

**MP\_0490.**  
**Programación de servicios y procesos**  
**UF3. Generación de servicios en red**

## **3.4. Servicios web REST**

# Índice

---

☰	Objetivos	3
☰	Introducción a REST	4
☰	RESTful	6
☰	Instalar Apache Tomcat	7
☰	Funcionamiento básico de Apache Tomcat	9
☰	Tipos MIME	13
☰	Descargar la librería Jersey	14
☰	Anotaciones	17
☰	Crear y configurar el proyecto web dinámico	23
☰	Crear el modelo de datos	40
☰	Crear el controlador	41
☰	Archivo web.xml	44
☰	Desplegar el proyecto en Apache Tomcat	50
☰	Poner a prueba el servicio web	53
☰	Códigos de estado HTTP	55
☰	Servicio web que devuelve HTML	57
☰	Servicio web que devuelve XML o JSON	61
☰	Parámetros en la ruta (@PathParam)	66
☰	Parámetros fuera de la ruta (@QueryParam)	68
☰	Consumo de servicios web RESTful	70
☰	Resumen	73

# Objetivos

---

En esta lección perseguimos los siguientes objetivos:

- 1 Comprender qué es un servicio web.
- 2 Aprender a instalar Apache Tomcat.
- 3 Crear servicios web utilizando la tecnología RESTful.
- 4 Consumir servicios web RESTful.

---

¡Ánimo y adelante!

# Introducción a REST

**Un servicio web es un sistema software definido para permitir la comunicación a través de la red entre dos máquinas.**

La comunicación es posible mediante el intercambio de mensajes, utilizando, por lo general, **el protocolo HTTP**.

Importantes sitios web, como Facebook, Twitter o Instagram, son aplicaciones que hacen uso de servicios web (*web services*).

Un *web service* es, como hemos mencionado, un **conjunto de estándares y protocolos utilizados para intercambiar datos entre aplicaciones**. Y existen numerosas tecnologías relacionadas con el mundo de los servicios web.

**Veamos, a continuación, aquellas cuyo uso está más extendido:**

- **SOAP (Simple Object Access Protocol)**: protocolo que utiliza el lenguaje XML para definir el lenguaje utilizado entre dos procesos para comunicarse. SOAP emplea XML para definir un conjunto de tipos de datos, y configurar los mensajes de solicitud y de respuesta.
- **REST (Representational State Transfer)**: utiliza el protocolo HTTP para la comunicación entre máquinas. Puesto que utiliza el protocolo HTTP, que se caracteriza por no tener estado, los servicios REST corresponden al tipo de servicios *stateless* (sin estado); es decir, que en cada solicitud que llega de un cliente, el servidor no tiene información de las solicitudes anteriores del mismo cliente.

REST es una tecnología altamente utilizada hoy en día en los sistemas distribuidos que se comunican a través de Internet.



## Principios en los que se basa REST

- Utiliza el protocolo **HTTP** para establecer comunicaciones cliente/servidor.
- El cliente solicita recursos al servidor a través de una **URI (Uniform Resource Identifier)** que sirve para identificar inequívocamente un recurso dentro de la red.
- Los datos se transmiten con un formato específico identificado mediante la nomenclatura **MIME** (image/jpeg, text/html, text/xml, video/mpeg, etc). Más adelante hablaremos más en profundidad de estos tipos MIME.
- REST utiliza las acciones estándar definidas por el protocolo HTTP: **GET, POST, PUT, DELETE**, etc.
- El servidor **expone recursos** que podrán ser devueltos en distintos formatos como **texto, HTML, XML, JSON**, etc.
- Para cada solicitud que realiza un cliente se obtiene un **código de respuesta**, una característica propia del protocolo HTTP. Entre estas respuestas, seguro que ya te has acostumbrado a ver el código **404** (recurso no encontrado).

# RESTful

---

Cuando hablamos de REST hablamos de un tipo de arquitectura de software para el desarrollo de servicios web.

**RESTful**, sin embargo, es algo más concreto. Denominamos así a los servicios web que siguen los principios que rigen la arquitectura REST.

Los servicios RESTful presentan las siguientes ventajas:

- **Separación de los datos del recurso y su representación.** El mismo recurso puede enviarse representado en formato XML, JSON o texto plano.
- **Visibilidad:** se accede a cada recurso a través de una URI, de manera muy simple.
- **Seguridad:** el acceso a un recurso nunca cambiará su estado.
- **Escalabilidad:** los servicios web REST son fácilmente escalables, lo que significa que pueden crecer a lo largo del tiempo sin que suponga un coste muy alto.
- **Rendimiento:** el rendimiento hace referencia al tiempo medio de demora en responder a una petición de un cliente. Los servicios REST, precisamente por utilizar un protocolo sin estado, son rápidos.

# Instalar Apache Tomcat

**Los servicios web deben estar enmarcados dentro de un servidor web o un servidor de aplicaciones.**

Para los ejemplos que vamos a desarrollar utilizaremos Apache Tomcat.



**RECUERDA:** Apache Tomcat, IIS y WildFly son servidores que proporcionan páginas web a los clientes que las solicitan desde un navegador web como Internet Explorer, Google Chrome, Firefox, etc. El protocolo de nivel de aplicación que utiliza Internet para el tráfico de páginas web es el protocolo HTTP (*Hipertext Transfer Protocol*).

En esta ocasión, Apache Tomcat proporcionará los servicios web RESTful a los clientes que los soliciten.

Desde aquí, descargaremos la versión 9 ejecutable de Apache Tomcat como servidor web. De este modo se instalará como un servicio más de Windows, y podremos iniciarla y detenerla cuando sea necesario.

Puedes descargar Apache Tomcat gratuitamente desde su web oficial:

APACHE TOMCAT

## Pasos para instalar Apache Tomcat

La instalación de Apache Tomcat consta de varios pasos, por los que irás avanzando simplemente pulsando el botón “Next”. Pero hay dos de ellos a los que merece la pena prestar más atención.

- **Donde se nos solicita la ubicación de la instalación de Java:** Apache Tomcat requiere que esté instalado Java. En la gran mayoría de los casos el instalador detecta la ubicación de la instalación de Java sin mayor problema, pero en algunas ocasiones puede ser necesario especificarla.
- **Donde aparecen los valores de configuración del servidor:** presta atención al valor de "HTTP/1.1 Connector Port", que será 8080. Recuerda que una petición a un servidor se realiza a través del nombre del servidor y un puerto. Pues bien, el nombre del servidor, al tratarse de nuestro propio equipo, será *localhost* y el puerto el **8080**, que es el valor que aparece por defecto.

---

El siguiente [vídeo](#) muestra cómo descargar e instalar Apache Tomcat en tu equipo.

# Funcionamiento básico de Apache Tomcat

Cuando escribes una dirección web en tu navegador realizas una petición a un servidor web que localizará ese recurso (en este caso, una página web) y te lo devolverá (en formato HTML) para que tu navegador pueda interpretarlo y mostrarlo.

Por ejemplo: si escribimos en el navegador <https://www.casadellibro.com/> estamos realizando una petición a un servidor web, que localiza el recurso (que en nuestro ejemplo es una página web) que estamos solicitando, y nos lo devuelve (en este caso, en formato HTML) para que podamos verlo en el navegador.

Este es el funcionamiento básico del protocolo HTTP y el principio por el que se rigen también los servicios web REST. El sitio web de la Casa del Libro estará localizado en un servidor web instalado en un equipo de cualquier parte del mundo, y miles de clientes pueden acceder a su contenido.

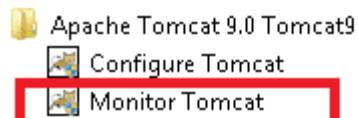
## Cómo funciona Apache Tomcat

Apache Tomcat te brinda la posibilidad de crear un servidor web local para poner a prueba los ejercicios que desarrollaremos en esta unidad. Veamos, por ejemplo, cómo iniciar y detener este servidor web, y cómo guarda los recursos.

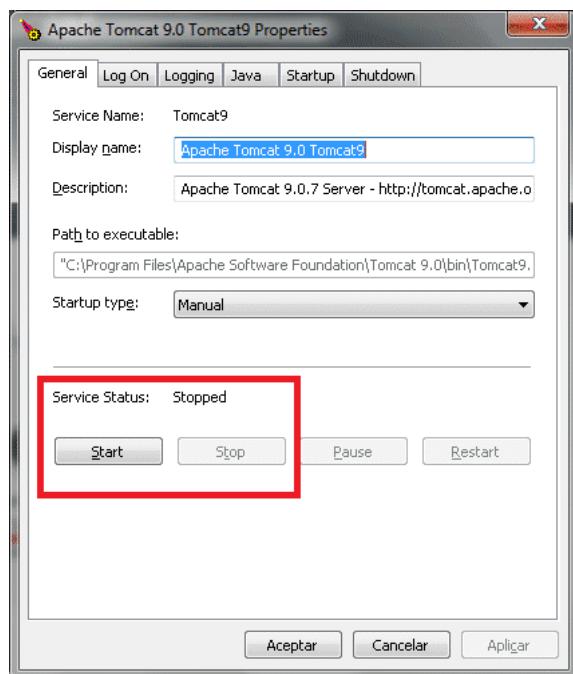
1

## Iniciar y detener Apache Tomcat.

Para iniciar y detener el servicio debes ejecutar el programa "**Monitor Tomcat**", que encontrarás en el grupo de programas de "Apache Tomcat 9".



Se abrirá el siguiente cuadro de diálogo:



Puedes utilizar los botones *Start* y *Stop* para iniciar y detener el servicio. Cuando el servicio esté iniciado, podrás acceder a la dirección <http://localhost:8080> en tu navegador para ver la página principal de administración de Apache Tomcat.

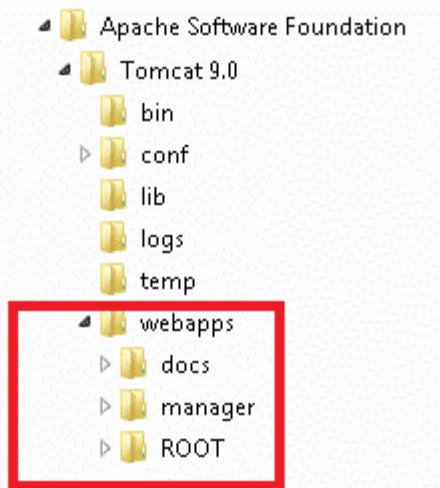
**(i)** ¡Ojo! Si el programa "**Monitor Tomcat**" no se abre, prueba a ejecutarlo como administrador haciendo clic derecho sobre el acceso al programa, y seleccionando "**Ejecutar como administrador**" en el menú contextual.

2

## Guardar los recursos en Apache Tomcat.

Apache Tomcat almacena los diferentes recursos (páginas web, imágenes y otros) en carpetas que llamaremos **aplicaciones web o webapps**. Para cada nueva aplicación, tiene que existir la correspondiente carpeta dentro de la ubicación "Apache Software Foundation/Tomcat 9.0/webapps".

Como mínimo, aparecerán tres carpetas que se crean durante la instalación.

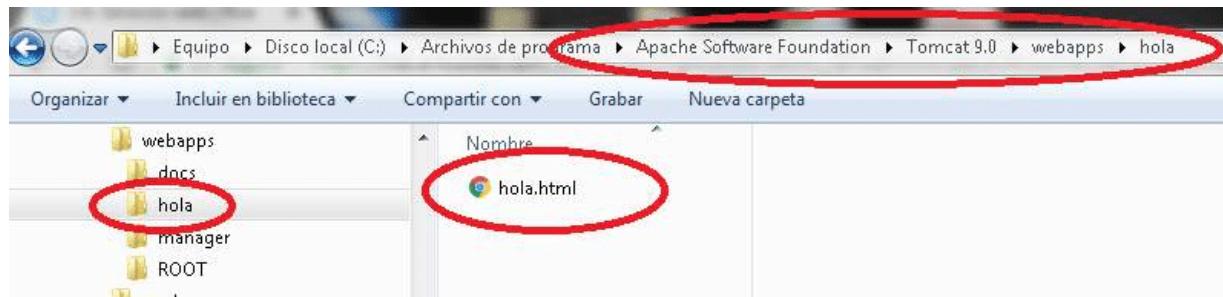


Carpeta *webapps* de Apache Tomcat que aloja las aplicaciones web.

### Hagamos una prueba:

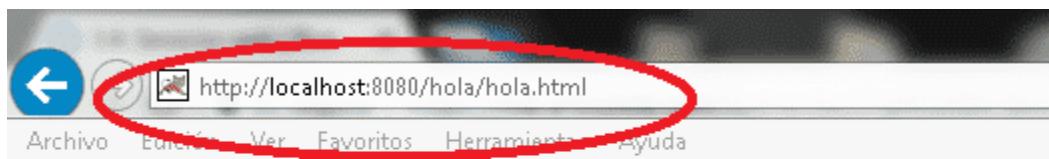
Dentro de la carpeta *webapps*, crea una subcarpeta llamada *hola* y, dentro de esta, un archivo denominado *hola.html* con el siguiente código HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Mi pagina hola mundo</title>
  </head>
  <body>
    <h1>Hola Mundo</h1>
  </body>
</html>
```



Nueva aplicación web, llamada *hola*, alojada en el servidor web Apache Tomcat.

Si el servicio de Apache Tomcat está iniciado, podrás acceder a la URL del nuevo sitio web y mostrar el nuevo recurso en el navegador.



## Hola Mundo

Cliente solicitando el recurso hola.html.

# Tipos MIME

Un concepto muy vinculado a los servicios REST son los denominados tipos MIME.

Los tipos MIME son unos códigos que utilizamos durante la implementación de los servicios web para indicarle al cliente cuál es el formato del mensaje que le hemos enviado.

A continuación, te mostramos un listado con los tipos MIME más utilizados, que seguro conoces en su mayoría.

- audio/mp4
- audio/mpeg
- image/gif
- image/bmp
- image/png
- text/css
- text/plain
- text/rtf
- text/xml
- text/json
- video/avi
- video/flv
- video/mpeg
- application/vnd.ms-excel

Pulsa el botón de la derecha para obtener más información acerca de los tipos MIME.

MIME WIKIPEDIA

## Descargar la librería Jersey

---

Para crear nuestro primer servicio web, vamos a necesitar una librería que implemente las principales funciones descritas en el modelo de arquitectura REST.

Si vamos a desarrollar en Java, como es nuestro caso, hay que hablar de **JAX-RS** (*Java API for RESTful Web Services*).

Pero JAX-RS sigue siendo una especificación (determinada para aplicaciones Java), y existen varias librerías que siguen dicha especialización y que pueden considerarse librerías JAX-RS. Aquí es donde llegamos a Jersey como API que sigue la especificación JAX-RS y que es, además, la más utilizada hoy en día en las aplicaciones Java empresariales.

JAX-RS define algunas anotaciones que permiten especificar la forma en la que una clase Java normal se convierte en un recurso web. Algunas de estas anotaciones son: `@GET`, `@POST`, `@Path`, `@Produces`. No es la primera vez que te encuentras con estas anotaciones y puede que te preguntes qué son realmente; lo aclararemos en el siguiente apartado.

---

Por fin, ha llegado el momento de descargar Jersey.

**Jersey***RESTful web Services in Java*

Haz clic en el botón de la derecha para entrar en la web oficial de Jersey.

**JERSEY WEB OFICIAL**

Si ya has entrado a la web, verás que tiene un aspecto similar a este:

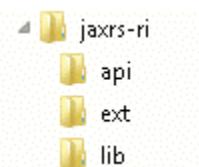
The screenshot shows the Jersey official website. At the top left is a yellow t-shirt icon followed by the word "Jersey". Below it says "RESTful Web Services in Java.". On the left side, there's a section titled "About" with a brief introduction to Jersey's purpose and history. Below that is a "Goals" section listing three bullet points: tracking the JAX-RS API, providing regular releases, and making it easy to build RESTful services. A note mentions the latest stable release is 2.27. To the right, there's a "Latest Articles" sidebar with links to "Jersey 2.27 Has Been Released" (Apr 13, 2018), "What's going on with Jersey" (Apr 03, 2018), and "What to do when JAX-RS cannot find it's Providers aka My message body writer is not used" (Mar 24, 2015). Further down are links to "JAX-RS Providers on Client and on Server" (Mar 17, 2015) and "Jersey 2.x Client on Android - Take 2" (Mar 17, 2015). At the bottom right of the main content area is a "Download" button with a cloud icon, which is circled in red in the original image.

Haz clic en el enlace **Download** que aparece a la derecha, y después en el enlace que te marcamos con un rectángulo rojo en la siguiente imagen:

This screenshot is identical to the one above, showing the Jersey website homepage. The "Download" button at the bottom right is highlighted with a red rectangle. Below the main content, there's a section for "JAX-RS 2.1 / Jersey 2.26+" with a note about Jersey 2.27 being the most recent release. It also lists two download links: "Jersey JAX-RS 2.1 RI bundle" and "Jersey 2.27 Examples bundle". A note at the bottom states that all Jersey 2 release binaries are available for download under the Jersey 2 maven root group identifier or g: well as from the [java.net maven repository](#).

Dará comienzo la descarga de un archivo denominado *jaxrs-ri-2.27.zip*.

Una vez completada la descarga, **descomprime el archivo en la ubicación que deseas**, donde guardes tus pruebas y ejercicios Java. Tras la descompresión, tendrás una carpeta denominada ***jaxrs-ri* con tres subcarpetas**, tal y como puedes apreciar en la siguiente imagen:



Cada una de estas subcarpetas contiene archivos .jar que forman parte de las librerías de Jersey.



Ten en cuenta que, tanto los sitios web como las librerías, se van actualizando y cambiando ligeramente. Es posible que en el momento en que estés revisando este apartado, tanto la web oficial de Jersey como la estructura de la librería no sean exactamente cómo te mostramos. Si tienes dudas, consulta a tu tutor.

## Anotaciones

---

A lo largo del curso nos hemos encontrado varias veces con las *anotaciones*, esas palabras raras que comienzan con el carácter @. Pero, ¿qué son realmente?

---

Las **anotaciones Java** proveen de un sistema para añadir metadatos al código fuente, que estarán disponibles para la aplicación en tiempo de ejecución.

### ¿Qué son los metadatos?

La traducción literal de metadato es *sobre dato*, y hace referencia a **un dato que describe a otro dato**. En aplicaciones empresariales, se utiliza para añadir información de configuración a un objeto cuya misión es servir como recurso a un proyecto.

[Obtén más información en la wikipedia.](#)



Un **ejemplo de anotación**, que con seguridad te has encontrado en más de una ocasión, es la anotación `@Override` delante de un método, para indicar que dicho método viene heredado de una clase base y está siendo sobrescrito.

## Ejercicio práctico

---

Para que comprendas qué son las anotaciones, vas a crear y a utilizar tu propia anotación personalizada.

---

Seguiremos los siguientes pasos:

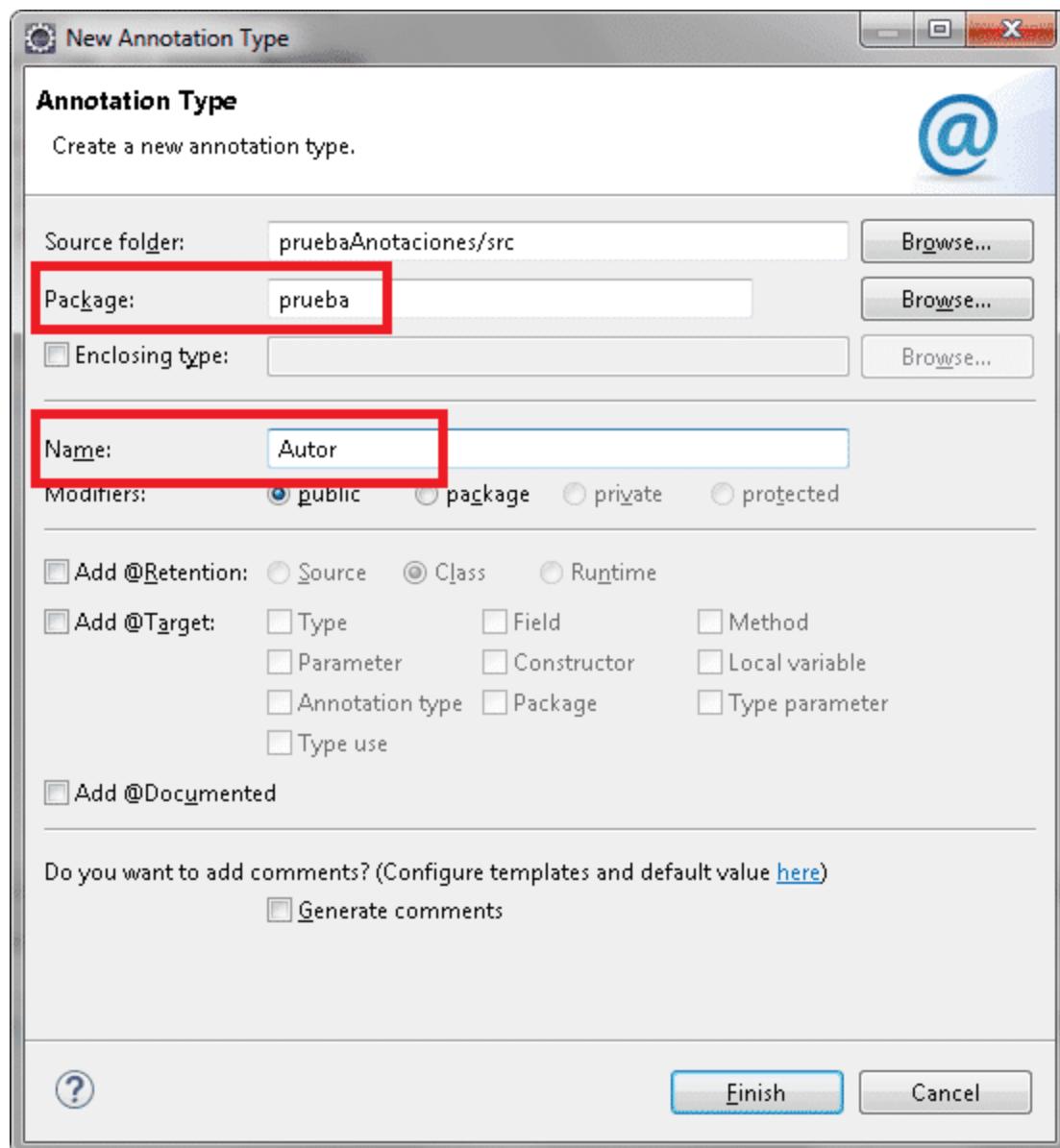
1. Crearemos la anotación personalizada `@Autor`, que servirá para añadir información a las clases y métodos que queramos sobre quién es el autor que desarrolló dichas clases o métodos. La anotación contará con las propiedades *nombre* y *dirección*.
2. Crearemos una clase llamada `Coche` y la anotaremos. Dentro de la clase `Coche` implementaremos un método llamado `acelerar()`, que también anotaremos.
3. En la clase `Principal` crearemos un objeto de la clase `Coche` y accederemos a los datos suministrados por la anotación `@Autor`.

Comencemos:

1

Crea la anotación personalizada `@Autor`.

- Crea un proyecto nuevo, denominado `pruebaAnotaciones` (*File / New / Java Project*).
- Haz clic derecho sobre el nombre del nuevo proyecto y selecciona en el menú contextual `New / Annotation`.



En el cuadro de diálogo **New Annotation Type** rellena los datos como ves en la imagen, para crear la anotación Autor en el paquete prueba. Termina haciendo clic en el botón **Finish**. Por ahora, tu anotación debe tener este aspecto.

