getRequestUri`: Consigue la URI absoluta de la petición actual.
<http://localhost:8080/tema5maven/rest/coches?id=5>

JAX-RS usa la clase `UriBuilder`³⁷ para la construcción de URIs seguras ya sea desde cero o desde URIs ya existentes.

Los siguientes son ejemplos que muestran la construcción de URIs:

- A partir del path relativo de la clase:

```
UriBuilder uriBuilder=UriBuilder.fromResource(Coche.class);
```
- A partir de un path dado usando patrones. La primera se resuelve con `resolveTemplate` y la segunda con `build` (método que construye la URI a partir de `UriBuilder`). Al método `build` se le pueden pasar, en orden, los valores de las patrones que se han de asignar.

```
URI uri=UriBuilder.fromPath("http://localhost/{arg1}/{arg2}").  
    resolveTemplate("arg1", "para3").build("para4");
```
- Indicando que el path base va a ser el indicado por la anotación `path` del método `quienEres` de la clase `Coche` (`@Path("modelos/{modelo}")`). Se le indica el protocolo, equipo y puerto de conexión. Con `build` se le asigna un valor ya que el path del método usa un patrón.


```
URI uri=UriBuilder.fromMethod(Coche.class, "quienEres").scheme("http")  
    .host("localhost").port(8080).path(coche.getModelo()).build("coche");
```
- Creamos la URI a partir de la ruta absoluta añadiéndole un path y una query.

```
UriBuilder ub=uriInfo.getAbsolutePathBuilder();  
URI uri2=uriInfo.getAbsolutePathBuilder().path("{coche}").  
    queryParams("modelo", "{value}").build("renault", "megane");
```

Para consumirlo desde nuestro cliente java usaríamos:

```
final Response response = target.request().get();  
URI autor = response.getLink("autor").getUri();  
URI genero = response.getLink("genero").getUri();  
return response;
```

³⁷ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/core/UriBuilder.html>

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

4. Apéndice I. Creación de un cliente Rest³⁸

En este punto mostraremos el uso de la API de cliente JAX-RS. Es una API basada en Java para la comunicación con los servicios Web RESTful.

Un recurso en la API de cliente JAX-RS es una instancia de la clase WebTarget³⁹ que encapsula una URI. Las representaciones son tipos Java y pueden contener enlaces a nuevas instancias de WebTarget.

Para empezar construiremos una instancia de Client.

```
Client client = ClientBuilder.newClient();
```

ClientBuilder es una API de JAX-RS usada para crear nuevas instancias de cliente. ClientBuilder⁴⁰ se puede utilizar además para configurar las propiedades adicionales, como pueden ser opciones de SSL, ...

Una vez creada la instancia de cliente se puede crear un WebTarget sobre ella.

```
WebTarget target = client.target("http://localhost:8080/tema5maven/rest/coches");
```

En el ejemplo anterior el URI pasado al método es la URI del recurso web de destino. En escenarios más complejos podría ser la URI raíz de toda la aplicación REST, de la que se pueden derivar instancias de WebTarget que representan recursos individuales y configurar estos de forma individual.

Cada instancia WebTarget hereda la configuración de su padre y puede ser configurado a medida sin afectar a la configuración del componente padre.

Supongamos que tenemos un WebTarget en la URI <http://localhost:8080/tema5maven/rest> que representa la raíz de contexto de una aplicación REST y hay un recurso expuesto en la URI <http://localhost:8080/tema5.prueba1/rest/coches>. Como ya se ha mencionado, de una instancia de WebTarget se pueden derivar otros WebTarget. Por ejemplo:

```
WebTarget webTarget = client.target("http://localhost:8080/tema5maven/rest/");
WebTarget newTarget = webTarget.path("coches");
```

Donde newTarget apunta al recurso <http://localhost:8080/tema5maven/rest/coches>. Si configuramos newTarget con un filtro específico para el recurso (no los vamos a ver), no influirá en la instancia webTarget original. Sin embargo, si el webTarget original tiene algún filtro propio este será heredado por newTarget ya que se ha creado a partir de webTarget. Este mecanismo permite compartir la configuración común de recursos relacionados (normalmente alojados bajo la misma raíz URI, en nuestro caso representado por la instancia webTarget), al tiempo que permite una mayor especialización de configuración basada en las necesidades específicas de cada recurso individual. Los mismos principios de configuración de herencia (para permitir la propagación de la configuración común) y desacoplamiento (para permitir la personalización de configuración individuales) se aplica a todos los componentes de JAX-RS API Client que se analizan a continuación.