```
package prueba;

public @interface Autor {
```

Las anotaciones son como interfaces especiales. Observa que la palabra *interface* va precedida por el símbolo @.

Ahora, completa el código de la anotación de la siguiente manera:

```
package prueba;

import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
public @interface Autor {
    String nombre() default "Telefónica";
    String direccion() default "Distrito Telefónica";
}
```

Ahora, la anotación cuenta con los atributos o datos *nombre* y *direccion* cuyos valores predeterminados son "Teléfonica" y "Distrito Telefónica".

La anotación *@Retention(RetentionPolicy.RUNTIME)* permite que la anotación *@Autor* esté disponible en tiempo de ejecución.

2

Crea una clase llamada *Coche* y anótala.

La nueva clase *Coche* tendrá el siguiente código:

```
import prueba.Autor;

@Autor
public class Coche {
    private String marca;
    private String modelo;
    private int velocidad;

    public Coche(String marca, String modelo) {
        this.marca = marca;
        this.modelo = modelo;
        this.velocidad = 0;
    }

    public String getMarca() {
        return marca;
    }
    public String getModelo() {
        return modelo;
    }
    public int getVelocidad() {
        return velocidad;
    }

    @Autor(nombre="Perico de los palotes", direccion="C/ Palotes, 54")
    public void acelerar() {
        this.velocidad = this.velocidad + 10;
    }

    @Override
    public String toString() {
        return "Coche [marca=" + marca + ", modelo=" + modelo + ", velocidad=" + velocidad + "]";
    }
}
```

Hemos anotado la nueva clase *Coche* sin especificar ningún valor para los atributos *nombre* y *dirección*, por lo tanto, habrá tomado los valores predeterminados "Telefónica" y "Distrito Telefónica". Sin embargo, el método *acelerar()* ha sido anotado especificando el valor "Perico de los Palotes" para el atributo *nombre* y "C/ Palotes, 54" para el atributo *direccion*.

3

En la clase *Principal*, crea un objeto de la clase *Coche* y accede a los datos suministrados por la anotación `@Autor`.

Por último, crea la clase *Principal* con el siguiente código:

```
import prueba.Autor;

public class Principal {

    public static void main(String[] args) throws NoSuchMethodException, SecurityException {
        Coche miCoche = new Coche("Ford", "Fiesta");
        System.out.println(miCoche);

        // Accediendo a los datos de la anotación del método.
        Autor a1 = miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);
        System.out.println("Nombre autor: " + a1.nombre());
        System.out.println("Dirección autor: " + a1.direccion());

        // Accediendo a los datos de la anotación de la clase.
        Autor a2 = miCoche.getClass().getAnnotation(Autor.class);
        System.out.println("Nombre autor: " + a2.nombre());
        System.out.println("Dirección autor: " + a2.direccion());
    }
}
```

Veamos las partes más importantes de la clase *Principal*:

**Coche miCoche = new Coche("Ford", "Fiesta");**

Creamos un objeto de la clase *Coche* y después mostramos su estado, invocando al método *toString()* con esta línea:

```
System.out.println(miCoche);
```

Recuerda que *toString()* es el método al que se invoca por defecto cuando no se especifica el método a ejecutar.

**Autor a1 = miCoche.getClass().getMethod("acelerar").getAnnotation(Autor.class);**

La expresión *miCoche.getClass()* devuelve un objeto de tipo *Class* que provee información sobre la clase a la que pertenece el objeto. De esta forma, estamos también obteniendo información sobre el método *acelerar()*, usando la expresión *getMethod("acelerar")*. Concretamente, estamos accediendo a los datos de la anotación.

**Autor a2 = miCoche.getClass().getAnnotation(Autor.class);**

Aquí estamos accediendo a los datos de la anotación de la clase.

# Crear y configurar el proyecto web dinámico

---

Ya hemos descargado la librería Jersey y ahora nos toca crear el proyecto Java en Eclipse.

Pero no será un proyecto cualquiera, sino un proyecto web dinámico.

Los proyectos web dinámicos tienen las siguientes características:

1. Son proyectos que se crean en la perspectiva Java EE de Eclipse IDE.
2. Tienen el objetivo de ser posteriormente desplegados y ejecutados en un **servidor web**, o **servidor de aplicaciones**. En nuestro caso, se tratará de un proyecto de servicios web que será desplegado en el servidor Apache Tomcat.
3. Puesto que el objeto principal de un proyecto web dinámico es su despliegue en un servidor web, será necesario configurar la forma en que se montará dicho despliegue. Nos referimos a la configuración del llamado *Web Deployment Assembly*.
4. Siguen la arquitectura cliente/servidor.

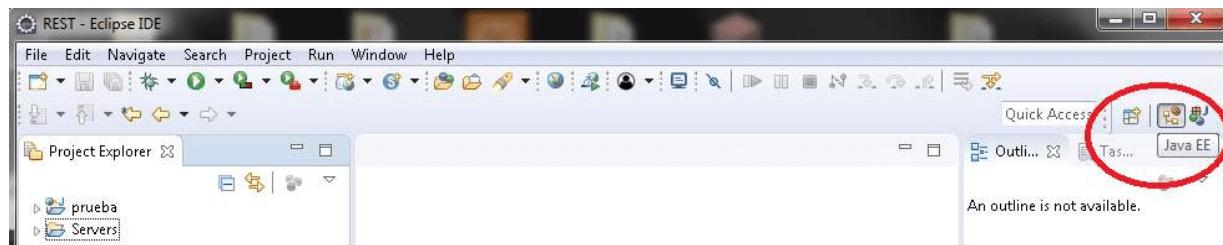
---

Sigue los pasos que te iremos indicando para crear y configurar el proyecto.

1

### Cambia a la perspectiva Java EE.

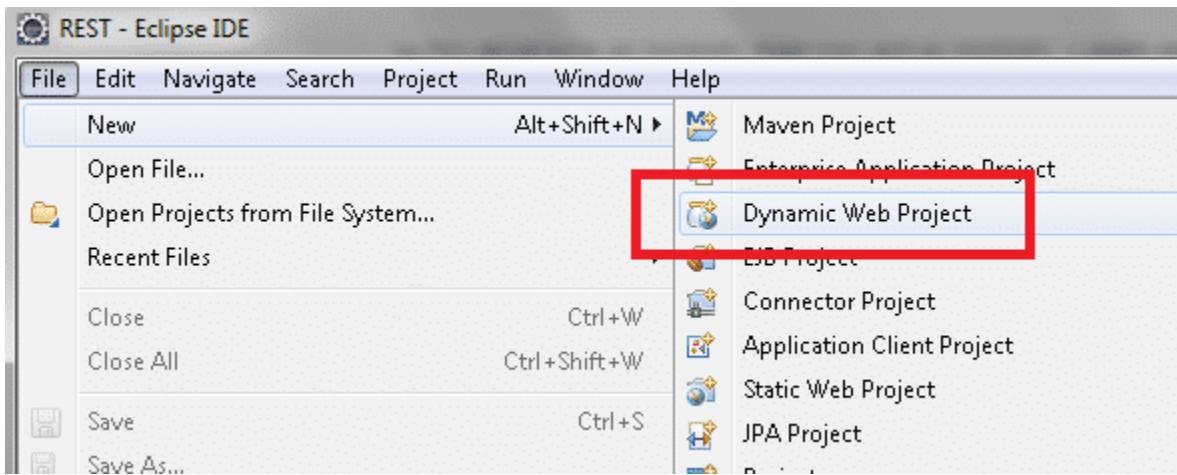
Para hacerlo, pulsa el botón que ves resaltado en la siguiente imagen, o bien el botón *Open perspective* situado justo a la izquierda.



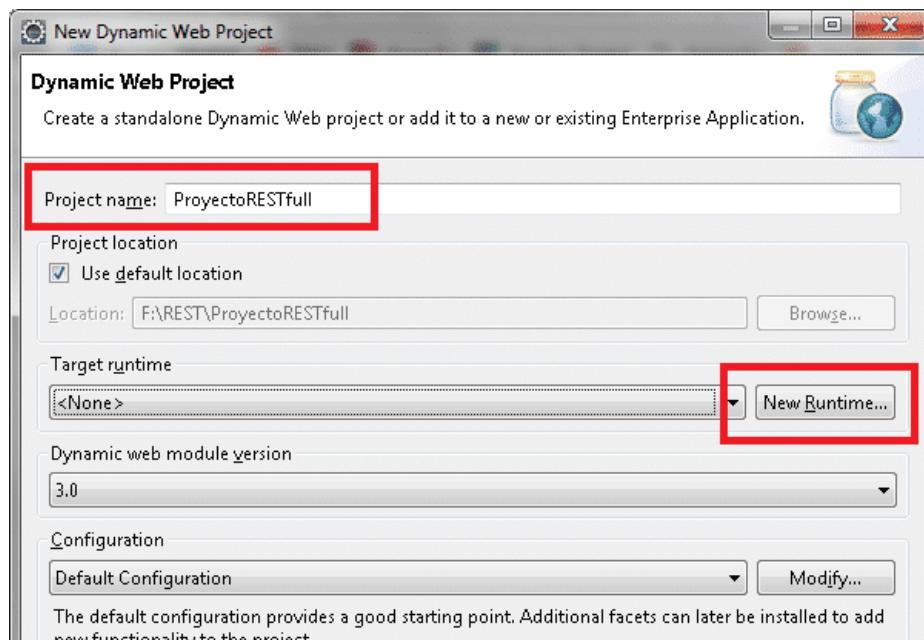
2

### Crea el nuevo proyecto y asócialo al servidor web Apache Tomcat.

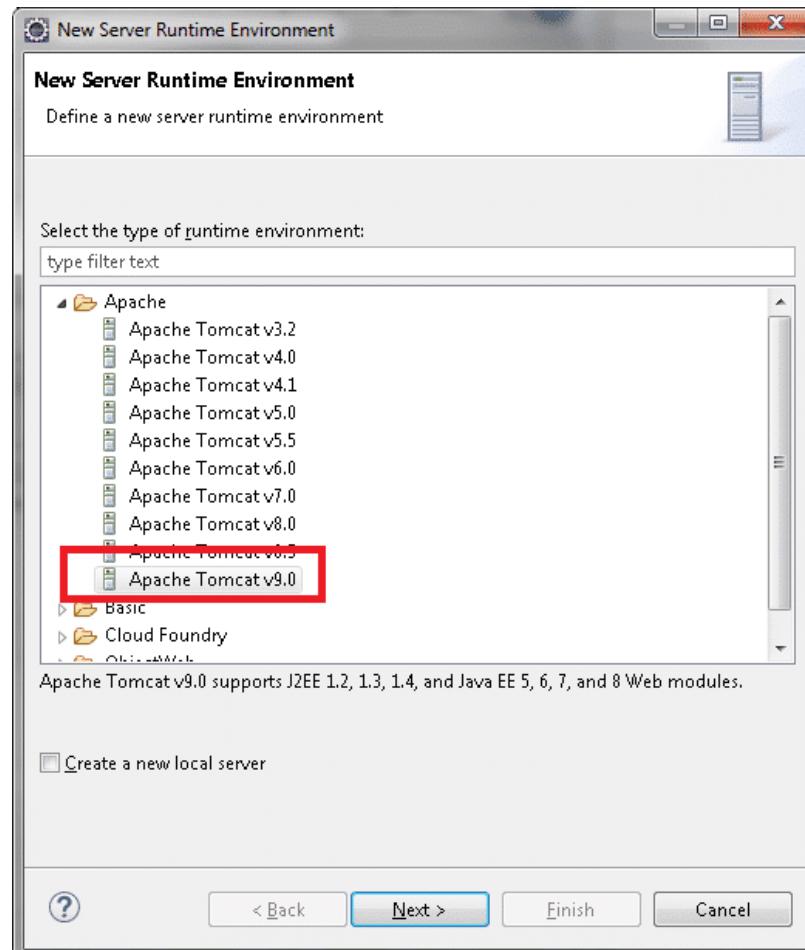
Selecciona en el menú *File / New / Dynamic Web Project*.



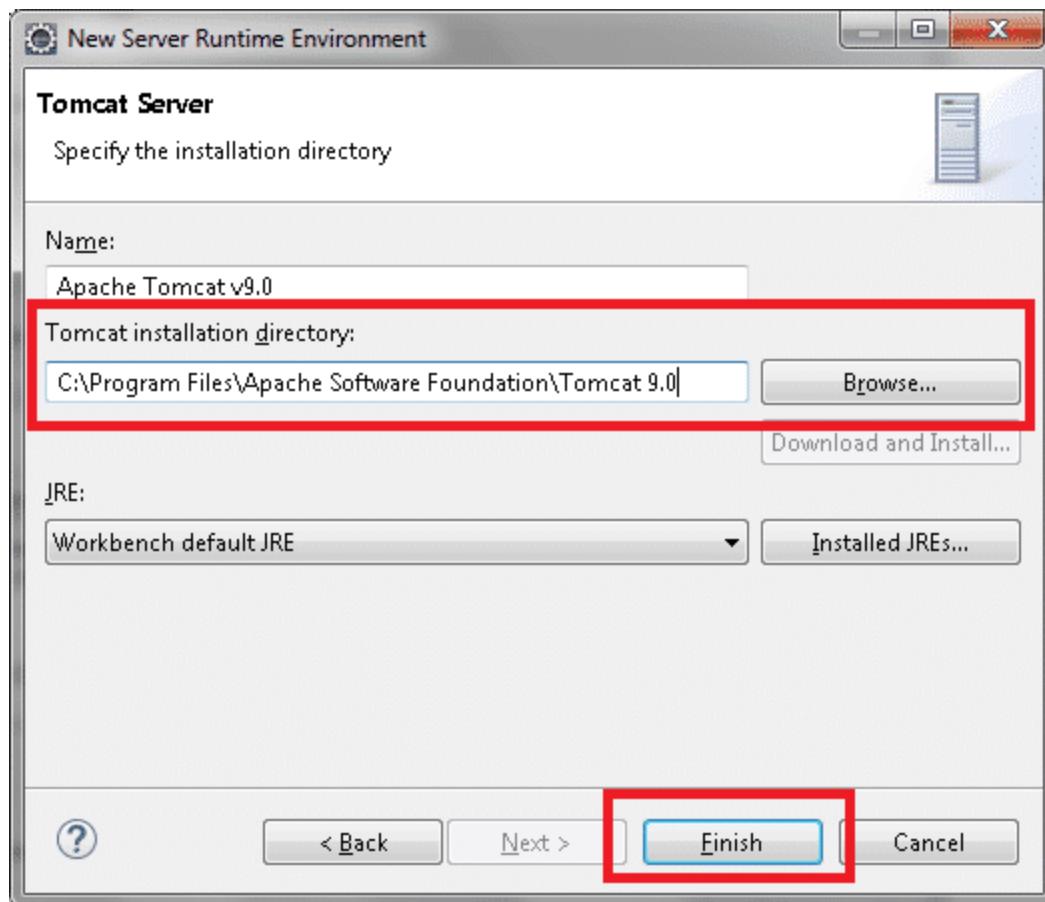
En el cuadro de diálogo *New Dynamic Web Project*, indica el nombre del nuevo proyecto (*Project name*). Pero no pulses *Finish* porque no has terminado; todavía tienes que configurar el *Target runtime*, donde especificarás la ubicación del servidor web en el que desplegarás el proyecto.



Haz clic en el botón **New Runtime ...** y selecciona **Apache Tomcat v9.0** en el cuadro de diálogo **New Server Runtime Environment**.

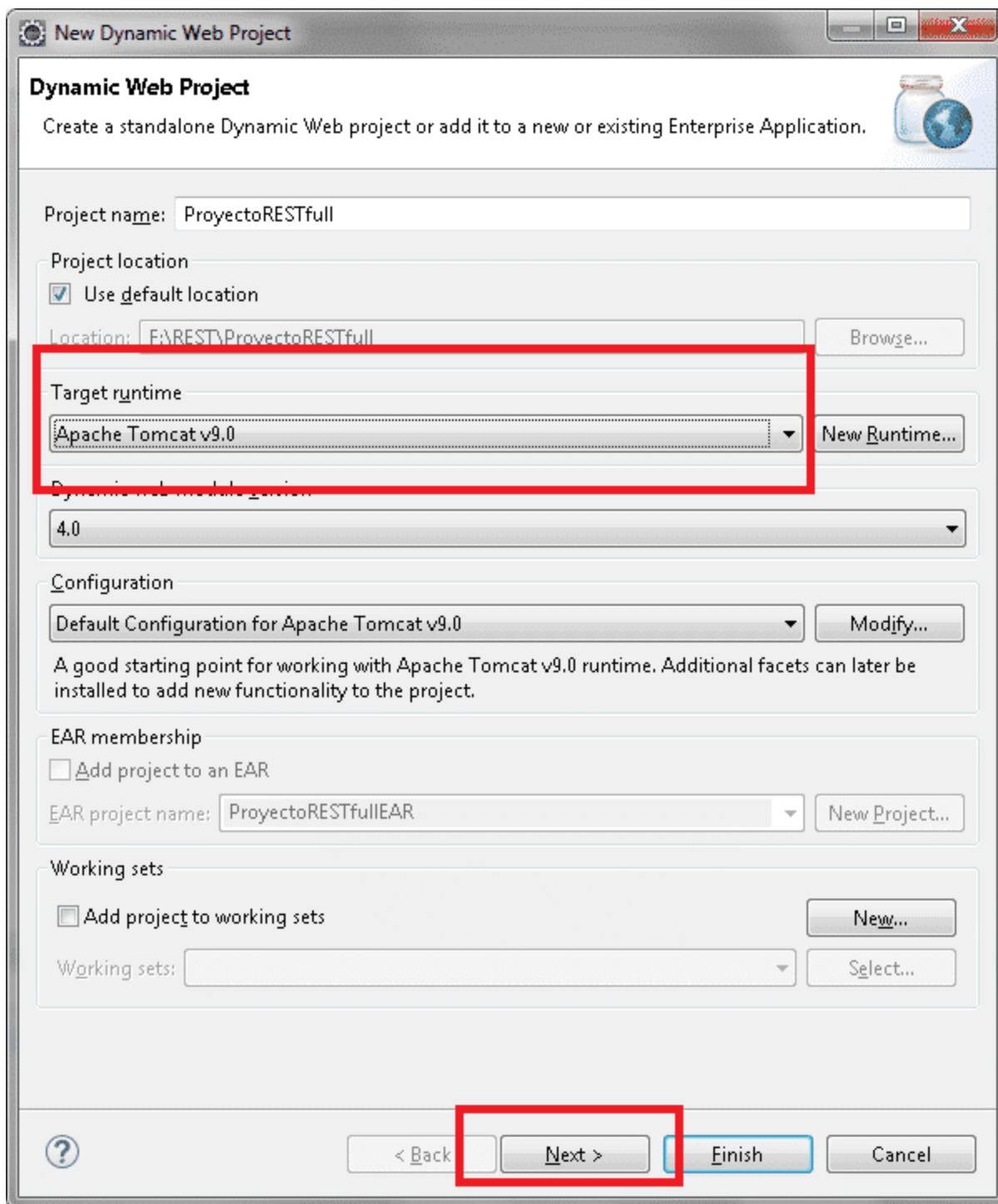


Pulsa el botón "*Next >*" y selecciona el directorio donde se encuentra la instalación de Apache Tomcat, pulsando el botón *Browse* para navegar por el sistema de archivos hasta localizar la carpeta.