³⁸ Traducción libre de: <https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/client.html>

³⁹ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/client/WebTarget.html>

⁴⁰ <https://eclipse-ee4j.github.io/jersey.github.io/apidocs/2.22/jersey/javax/ws/rs/client/ClientBuilder.html>

COLEXIO VIVAS S.L.	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Imaginemos que tenemos un subrecurso en el path "<http://localhost:8080/tema5maven/rest/coches/modelos>". Se puede derivar un WebTarget para este recurso a partir de:

```
WebTarget targetMarca = newTarget.path("marca");
```

Vamos a suponer que el recurso marca acepta un parámetro de consulta para las solicitudes GET. El siguiente fragmento de código muestra un código que crea un nuevo WebTarget con el parámetro de consulta definida:

```
WebTarget targetMarcaParam = targetMarca.queryParam("Marca", "Renault");
```

Ahora vamos a centrarnos en la invocación de una petición HTTP GET. Para empezar a construir una nueva petición HTTP tenemos que crear una nueva Invocation.Builder.

```
Invocation.Builder invocationBuilder =  
    targetMarcaParam.request(MediaType.APPLICATION_XML);  
invocationBuilder.header("algunaCabecera", "true");
```

Se crea una nueva instancia de Invocation.Builder usando request. Aparte de request WebTarget proporciona métodos los cuales, entre otros, nos permiten definir el tipo de medio de la representación solicitada a devolver a partir del recurso. Aquí estamos diciendo que solicitamos un tipo "application/xml". Esto indica a Jersey que añada la cabecera HTTP Accept con valor: application/xml a nuestra solicitud.

Se usa invocationBuilder para configurar parámetros específicos de solicitud. En nuestro ejemplo agregamos el parámetro algunaCabecera con valor verdadero a la cabecera HTTP.

Una vez personalizada la solicitud debemos invocarla. En caso de que no se quiera hacer ningún procesamiento por lotes en sus invocaciones realizaremos las peticiones directamente mediante una instancia de Invocation.Builder.

```
Response response = invocationBuilder.get();
```

Aunque breve, el código del ejemplo realiza múltiples acciones. En primer lugar, se va a construir la petición desde invocationBuilder. La URI de solicitud será "<http://localhost:8080/tema5maven/rest/coches/marca?Marca=Renault>" y la solicitud contendrá en las cabeceras HTTP: algunaCabecera:true y Accept:application/xml y será enviada al recurso remoto. Digamos que el recurso a continuación, devuelve un mensaje con código HTTP 200 y un response con un contenido XML que contendrá el valor devuelto por el recurso Marca. Ahora podemos observar la respuesta devuelta:

```
System.out.println(response.getStatus());  
Car coche=response.readEntity(Car.class);  
System.out.println(String.format("Marca %s y modelo %s",  
                                coche.getMarca(),coche.getModelo()));
```

Que producirá la siguiente salida a la consola:

```
200  
Marca Audi y modelo A4
```

Como podemos ver, la solicitud se ha procesado satisfactoriamente (código 200).

<div>COLEXIO</div> <div>VIVAS</div> <div>S.L.</div>	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

Si utilizamos el estilo de programación fluido que proporciona la API el código anterior se puede escribir de forma más compacta de la siguiente manera:

```
Client client = ClientBuilder.newClient();
Car coche = client
    .target("http://localhost:8080/tema5maven/rest/")
    .path("coches/marca")
    .queryParams("Marca", "Renault")
    .request(MediaType.APPLICATION_XML)
    .header("algunaCabecera", "true")
    .get(Car.class);

System.out.println(String.format("Marca %s y modelo %s",
                                coche.getMarca(),coche.getModelo()));

JsonReader jsonReader=Json.createReader(new StringReader(targetMarca.
    request(MediaType.APPLICATION_JSON).get(String.class)));

JsonObject obj= jsonReader.readObject();
jsonReader.close();
System.out.println(String.format("Mar: %s  Mod: %s",
                                obj.getString("marca"),obj.getString("modelo")));
```

Imaginemos que queremos recuperar el código HTML de una página web mediante una petición GET:


```
@GET
@Produces(MediaType.TEXT_HTML)
@Path("web")
public String web(){
    Client client = ClientBuilder.newClient();
    Response res=client.target("http://www.google.es").
        request(MediaType.TEXT_HTML).get();

    String cad=res.readEntity(String.class);
    System.out.println(res.getStatus()+"\n"+cad);
    return cad;
}
```

Imaginemos ahora que queremos invocar una petición POST. Podemos usar la instancia targetMarca creada anteriormente y llamar a post en lugar de get.

```
Car c2=new Car();
c2.setMarca("BMW");
c2.setModelo("Serie 3");
Response ob=new Target.request(MediaType.APPLICATION_JSON)
    .post(Entity.entity(c2,MediaType.APPLICATION_XML), Response.class);

Car coche=ob.readEntity(Car.class);
System.out.println("coche: "+coche.getMarca()+" "+coche.getModelo());
```

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos					CURSO:	2º
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

```
Form form = new Form();
form.param("marca", "Renault");
form.param("modelo", "Clio");
coche=newTarget.path("form")
    .request(MediaType.APPLICATION_JSON)
    .post(Entity.entity(form,
        MediaType.APPLICATION_FORM_URLENCODED_TYPE), Car.class);

System.out.println("coche: "+coche.getMarca()+" "+coche.getModelo());
```

4.1.Cerrando conexiones


Las conexiones se abren para cada solicitud y se cierran después de que se recibe una respuesta y una entidad se procesa.