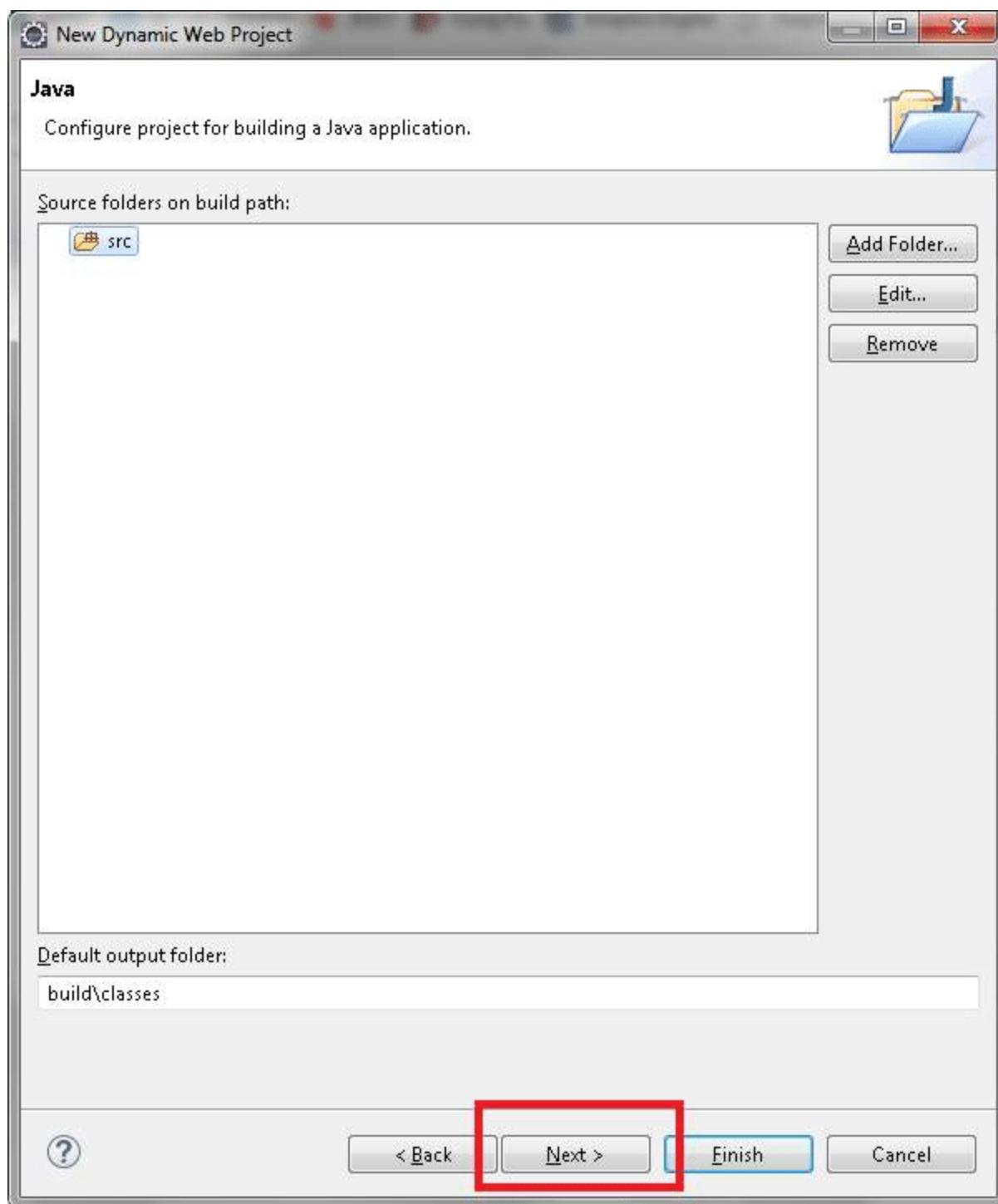
Una vez establecido el *Tomcat installation directory*, haz clic en el botón *Finish*.

Ahora, el cuadro de diálogo *New Dynamic Web Project* refleja el nuevo *Target runtime*.



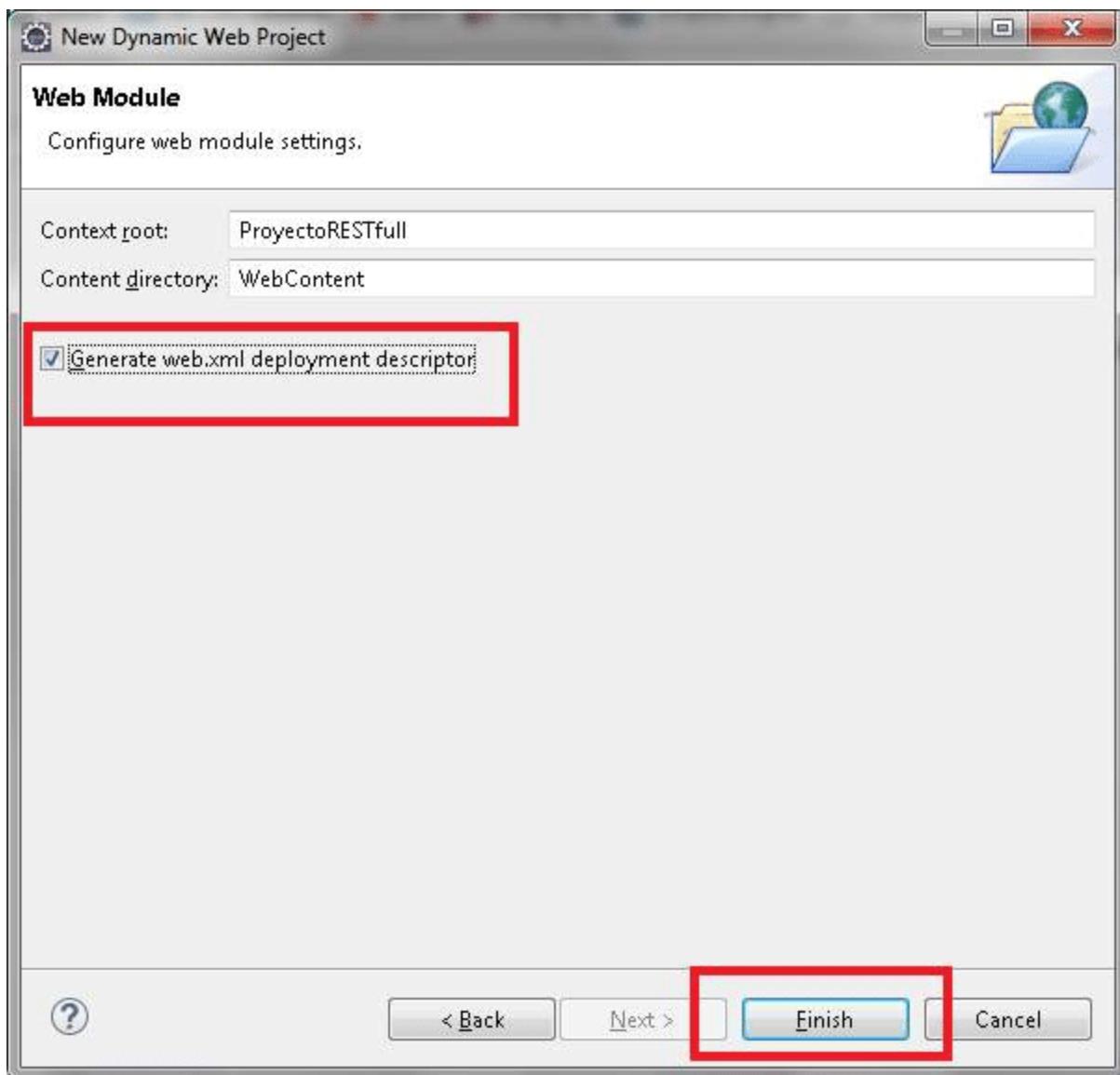
Para nuevos proyectos dentro del mismo espacio de trabajo, no será necesario volver a establecer el *Target runtime*.

Ahora pulsa el botón *Next >* para avanzar al siguiente paso, en el que se especifica la carpeta destino del código fuente de las clases Java.

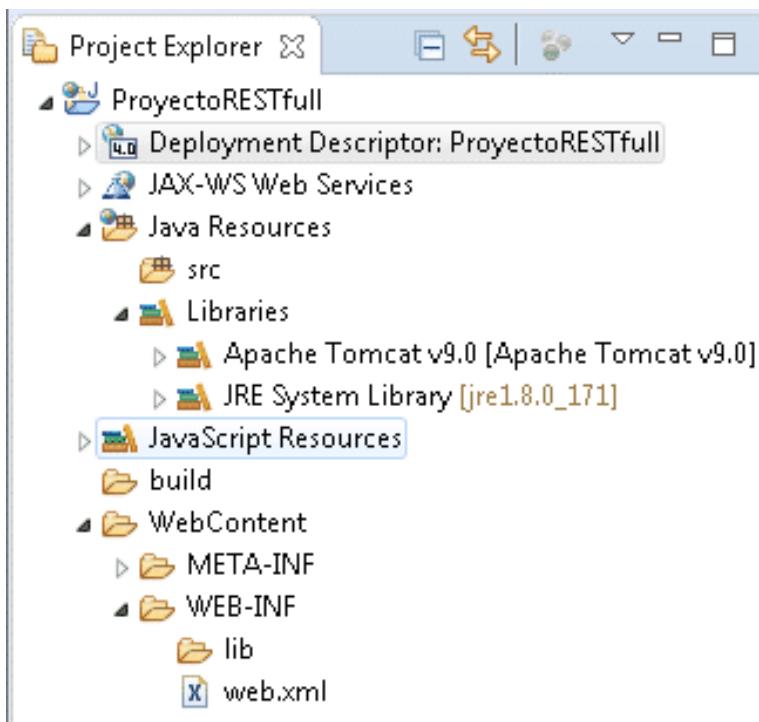


Aquí no es necesario realizar ninguna acción, así que vuelve a pulsar el botón *Next >*.

Y te encuentras en el último paso, donde tendrás que marcar la casilla de verificación **Generate web.xml deployment descriptor**. El archivo *web.xml* es fundamental, porque es utilizado para especificar valores importantes de configuración en el despliegue de la aplicación en el servidor web.



Por último pulsa el botón *Finish*. Tu proyecto web dinámico ya está creado y tiene el siguiente aspecto:

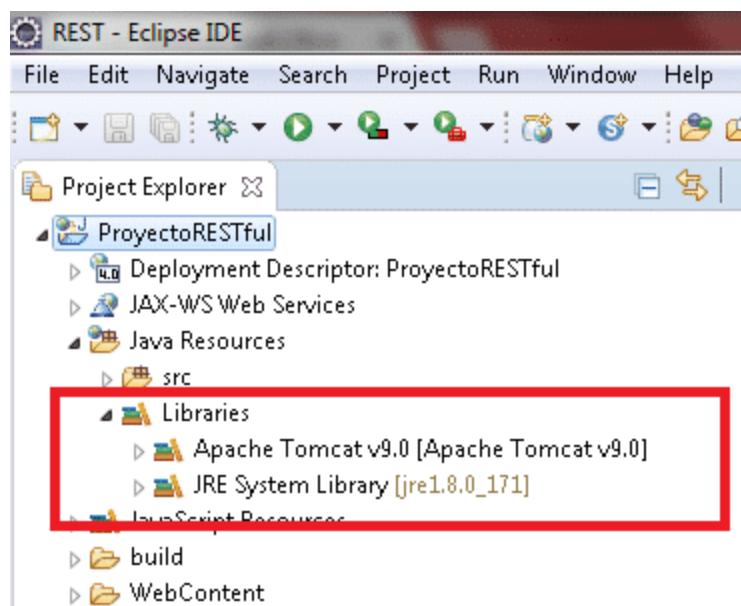


**NOTA.** En la mayoría de los casos, los proyectos web dinámicos se crean para desarrollar sitios web dinámicos y la carpeta *WebContent* es la que contendrá todos los recursos del sitio web: páginas, imágenes, archivos css, etc. Recuerda que nuestro cometido es algo diferente, ya que **no queremos alojar un sitio web, sino un servicio web**.

3

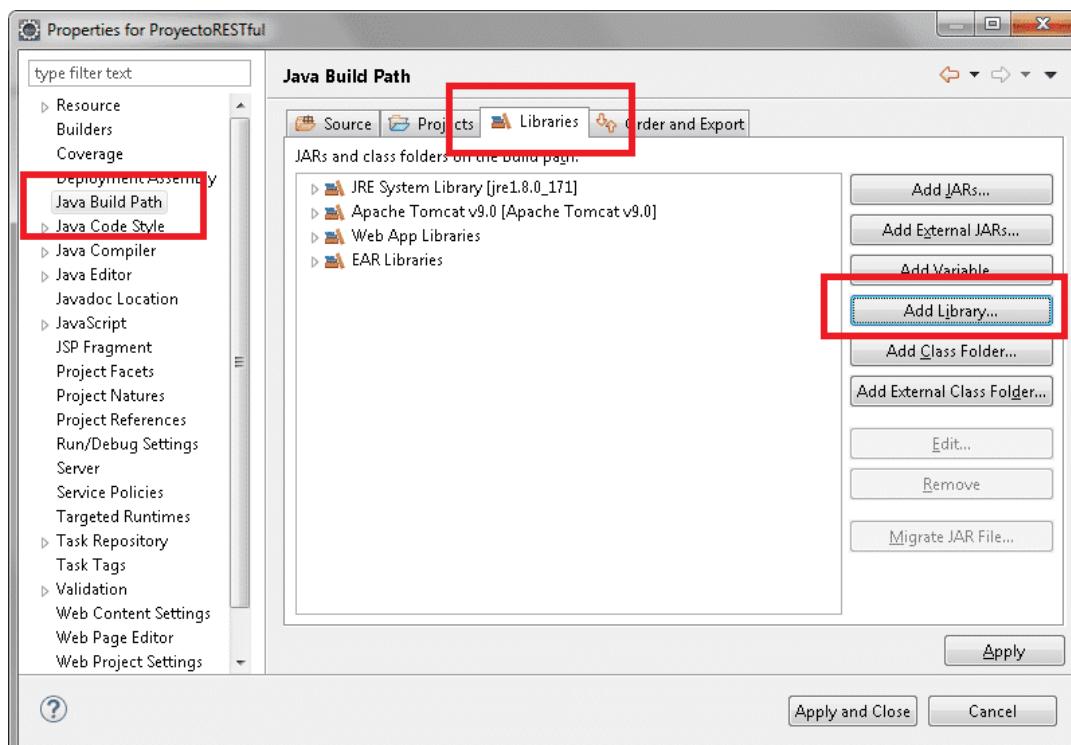
Importa la librería Jersey dentro del proyecto web dinámico.

Dentro de la carpeta *Libraries* del proyecto Eclipse se encuentran las librerías de clases que formarán parte del proyecto; por ahora, están las librerías del JRE (siempre presentes) y las propias de *Apache Tomcat*.

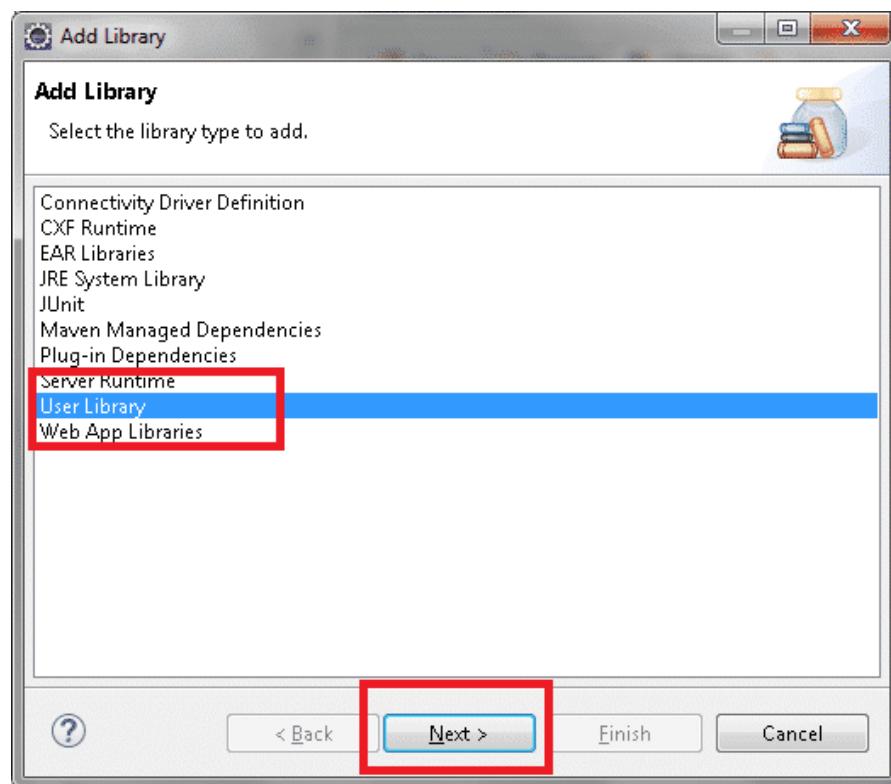


Tu siguiente tarea será lograr que de la carpeta *Libraries* cuelgue también la librería Jersey, así que vamos manos a la obra.

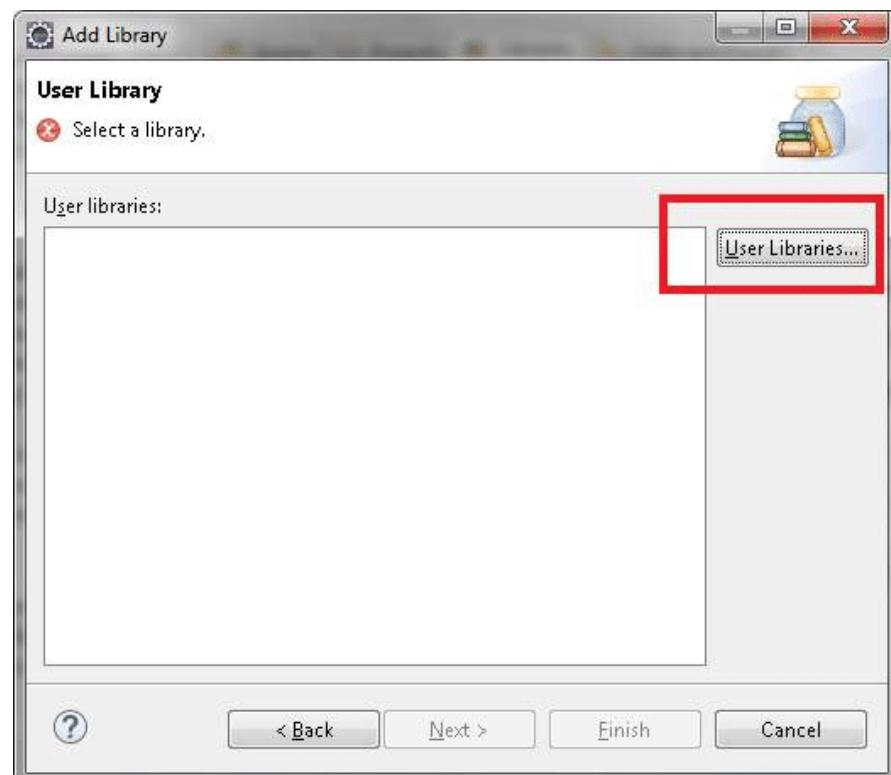
Haz clic derecho sobre el nombre del proyecto, selecciona la opción *Properties* en el menú contextual, selecciona *Java Build Path* dentro del listado de la izquierda, abre la ficha *Libraries* y, por último, pulsa el botón *Add Library...*.



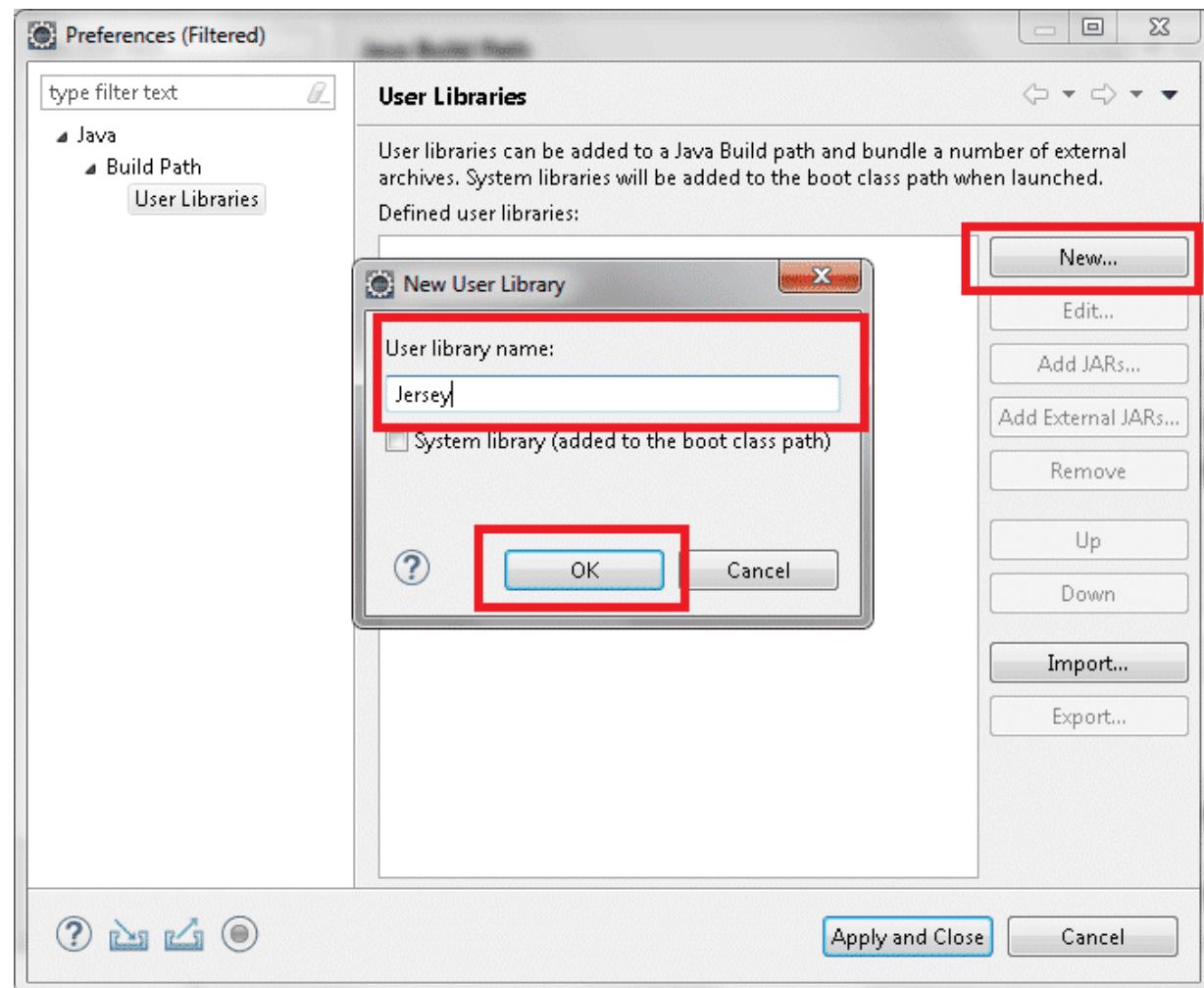
Selecciona *User Library* y pulsa *Next >*.



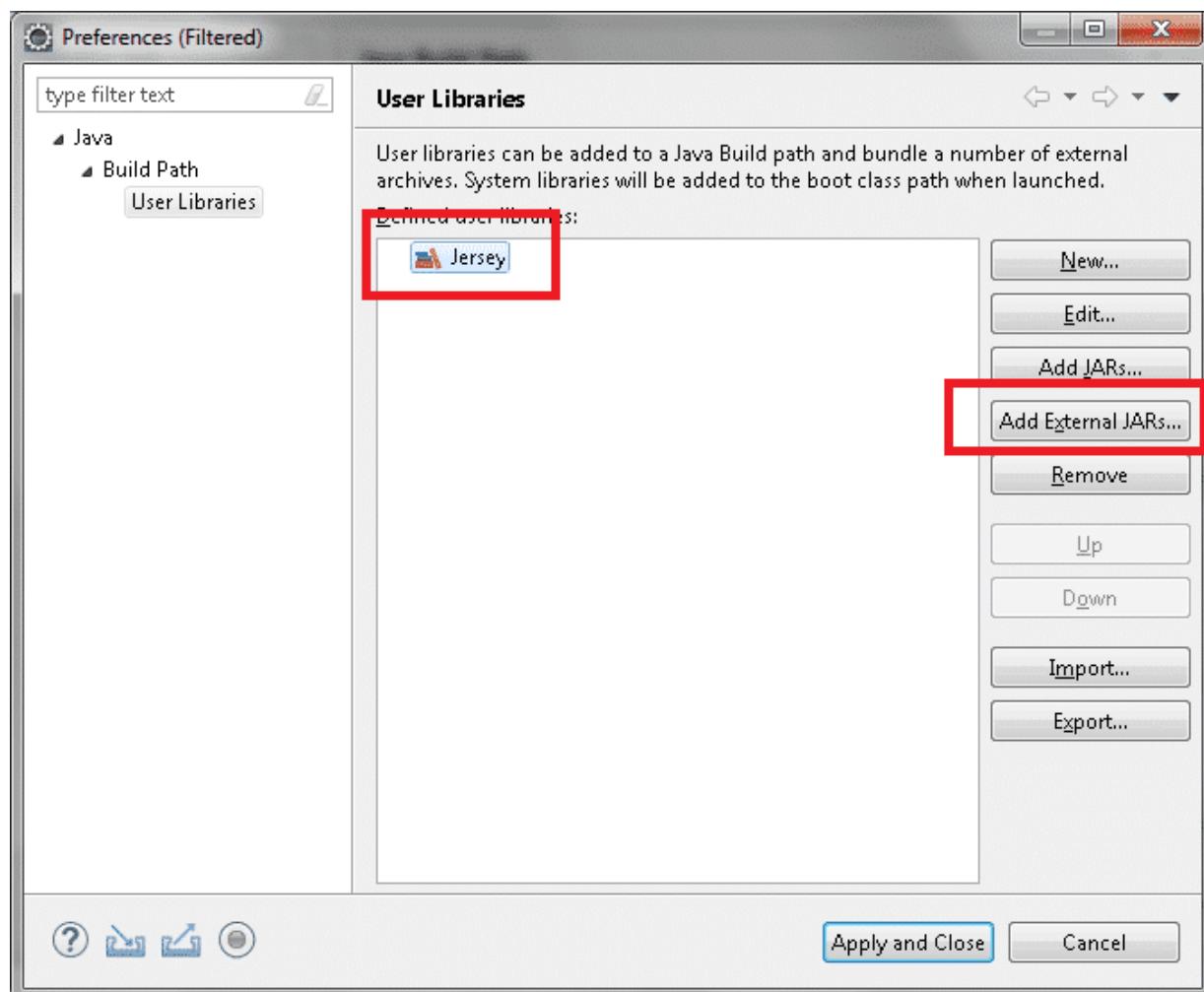
Pulsa el botón *User Libraries* ....



Haz clic en el botón **New**, escribe **Jersey** como nombre de la nueva librería y haz clic en el botón **OK**.

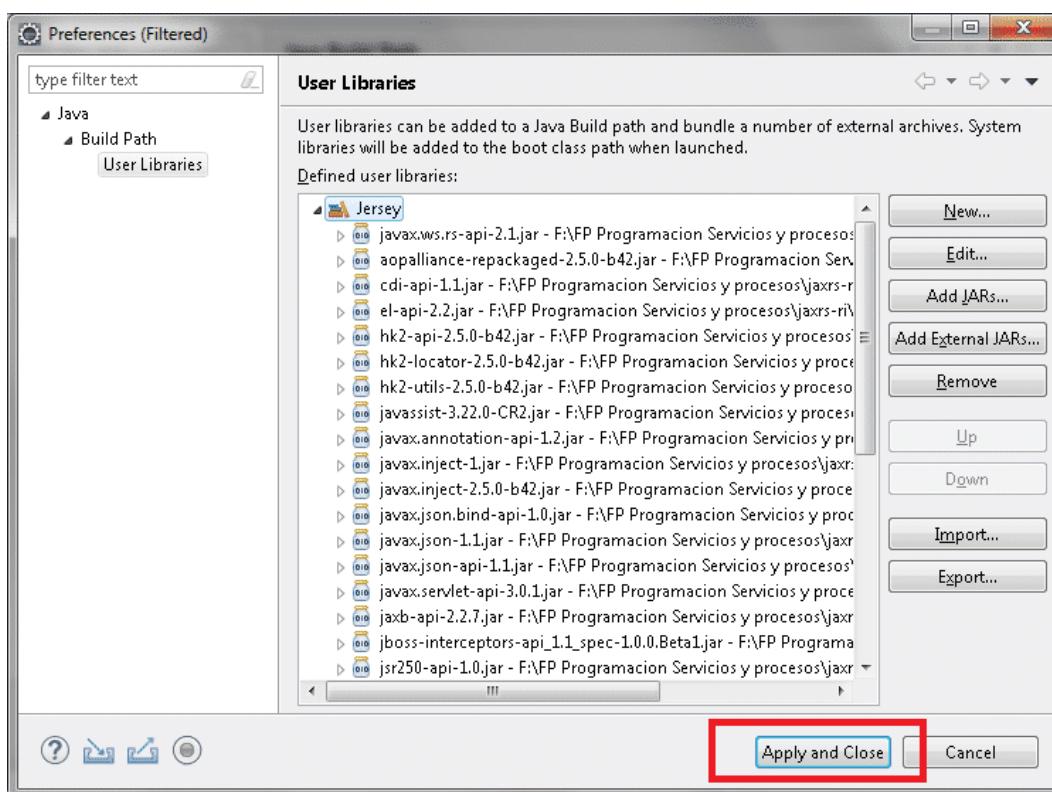


Ahora, el cuadro de diálogo te presenta la nueva librería Jersey. Debes tenerla seleccionada y hacer clic en el botón **Add External JARs...**.

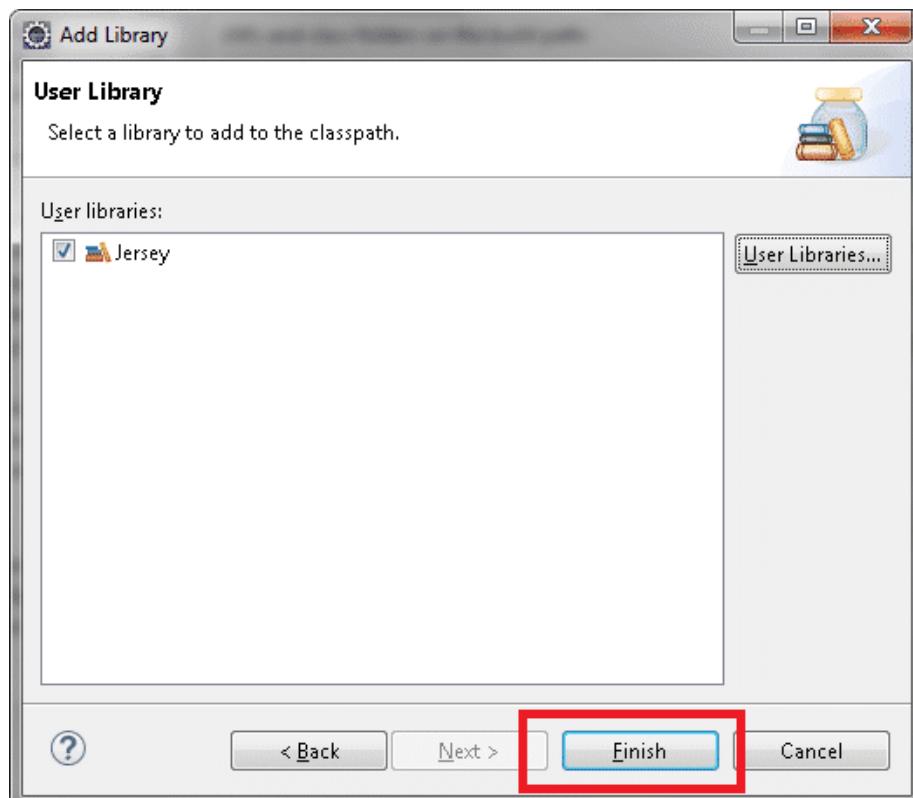


Localiza la carpeta *jaxrs-ri* y ve seleccionando todos y cada uno de los archivos .jar de las tres subcarpetas que componen la librería Jersey. Cuando selecciones los de la primera carpeta, tendrás que volver a seleccionar la librería *Jersey* y pulsar de nuevo el botón *Add External JARs ...* para pasar a la siguiente carpeta.

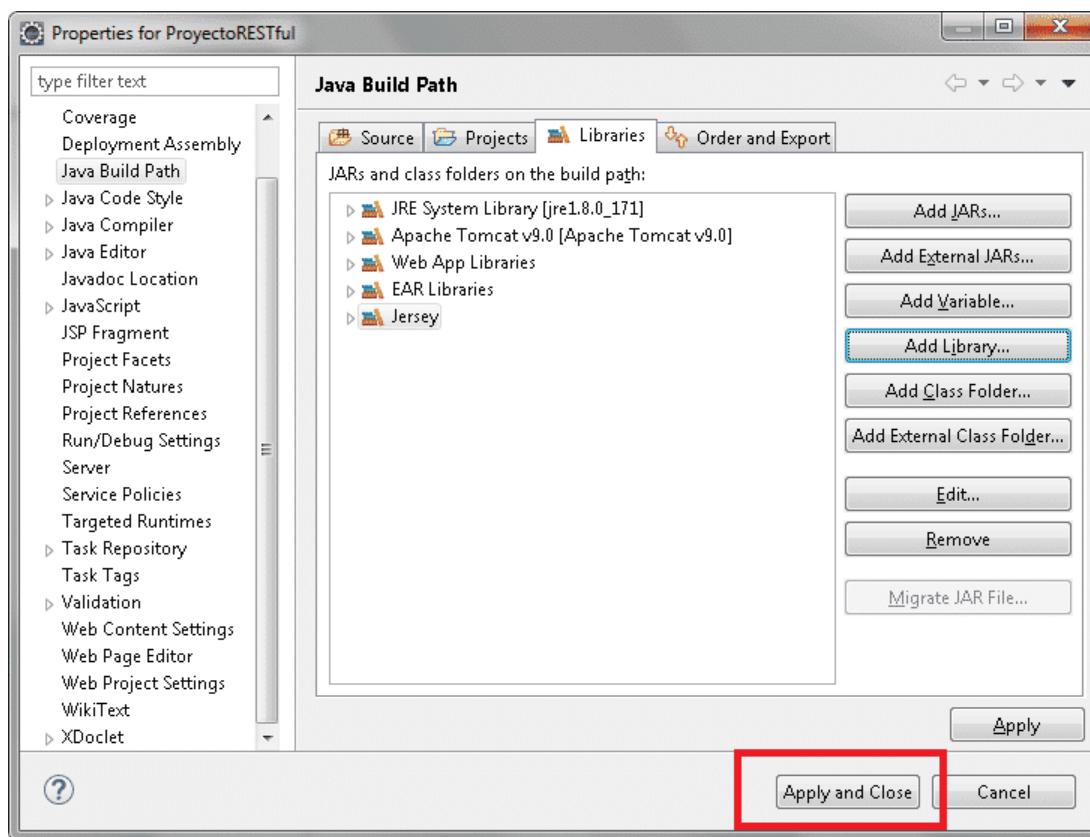
Cuando tengas incluidos todos los archivos .jar, haz clic en el botón *Apply and Close*.



En el siguiente cuadro de diálogo, pulsa *Finish*.

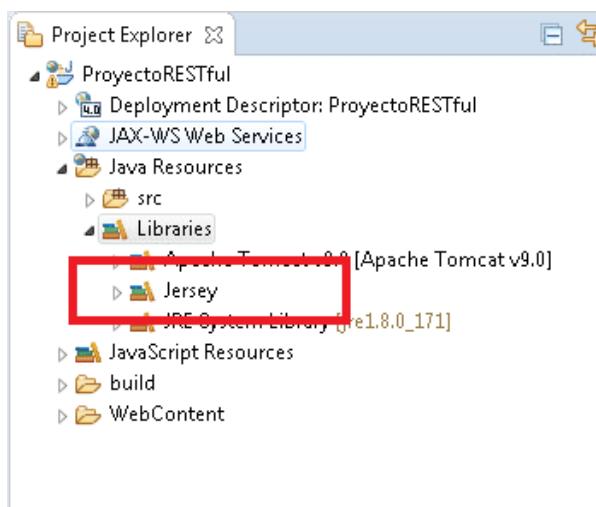


Ahora, pulsa **Apply and Close** en el cuadro de diálogo *Properties*.



---

Si has seguido bien los pasos habrás logrado el objetivo. Tu proyecto ya incluye la librería Jersey.



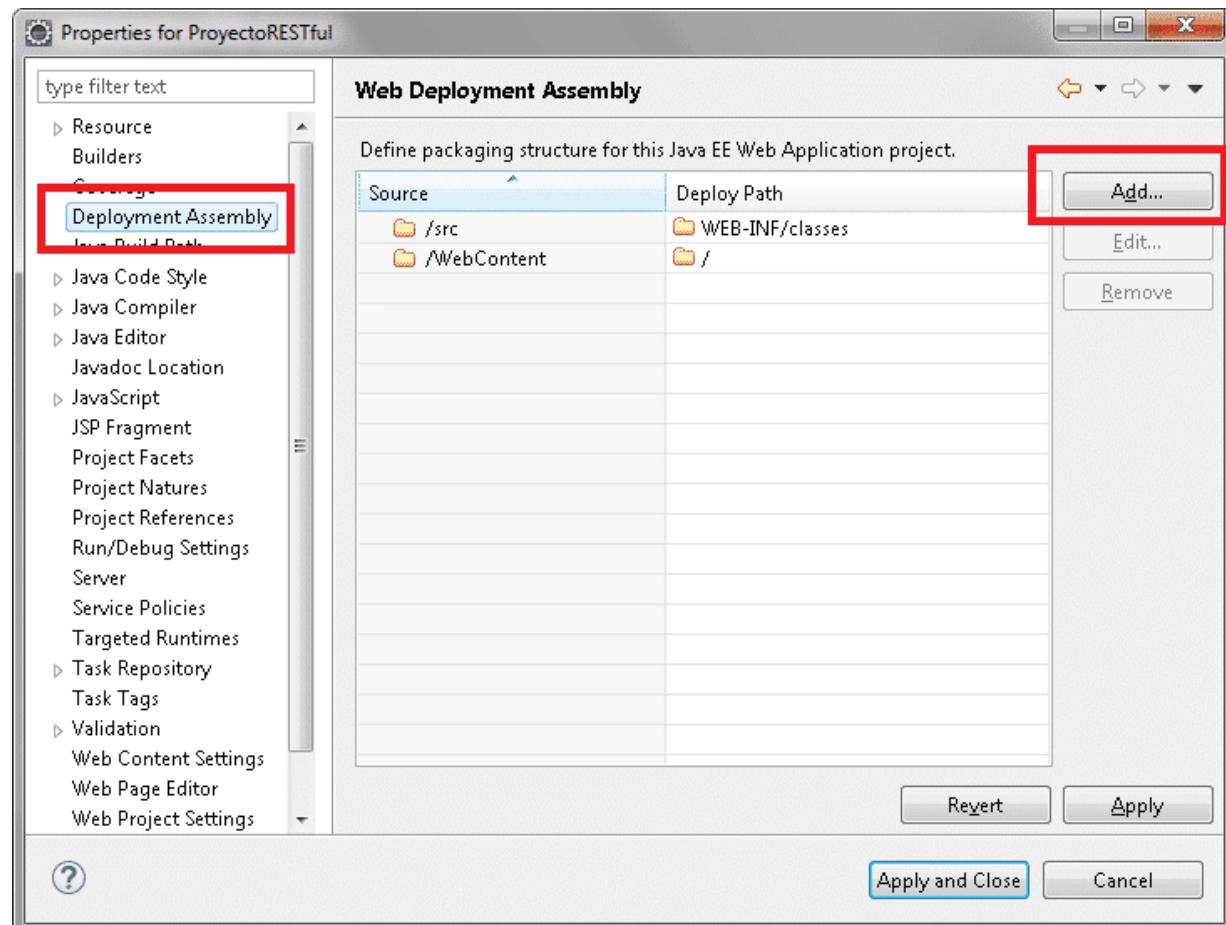
4

#### Configura el Web Deployment Assembly.

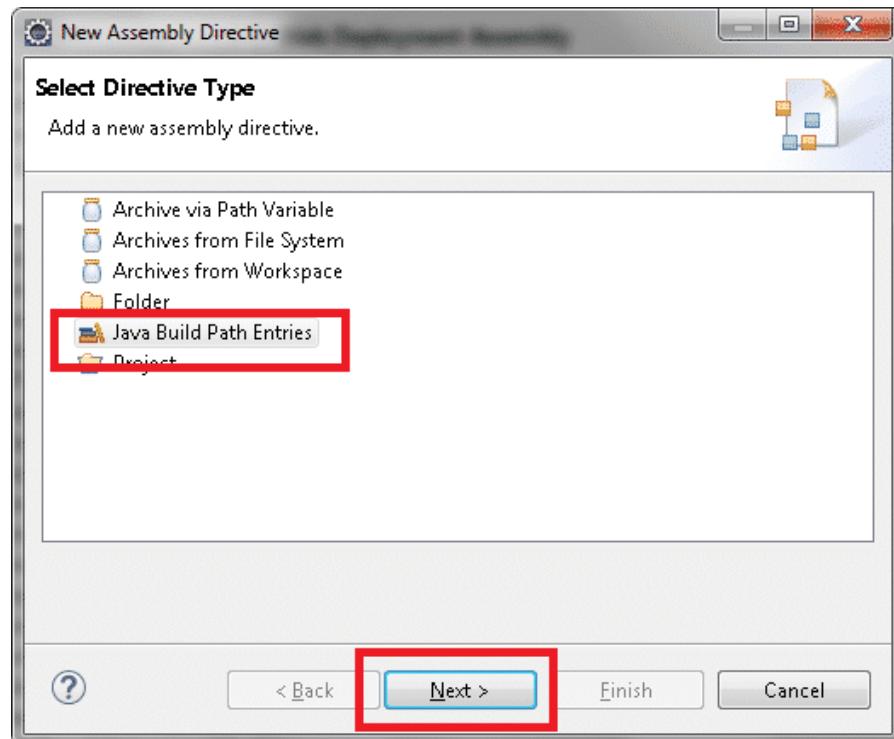
**i** La configuración del **Web Deployment Assembly** establece los elementos del proyecto Eclipse que serán exportados a Apache Tomcat a la hora de realizar el despliegue de la aplicación.

Vamos a ver qué pasos tendrás que realizar para configurar el futuro despliegue de la aplicación en Apache Tomcat. Recuerda que necesitamos que, a la hora de desplegar la aplicación, también se incluyan en el servidor los archivos de la librería Jersey.

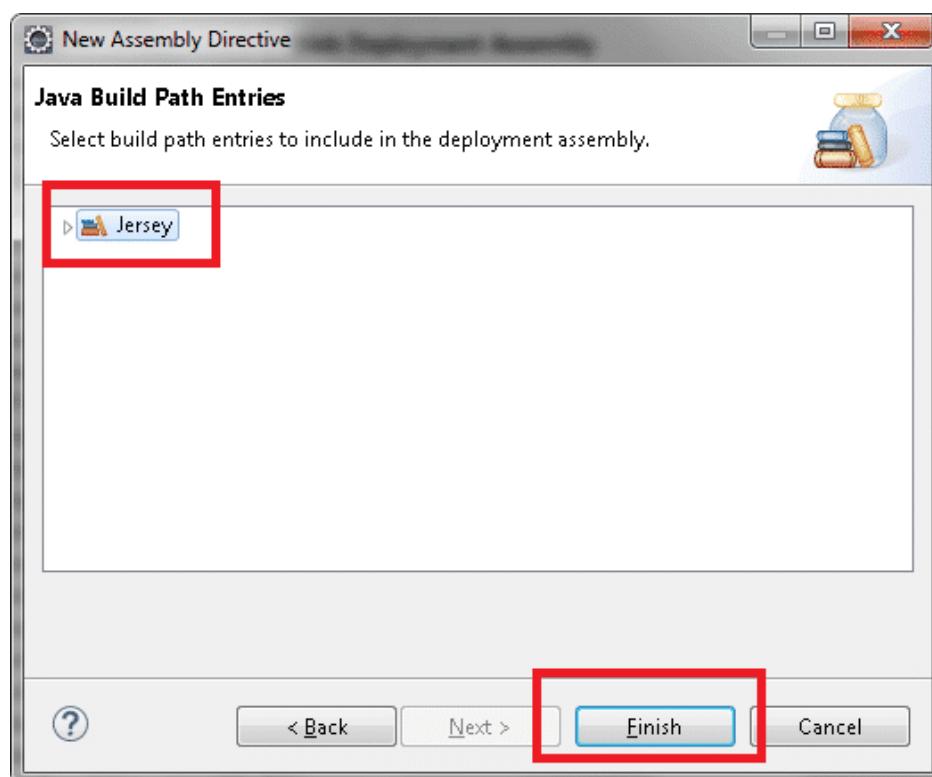
Haz clic derecho sobre el nombre del proyecto, selecciona la opción **Properties** en el menú contextual, selecciona **Deployment Assembly** en las opciones de la izquierda y pulsa el botón **Add....**



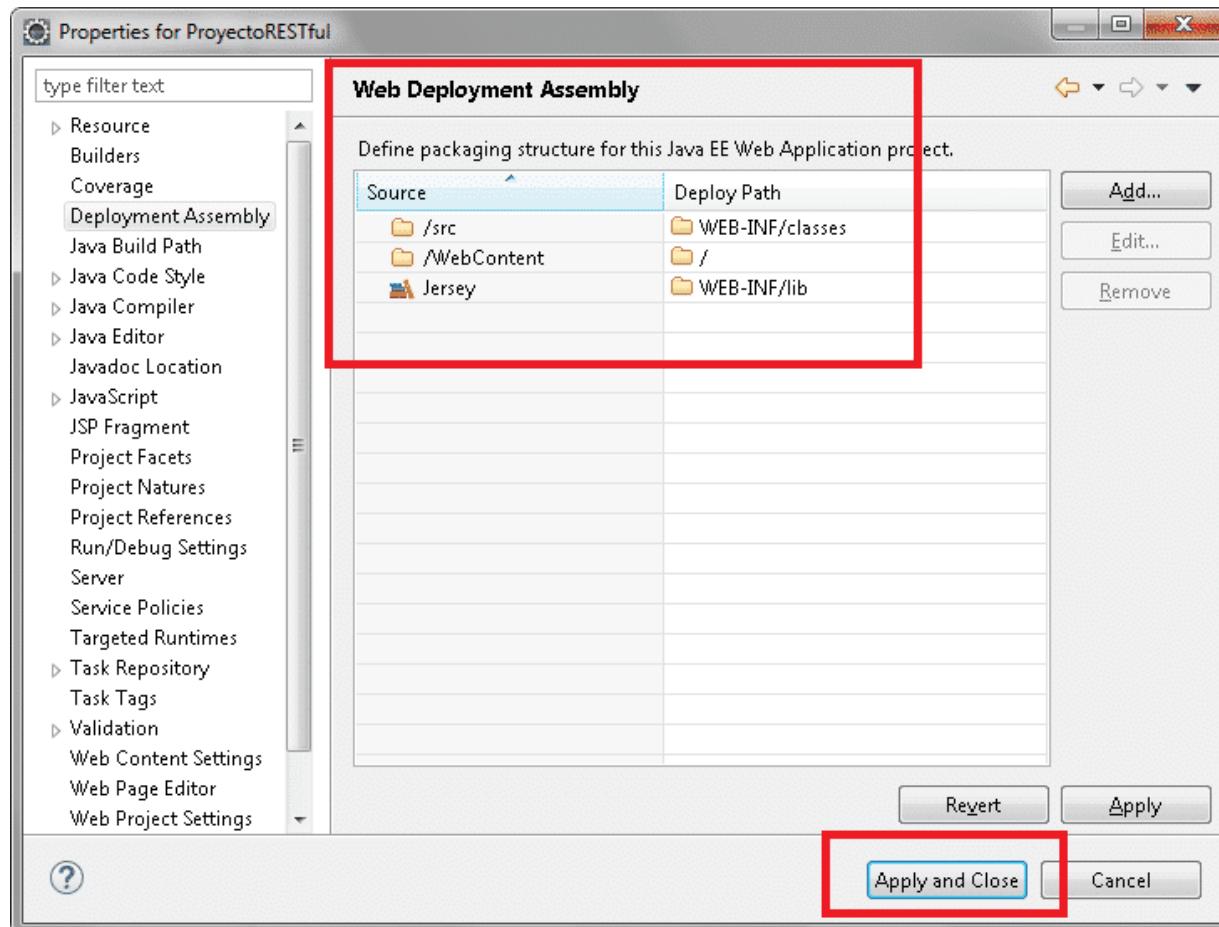
En el siguiente cuadro de diálogo, selecciona **Java Build Path Entries** y haz clic en el botón **Next >**.



En el siguiente cuadro, selecciona **Jersey** y haz clic en el botón **Finish**.



Ahora, el cuadro de diálogo *Properties* está mostrando los elementos que se exportarán en el proyecto web Apache Tomcat y la ubicación en la que estará cada uno de ellos.



Para terminar, haz clic en el botón *Apply and Close*.

Ya tienes el proyecto perfectamente configurado para comenzar a implementar tu nuevo servicio web REST.

# Crear el modelo de datos

Por lo general, un proyecto de servicios web Java debe contar con una o varias clases de modelo.

Se denominan **clases de modelo** porque tienen como objetivo contener los datos que el servicio web suministrará a los clientes que los soliciten, aunque se trata de clases Java normales y corrientes.

Como se trata de un primer servicio web muy sencillo, nuestra clase de modelo se limitará a **portar un mensaje y mantener un contador de peticiones** (contador de objetos *Mensaje* que se creen).