Veamos el siguiente ejemplo:

```
final WebTarget target = ... some web target
Response response = target.path("resource").request().get();
System.out.println("Conexión aun abierta.");
System.out.println("Response de tipo cadena: " +
    response.readEntity(String.class));
System.out.println("Ahora la conexión está cerrada.");
```

Si no se lee la entidad se necesita cerrar el Response de forma manual:

```
response.close()
```

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos				CURSO:	2º	
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

5. Apéndice II: Control de errores

Una forma de depurar errores es crear un ExceptionMapper simple para atrapar las excepciones que no están mapeadas. Cuando no hay un mapeador, a menudo la excepción se propaga hasta el nivel del contenedor, lo que solo provoca el error genérico 500 del servidor (que la mayoría de las veces es de poca ayuda).

```
@Provider
public class DebugExceptionMapper
    implements ExceptionMapper<Exception> {


    @Override
    public Response toResponse(Exception exception) {
        exception.printStackTrace();
        return Response.serverError().entity(exception.getMessage()).build();
    }
}
```

Si por ejemplo a la clase Car vista en los ejemplos se le añade el siguiente constructor:

```
public Car(String marca, String modelo) {
    super();
    this.marca = marca;
    this.modelo = modelo;
}
```


El servicio web fallará ya que JAXB requiere que exista, al menos, el constructor por defecto.

Implementando la clase anterior cuando se lanza la excepción esta se imprime pudiéndose ver el mensaje de excepción

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos				CURSO:	2º	
	PROTOCOLO:	Apuntes clases		AVAL:	2	DATA:	2022/2023	
	UNIDAD		COMPETENCIA:					

6. Bibliografía

1. [Soap](#)
2. [Web Services Description Language. Usados en SOAP.](#)
3. RESTful Web Services
 1. [Servicios web de tipo REST](#)
 2. [Tutorial del REST](#)
 3. [Recursos](#)
 4. Crud: de Create, read, update y_delete
 1. [CRUD castellano](#)
 2. [CRUD inglés](#)
 5. [HTTP](#)
 6. [Mime](#)
 7. [Media type](#)
 8. [Cabeceras HTTP en Castellano](#)
 9. [Listado completo de cabeceras HTTP en Inglés](#)
 10. [Hipermedia como motor del estado de la aplicación](#)
 11. [Hypermedia as the Engine of Application State](#)
 12. [Autorización segura para APIs](#)
 13. [Uri o Identificador de Recursos Uniforme](#)
4. Tomcat
 1. [Tomcat](#)
 2. [Página web Tomcat](#)
5. [POJO](#)
6. JAX-RS: Java API for RESTful Web Services
 1. [JAX-RS](#)
 2. [Paquetes utilizados](#)
 3. [Anotación Java](#)
7. Jersey
 1. [Jersey: Implementación de referencia en java](#)
 2. [Guia de usuario](#)
 3. [API](#)
 4. [Jars de jaxrs-ri](#)
8. JAXB: Java Architecture for XML Binding (JAXB) (mapeo entre objetos Java y XMI)
 1. [JAXB](#)
 2. [Página web de JAXB](#)
 3. [Paquetes JAXB](#)
 4. [Soporte de JAXB en java 9,10, 11 y versiones posteriores](#)

	RAMA:	Informática	CICLO:	Desenvolvemento de Aplicacions Multiplataforma				
	MÓDULO:	Acceso a datos				CURSO:	2º	
	PROTOCOLO:	Apuntes clases	AVAL:	2	DATA:	2022/2023		
	UNIDAD		COMPETENCIA:					

9. [Java Servlet](#)
10. [Java Servlet](#)
11. [Descriptor de despliegue](#)
12. [Jakarta Servlet](#)
13. [Maven - guía para establecer el nombre de: groupId, artifactId y version](#)
14. [Archivo de aplicación web](#)
15. RESTER
 1. [Plugin Rester para Firefox](#)
16. Clases jax-rs
 1. [Response](#)
 2. [Response.ResponseBuilder](#)
 3. [Códigos de estado HTTP](#)
 4. [GenericEntity](#)
 5. [Link](#)
 6. [URI info](#)
 7. [UriBuilder](#)
 8. [Client API \(Traducción libre\)](#)
 9. [WebTarget](#)
 10. [ClientBuilder](#)