```
package com.itt.ws.modelo;

public class Mensaje {
    private String texto;
    private static long peticiones;

    public Mensaje(String texto) {
        this.texto = texto;
        peticiones++;
    }

    public String getTexto() {
        return texto + "\n" + "Acceso nº " + peticiones;
    }

    public void setTexto(String texto) {
        this.texto = texto;
    }

    public static long getPeticiones() {
        return peticiones;
    }
}
```

## Crear el controlador

También será imprescindible contar con una clase que actúe de controlador y que será la encargada de recibir peticiones de clientes y responder, haciendo uso de los datos expuestos en las clases de modelo.

Es decir: **el controlador hará de intermediario entre el proceso cliente y el servidor de servicios web.**

Dentro de esta clase, haremos uso de las anotaciones definidas por JARX-RS para especificar la forma en que se configurará el recurso web servido.

```
package com.itt.ws.control;

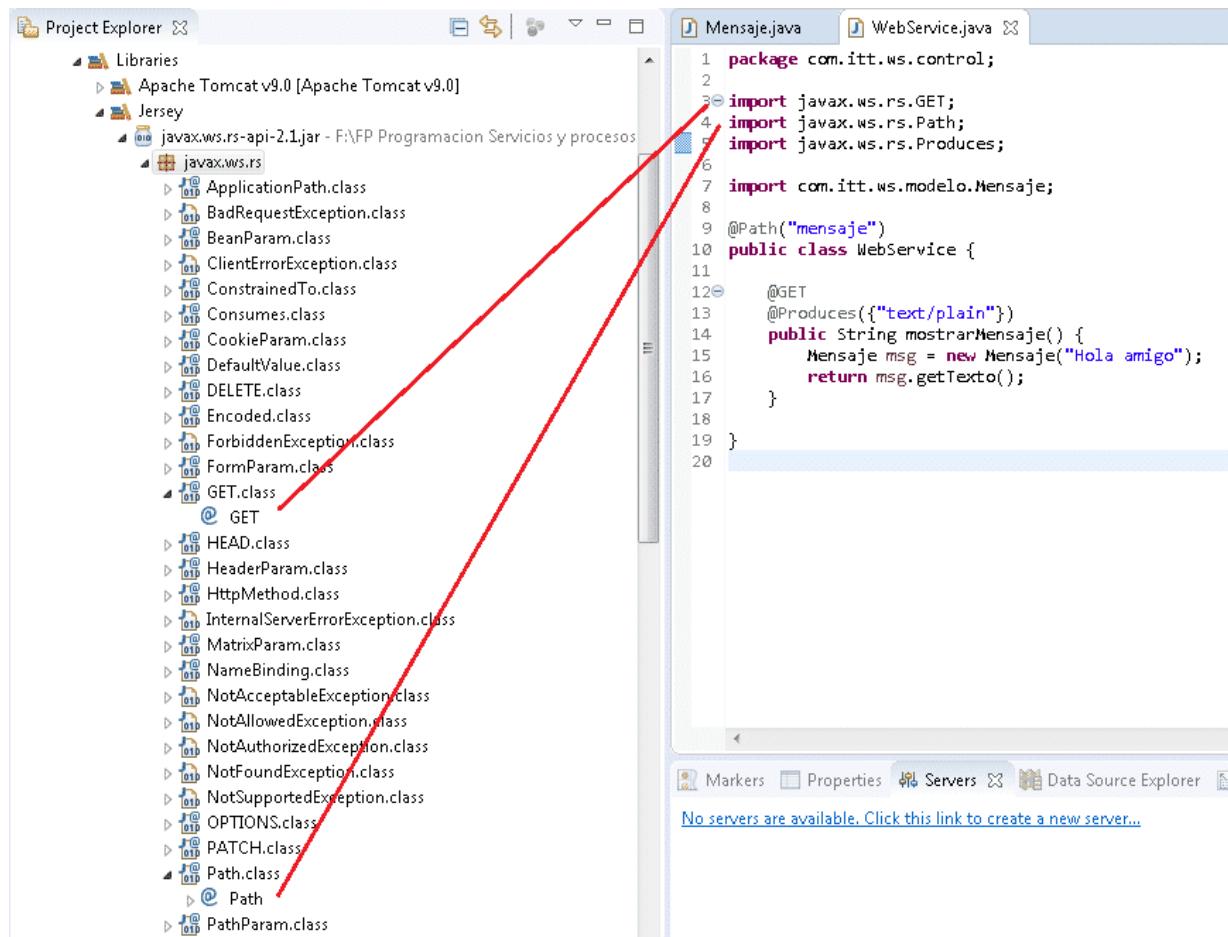
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

import com.itt.ws.modelo.Mensaje;

@Path("mensaje")
public class WebService {

    @GET
    @Produces({"text/plain"})
    public String mostrarMensaje() {
        Mensaje msg = new Mensaje("Hola amigo");
        return msg.getTexto();
    }
}
```

Observa que las anotaciones que estamos utilizando forman parte de la librería Jersey que has incluido en el proyecto:



Ahora, vamos a **analizar las anotaciones que hemos utilizado**:

- La anotación `@Path` indica la ruta que debe seguir el usuario para acceder al servicio web. También puede especificarse por métodos. Lo comprenderás mejor cuando veamos cómo se accede a un servicio web REST.
- La anotación `@GET` indica el método HTTP que se utilizará para recibir peticiones. El método `mostrarMensaje` atenderá peticiones de clientes de tipo `GET`, y también podemos utilizar `@POST` para atender peticiones de tipo `POST`.
- La anotación `@Produces` indica el tipo MIME de la respuesta del servidor.

Nuestro controlador tiene como misión aceptar peticiones de clientes y enviar las respuestas. La entrada de la petición (*request*) se efectúa por el método *mostrarMensaje* y la respuesta (*response*) se efectúa a través del valor que devuelve el mismo método. Estos controladores también pueden ser denominados ***servlet***.

---

Los *servlet* son objetos especiales que tienen como misión aceptar peticiones y enviar respuestas, utilizando el protocolo HTTP.

# Archivo web.xml

---

Los sitios web dinámicos guardan valores de configuración en un documento XML al que denominamos *Deployment Descriptor*.

El nombre físico de dicho fichero es **web.xml**.

Puesto que cuando se crea un proyecto web dinámico en la mayoría de los casos es para desarrollar un sitio web, el fichero *web.xml* viene configurado por defecto para ese fin. Pero no es este nuestro caso, y por lo tanto, personalizaremos su contenido como sea necesario para un proyecto cuyo cometido es contener servicios web REST.

En primer lugar, borra la entrada de la etiqueta *<welcome-file-list>* dejando el archivo *web.xml* de la siguiente manera:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="ht-
tp://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.or-
g/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd" id="We-
bApp_ID" version="4.0">
    <display-name>ProyectoRESTful</display-name>
</web-app>
```

Ahora, añade las entradas <servlet> y <servlet-mapping> dejando el archivo *web.xml* tal y como aparece en el siguiente bloque de código:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_4_0.xsd"
id="WebApp_ID" version="4.0">

    <display-name>ProyectoRESTful</display-name>

    <servlet>
        <servlet-name>JerseyRESTService</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.itt.ws.control</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>JerseyRESTService</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>

</web-app>
```

## Análisis del archivo web.xml

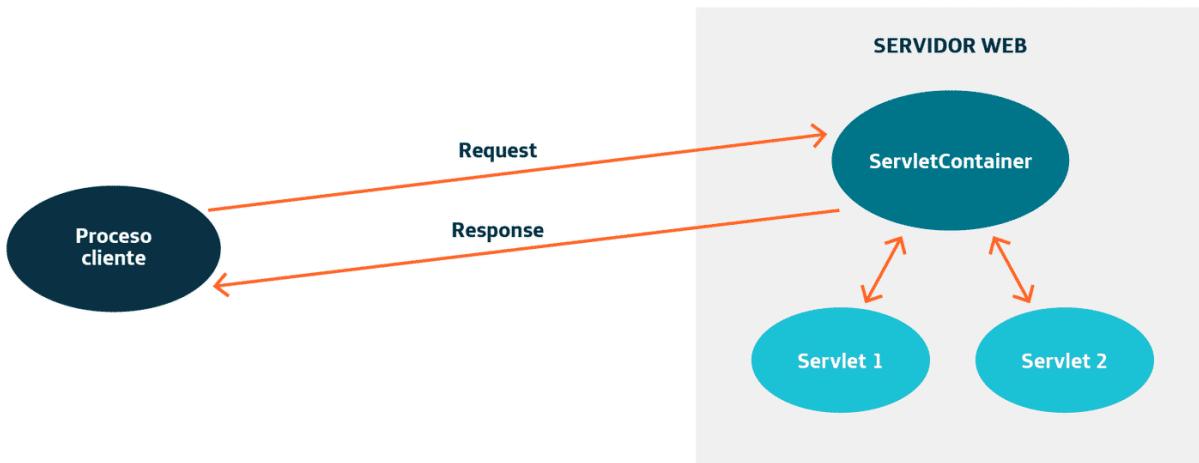
1

El objeto *ServletContainer*.

Una vez que el proyecto esté desplegado en Apache Tomcat, lo primero que hará el servidor cada vez que sea iniciado es localizar el fichero *web.xml*. Cada entrada de etiquetas XML <servlet> ... </servlet> encierra la configuración de un elemento muy relacionado con los proyectos web, al que denominamos *servlet*. Un *servlet* es un objeto capaz de recibir peticiones de clientes (*request*) y enviar una respuesta (*response*).

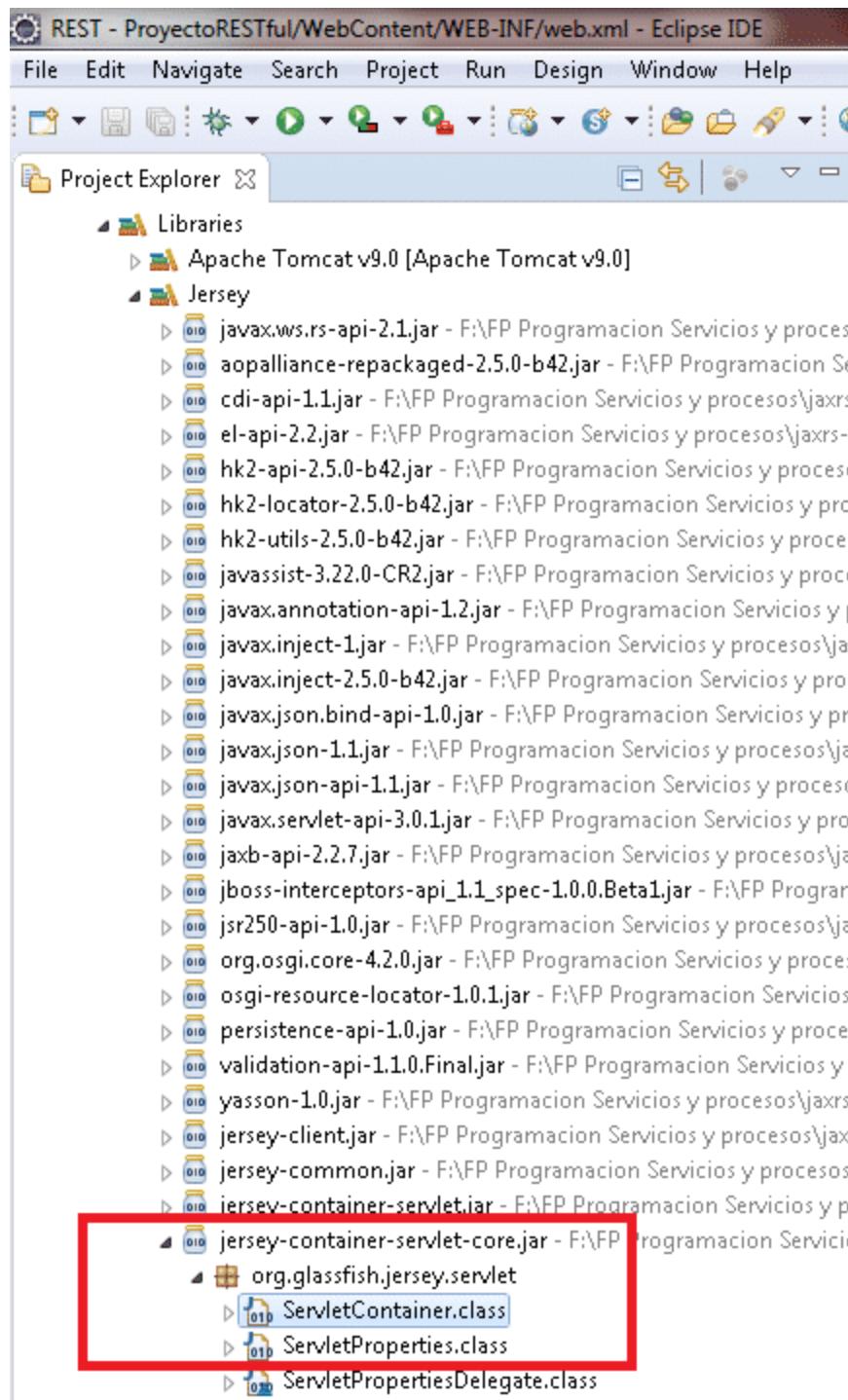
```
<servlet>
    <servlet-name>JerseyRESTService</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContai-
ner</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>com.itt.ws.control</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

RESTful gestiona todas las peticiones de clientes a través de un controlador o *servlet* único, el *ServletContainer*, que se encarga de recibir todas las peticiones y entregarlas al controlador secundario que corresponda (o *servlet* secundario). En nuestro caso, solo tenemos un controlador secundario, la clase *WebService* que hemos creado en el apartado anterior.



El *ServletContainer* recibe todas las peticiones y las entrega al controlador o *servlet* secundario que corresponda. El controlador secundario procesa la petición y emite una respuesta que enviará al *ServletContainer* para que la envíe, a su vez, al cliente.

Te proponemos que examines la librería Jersey dentro del proyecto Eclipse para localizar la clase *ServletContainer*.



La entrada <servlet-name> establece un nombre lógico para hacer referencia al *ServletContainer*. Aquí podríamos poner cualquier nombre que nos inventemos.

```
<servlet-name>JerseyRESTService</servlet-name>
```

En la entrada `<init-param>` estamos especificando el paquete Java donde se encuentran las clases que actúan como controladores, o *servlet* secundarios. Estos controladores, como ya hemos comentado, serán gestionados por el *ServletContainer*.

```
<init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>com.itt.ws.control</param-value>
</init-param>
```

```
<servlet>
    <servlet-name>JerseyRESTService</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.itt.ws.control</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

```
package com.itt.ws.control;

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

import com.itt.ws.modelo.Mensaje;

@Path("mensaje")
public class WebService {

    @GET
    @Produces({"text/plain"})
    public String mostrarMensaje() {
        Mensaje msg = new Mensaje("Hola Amigo");
        return msg.getTexto();
    }
}
```

La entrada `<load-on-startup>` determina, en caso de que tengamos varias entradas `<servlet>`, el orden en que el servidor va interpretando cada una de ellas.

```
<load-on-startup>1</load-on-startup>
```

2

### El *servlet-mapping*.

La entrada `<servlet-mapping>` establece el *mapping* asociado al *servlet* controlador principal. Con *mapping* nos referimos a la **forma en que los clientes acceden a los distintos recursos que presta el servidor**.

```
<servlet-mapping>
    <servlet-name>JerseyRESTService</servlet-name>
    <url-pattern>/*</url-pattern>
</servlet-mapping>
```

Con `/*` indicamos que el controlador principal atenderá todas las peticiones que partan de la raíz del sitio donde está alojado el servicio.

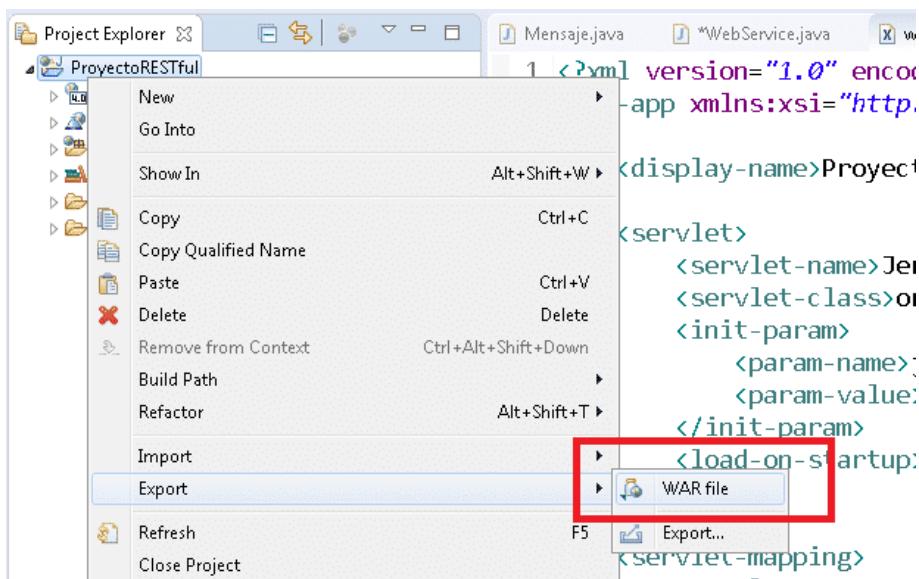
# Desplegar el proyecto en Apache Tomcat

Antes de desplegar el proyecto, recuerda que Apache Tomcat guarda sus aplicaciones en la carpeta `webapps`.

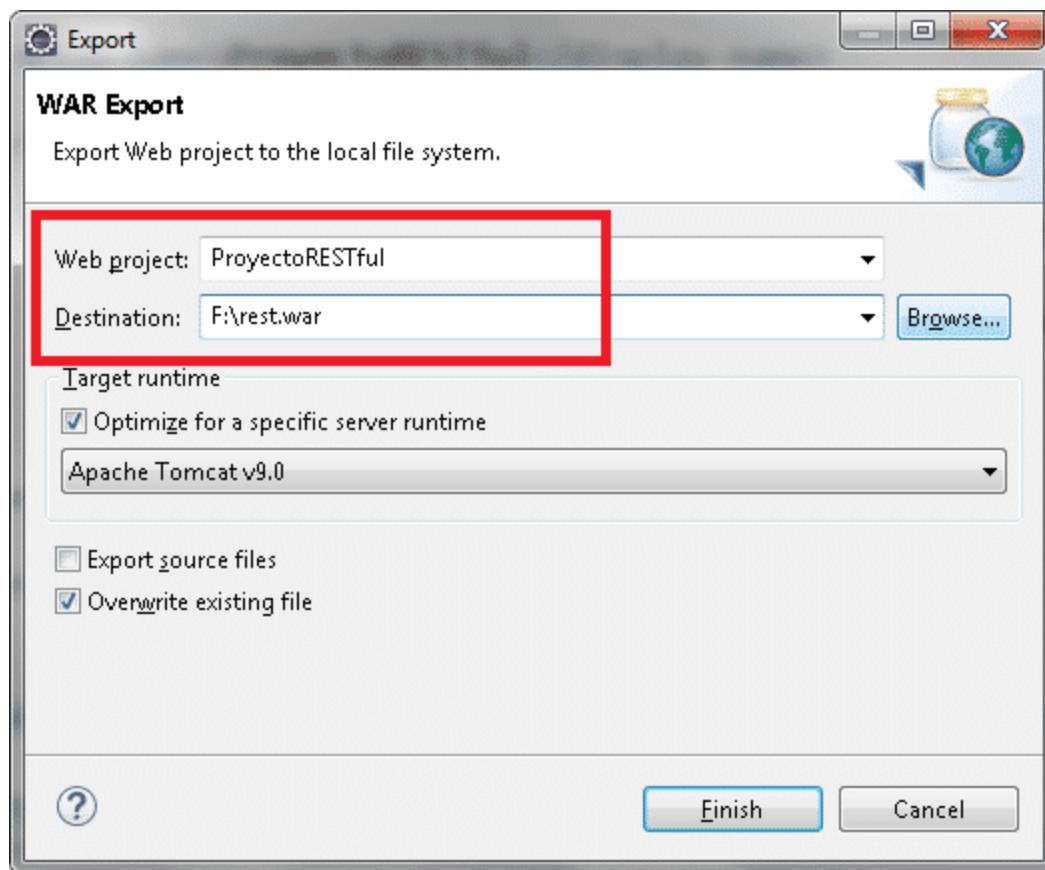
Justo allí es donde tendremos que dejar desplegada nuestra aplicación de servicios web REST.

## Exportación a WAR file

Una forma sencilla de desplegar nuestra aplicación en el formato que requiere el servidor Apache, es exportarla a formato `WAR file`. Puedes comenzar dicha exportación haciendo clic derecho sobre el nombre del proyecto y seleccionando en el menú contextual `Export / WAR file`.



En el cuadro de diálogo *Export* debes especificar el **nombre del proyecto que vas a exportar** y el **destino de la exportación**.



Como destino de la importación, en el ejemplo de la imagen hemos especificado F:\rest.war. Es importante tener en cuenta que el **nombre del archivo destino formará parte de la ruta para acceder a los recursos que presta el servicio web**, lo que significa que el acceso a los servicios comenzará con la ruta <http://localhost:8080/rest/>.



**CURIOSIDAD:** los archivos .war son como archivos .zip; de hecho puedes probar a renombrar la extensión, abrir el archivo con winzip para comprobar su contenido y, a continuación, volver a renombrarlo como .war.

## Despliegue en el servidor

Para comenzar, detén el servicio de Apache Tomcat, si es que está iniciado, tal y como aprendiste en el apartado *Funcionamiento básico de Apache Tomcat*. Una vez detenido, copia el archivo *rest.war* en la carpeta *webapps* de la instalación de Apache Tomcat.

Cuando inicies de nuevo el servicio, Apache Tomcat descomprimirá el archivo *rest.war* y creará una carpeta denominada *rest* dentro de *webapps*. **La carpeta *rest* es la nueva aplicación ya desplegada y lista.**

---

¡Tu nuevo servicio web ya está disponible!

## Poner a prueba el servicio web

Ya has terminado todo el proceso. Es el momento de comprobar si todo funciona correctamente.

Y, para comprobar que tu primer servicio web RESTful funciona, simplemente debes solicitarlo a través de tu navegador web, como puedes ver en la imagen:



Si analizamos la URL escrita, verás que está compuesta por las siguientes áreas:

- **localhost:8080**  
Host y puerto del servidor web: como hemos montado un servidor local, el host es *localhost*.

- **rest**  
Nombre de la subcarpeta dentro de `webapps` donde está ubicado el proyecto web.

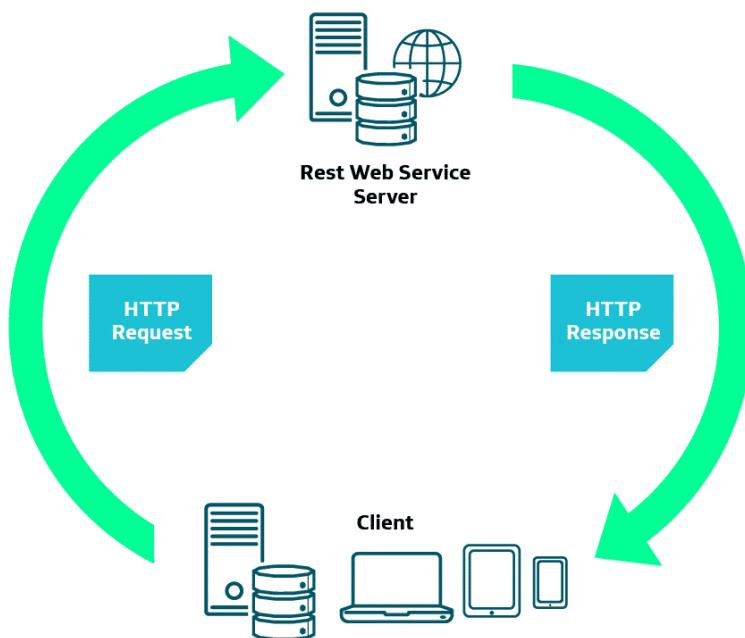
- **mensaje**  
Es el `path` especificado en nuestro controlador:

```
@Path("mensaje")
public class WebService {
    ...
}
```

---

Ahora que has puesto a prueba el funcionamiento de los servicios REST puedes comprobar que siguen la arquitectura cliente/servidor utilizando el protocolo HTTP.

- El cliente realiza una petición (*HTTP Request*).
- El servidor que hace de contenedor de servicios REST atiende dicha petición enviando una respuesta (*HTTP Response*).

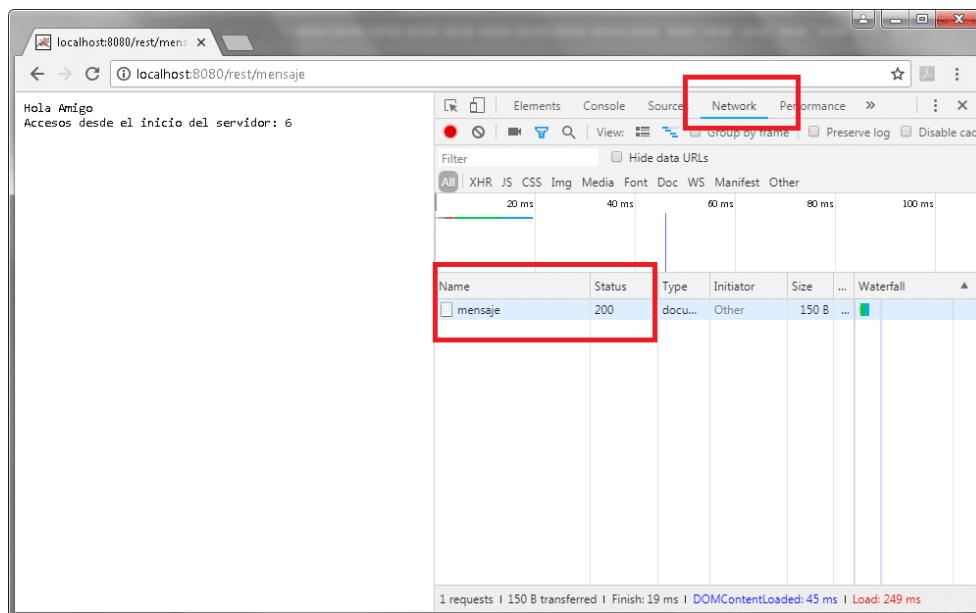


# Códigos de estado HTTP

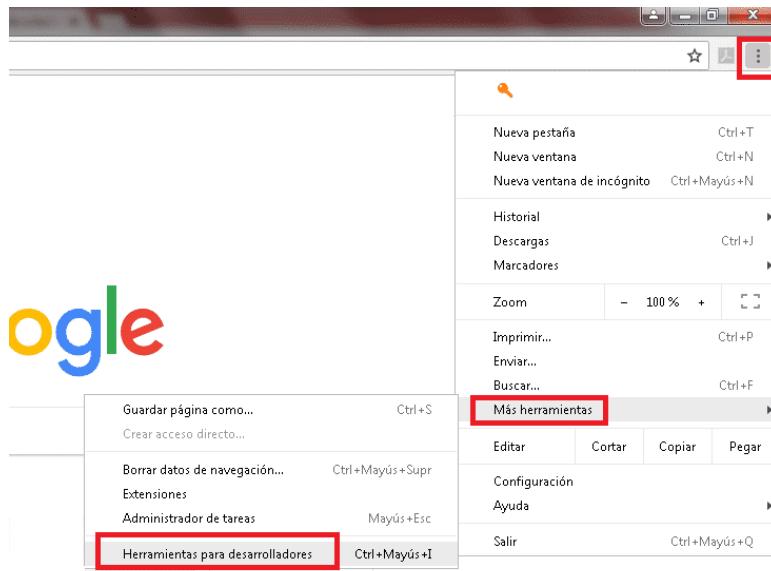
El protocolo HTTP también define una serie de códigos de respuesta.

Los códigos de respuesta que más solemos encontrar son el 404 (recurso no encontrado) o 200 (OK).

Las herramientas para desarrolladores de Google Chrome nos permiten ver en la ficha Network los métodos que se están ejecutando en las diferentes peticiones y los códigos de respuesta.



Puedes abrir las herramientas para desarrolladores pulsando **F12** o a través del menú *Más herramientas / Herramientas para desarrolladores*.



Los códigos de estado HTTP son números de tres cifras y pueden dividirse en cuatro grupos según la primera cifra:

1

**Códigos de estado que comienzan con un 1:** el servidor informa de que la petición actual todavía continua. El resto de cifras informan sobre el estado del procesamiento de la solicitud.

2

**Códigos de estado que comienzan por 2:** el servidor informa sobre una operación exitosa. Lo más común es el código 200 (OK).

3

**Códigos de estado que comienzan por 3:** el servidor ha recibido la solicitud, pero ha sido redireccionada de nuevo al cliente, que deberá tomar alguna decisión y volver a enviar la solicitud para que la operación sea efectuada con éxito.

4

**Códigos de estado que comienzan por 4:** el servidor informa de un error del cliente, en la mayoría de los casos porque se solicita un recurso que no existe y aunque el servidor ha recibido la solicitud, no puede atenderla. El caso más típico es el 404 (*no encontrado*).

5

**Códigos de estado que comienzan por 5:** ha ocurrido un error en el servidor durante la generación de la respuesta.

## Servicio web que devuelve HTML

---

En este apartado modificaremos nuestro servicio web para que devuelva el resultado en formato HTML, en lugar de texto plano.

Demostraremos en la práctica **cómo los servicios web REST ofrecen una separación entre dos datos que sirven y cómo los formatean a la hora de presentarlos.**

Podemos decir que **nuestro proyecto está dividido en dos capas:**

1

**Un modelo de datos:** que contendrá los datos que expone el servicio. Como nuestro ejemplo es muy simple, el modelo lo forma la clase *Mensaje*, que se limita a portar un texto y un contador de accesos.

2

**Un controlador:** que recibe la solicitud del cliente, la procesa y envía un resultado, consultando los datos del modelo. Finalmente, devuelve los datos al cliente formateados en la forma deseada. Para nuestro ejemplo, el controlador es la clase *WebService*.

Recuerda que la clase *WebService* atiende solicitudes a través del método *mostrarMensaje* que está encabezado por esta anotación:

```
@Produces({"text/plain"})
```

De este modo, estamos indicando que el resultado final que será devuelto al cliente será un texto sin formato, lo que solemos denominar **texto plano**.

---

## ¿Y si queremos devolver en mismo contenido, pero en formato de página web, es decir, en forma de código HTML?

En ese caso, basta con **modificar el tipo MIME de la anotación @Produces por text/html y devolver los datos del modelo encuadrados dentro de la estructura HTML de una página web**. Podría ser algo así:

```
package com.itt.ws.control;

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

import com.itt.ws.modelo.Mensaje;
```

```
@Path("mensaje")
public class WebService {

    @GET
    @Produces({"text/html"})
    public String mostrarMensaje() {
        Mensaje msg = new Mensaje("Hola Amigo");
        String html = "<!DOCTYPE html>";
        html = html + "<html>";
        html = html + "<head>";
        html = html + "<title>Mi primer servicio web REST</title>";
        html = html + "</head>";
        html = html + "<body style='background-color: aqua'>";
        html = html + "<h1>" + msg.getTexto() + "</h1>";
        html = html + "<h2>" + msg.getPeticiones() + "</h2>";
        html = html + "</body>";
        html = html + "</html>";

        return html;
    }

}
```

Como puedes apreciar, el controlador sigue consultando y exponiendo los datos del modelo, pero ahora enmarcados en una página HTML que estamos creando como un objeto *String*, que finalmente devuelve el método *mostrarMensaje*.

También hemos modificado ligeramente la clase *Mensaje* para que devuelva en líneas separadas el texto de saludo, y el contador de acceso en dos líneas de texto, separadas por medio de los métodos *getTexto()* y *getPeticiones()*.

```
package com.itt.ws.modelo;

public class Mensaje {
    private String texto;
    private static long peticiones;

    public Mensaje(String texto) {
        this.texto = texto;
        peticiones++;
    }

    public String getTexto() {
        return texto;
    }
}
```

```
}

public void setTexto(String texto) {
    this.texto = texto;
}

public String getPeticiones() {
    return "Accesos desde el inicio del servidor: " + peticiones;
}

}
```

Si quieres utilizar de nuevo el servicio web con estos cambios, tendrás que realizar los siguientes pasos:

- 1 Realiza de nuevo la exportación del proyecto a *WAR file* para actualizar tu fichero *rest.war*.
- 2 Detén el servicio de Apache Tomcat.
- 3 Copia otra vez el fichero *rest.war* en la carpeta *webapps* de Apache Tomcat.
- 4 Vuelve a iniciar el servicio de Apache Tomcat.

**¡Ojo!** Tendrás que repetir estos cuatro pasos cada vez que realices cambios en la aplicación.

Ya puedes acceder de nuevo al servicio a través del navegador, tal y como puedes ver en la siguiente imagen:



# Servicio web que devuelve XML o JSON

En el siguiente apartado vamos a añadir a nuestro proyecto de servicios web otra clase de modelo de datos y otro controlador.

Nuestra nueva clase de modelo se llamará *Persona* y usará las anotaciones de **JAXB** para que pueda convertirse fácilmente a formato XML.

```
package com.itt.ws.modelo;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlAttribute;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name = "persona")
public class Persona implements Serializable {

    private static final long serialVersionUID = 3575918590687798833L;

    private int idPersona;
    private String nombre;
    private String apellido;
    private int edad;

    public Persona() {

    }
```

```
public Persona(int id, String nombre, String apellido, int edad) {  
    this.idPersona = id;  
    this.nombre = nombre;  
    this.apellido = apellido;  
    this.edad = edad;  
}  
  
@XmlAttribute  
public int getIdPersona() {  
    return idPersona;  
}  
public void setIdPersona(int idPersona) {  
    this.idPersona = idPersona;  
}  
  
@XmlElement  
public String getNombre() {  
    return nombre;  
}  
public void setNombre(String nombre) {  
    this.nombre = nombre;  
}  
  
@XmlElement  
public String getApellido() {  
    return apellido;  
}  
public void setApellido(String apellido) {  
    this.apellido = apellido;  
}  
  
@XmlElement  
public int getEdad() {  
    return edad;  
}  
public void setEdad(int edad) {  
    this.edad = edad;  
}  
}
```

Para que nuestro servicio sea capaz de suministrar datos en formatos XML o JSON, necesitaremos de los tipos MIME; en este caso usaremos los tipos *text/xml* y *text/json*.

A continuación, te mostramos el **código del nuevo controlador** implementado en la clase *WebServicePersona*.

```
package com.itt.ws.control;

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

import com.google.gson.Gson;
import com.itt.ws.modelo.Persona;

@Path("datos")
public class WebServicePersona{

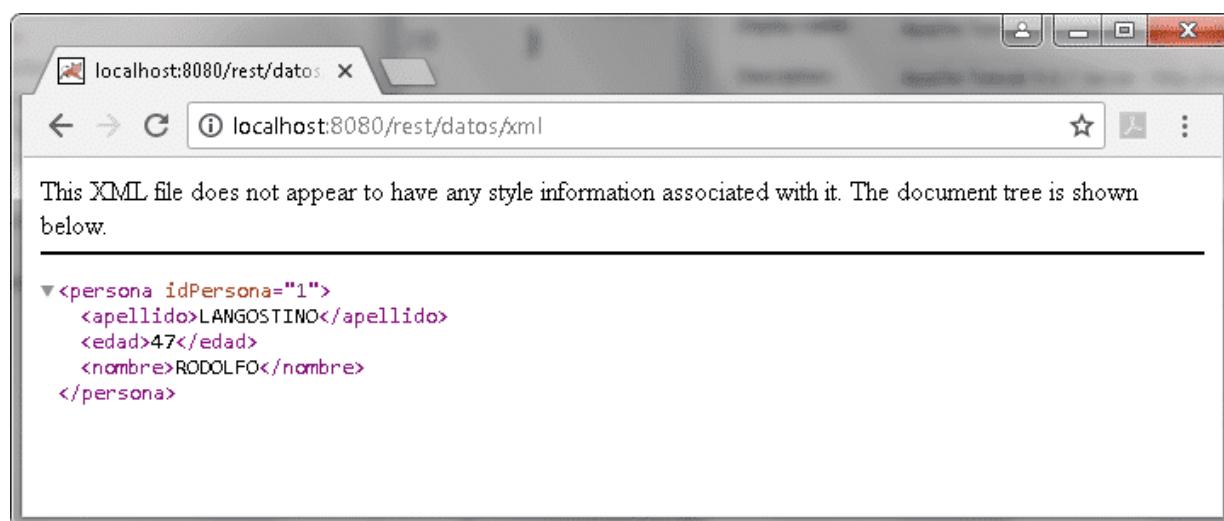
    @GET
    @Path("xml")
    @Produces({"text/xml"})
    public Persona personaXML() {
        Persona p = new Persona(1,"RODOLFO", "LANGOSTINO", 47);
        return p;
    }

    @GET
    @Path("json")
    @Produces({"text/json"})
    public String personaJSON() {
        Persona p = new Persona(1,"RODOLFO", "LANGOSTINO", 47);
        Gson gson = new Gson();
        String json = gson.toJson(p);
        return json;
    }
}
```

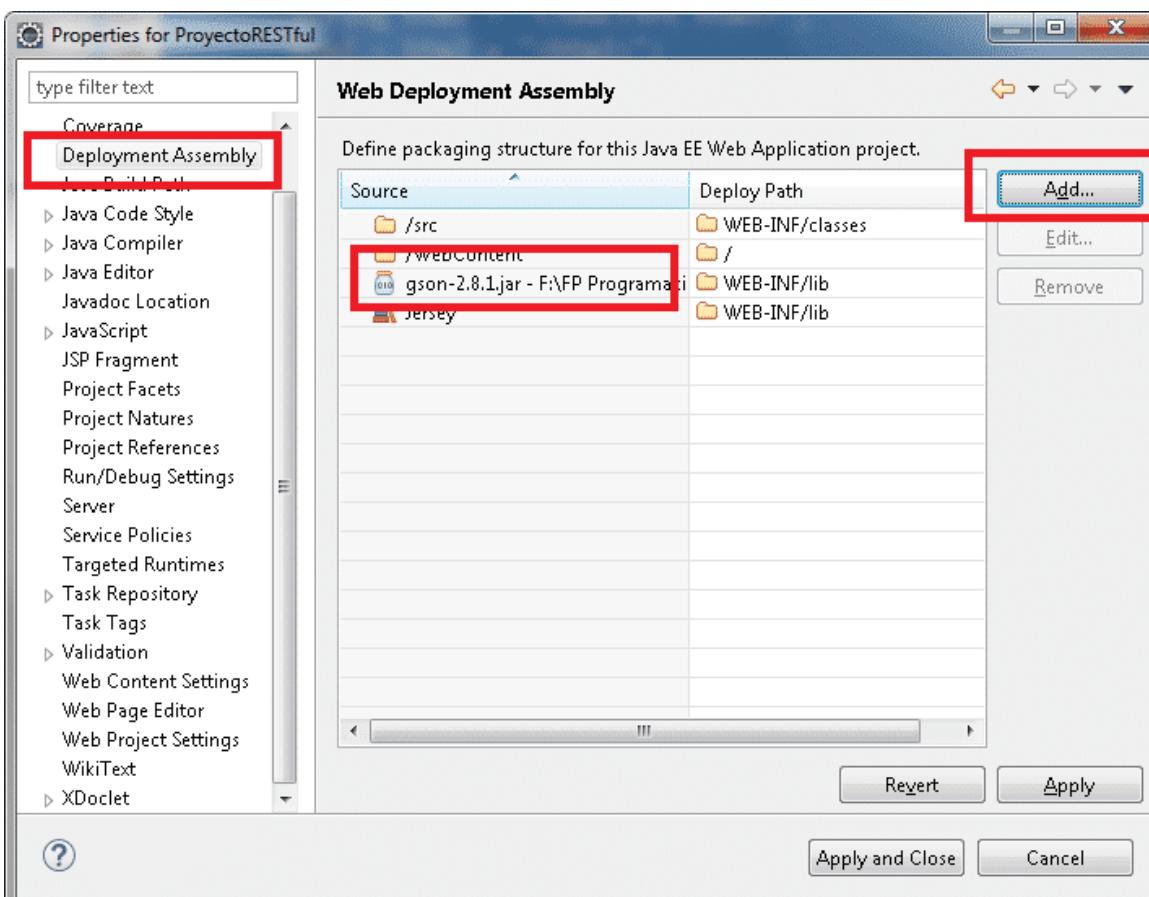
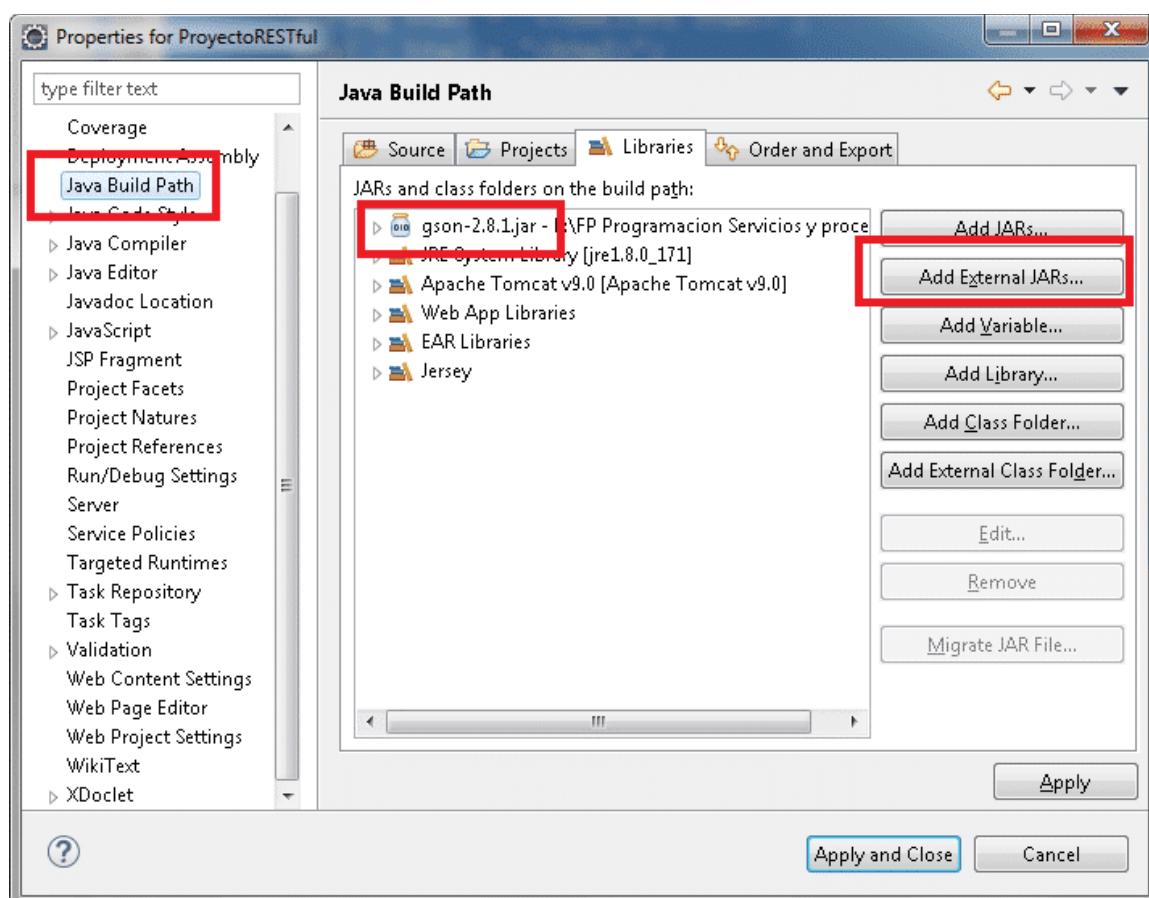
En este controlador hemos implementado dos métodos distintos: *personaXML()*, que suministrará datos en formato XML, y *personaJSON()*, que suministrará datos en formato JSON. Cada uno de los métodos atiende una petición HTML de tipo *GET*, ya que van encabezados por la anotación *@GET*. La anotación *@Produces* identifica el tipo MIME de la respuesta del servidor, y la anotación *@Path*, la ruta de acceso al recurso.

Puesto que la clase también está anotada con `@Path (datos)`, ambas rutas se concatenarán, de modo que nuestro servicio web ahora atiende tres tipos de peticiones distintas con los siguientes formatos:

- `http://localhost:8080/rest/mensaje`
- `http://localhost:8080/rest/datos/xml`
- `http://localhost:8080/rest/datos/json`



No olvides que, para que funcione el proyecto, tendrás que incluir la librería **GSON** y modificar el *Deployment Descriptor*, para que también sea incluida dicha librería en el archivo `rest.war`. Seguramente sabrás cómo hacerlo, y si no, las siguientes imágenes te refrescarán la memoria:



## Parámetros en la ruta (@PathParam)

Es posible que el cliente le pase al servidor un parámetro que formará parte de la ruta de la petición. Recuerda que la ruta viene establecida en la anotación `@Path`.

Vamos a volver a nuestro primer servicio, el que dejamos, finalmente, con formato HTML.

Modifica el código hasta dejarlo así:

```
package com.itt.ws.control;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;

import com.itt.ws.modelo.Mensaje;

@Path("mensaje/{nombre}")
public class WebService {

    @GET
    @Produces({"text/html"})
    public String mostrarMensaje(@PathParam("nombre") String nom) {
        Mensaje msg = new Mensaje("Hola " + nom);
        String html = "<!DOCTYPE html>";
        html = html + "<html>";
        html = html + "<head>";
        html = html + "<title>Mi primer servicio web REST</title>";
        html = html + "</head>";
        html = html + "<body style='background-color: aqua'>";
```

```
    html = html + "<h1>" + msg.getTexto() + "</h1>";
    html = html + "<h2>" + msg.getPeticiones() + "</h2>";
    html = html + "</body>";
    html = html + "</html>";

    return html;
}
```

Dentro de la ruta, y por medio de la anotación `@Path`, se puede especificar un **parámetro que permitirá al cliente comunicar algún dato al servidor**. Este parámetro llevará un nombre, que se especifica encerrado entre los caracteres `{` y `}`.

```
@Path("mensaje/{nombre}")
```

`mensaje` es parte de la ruta y `nombre` es el nombre de un parámetro que podrá reemplazarse por cualquier valor que suministre el usuario.

El método que se encargue de atender la solicitud **tendrá que incluir un parámetro que recoja el valor suministrado por el cliente**.

```
public String mostrarMensaje(@PathParam("nombre") String nom) {
```

El parámetro `nombre` de la ruta será recogido en la variable `nom`, que después se envía al constructor de la clase `Mensaje` para personalizar, así, el mensaje que se generará como respuesta.

**El resultado será el siguiente:**



# Parámetros fuera de la ruta (@QueryParam)

Otro sistema para que el cliente pueda enviar un dato al servidor durante la petición es a través de la anotación `@QueryParam`.

A diferencia de `@PathParam`, ahora el parámetro no forma parte de la ruta (`@Path`), sino que **será un parámetro con nombre** de los que se añaden a la petición tras el carácter de interrogación.

El controlador quedaría así:

```
package com.itt.ws.control;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;

import com.itt.ws.modelo.Mensaje;

@Path("mensaje")
public class WebService {

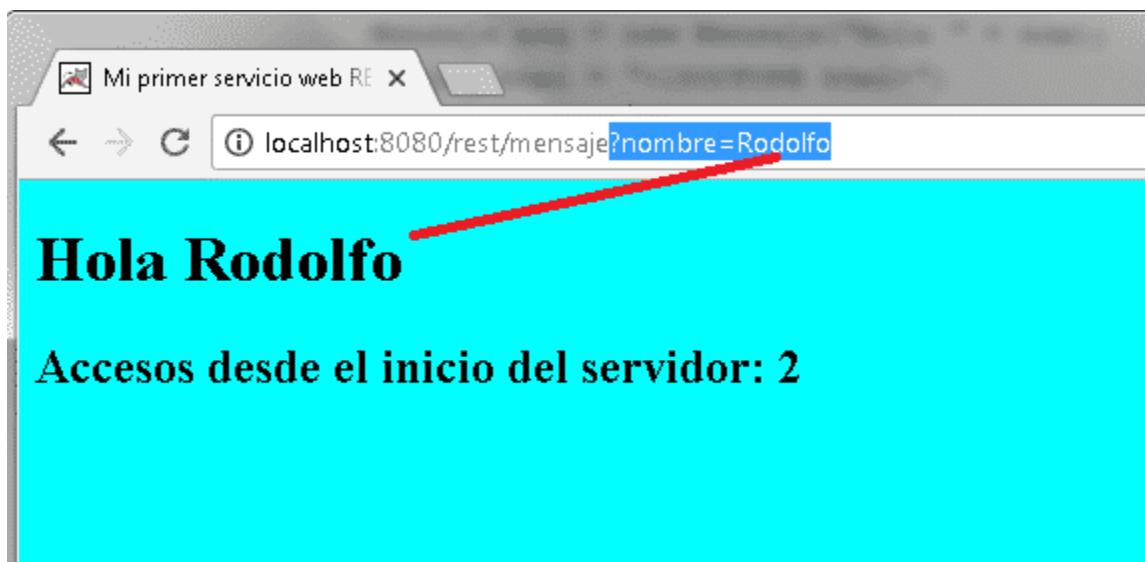
    @GET
    @Produces({"text/html"})
    public String mostrarMensaje(@QueryParam("nombre") String nom) {
        Mensaje msg = new Mensaje("Hola " + nom);
        String html = "<!DOCTYPE html>";
        html = html + "<html>";
        html = html + "<head>";
        html = html + "<title>Mi primer servicio web REST</title>";
        html = html + "</head>";
        html = html + "<body style='background-color: aqua'>";
        html = html + "<h1>" + msg.getTexto() + "</h1>";
        html = html + "<h2>" + msg.getPeticiones() + "</h2>";
        html = html + "</body>";
        html = html + "</html>";

        return html;
    }
}
```

Ahora, el parámetro también se recoge en la cabecera de la función *mostrarMensaje*, pero ya no está incluido en la anotación @Path.

---

**La solicitud al servicio desde el navegador será tal y como puedes ver en la siguiente imagen:**



## Consumo de servicios web RESTful

---

Hasta aquí, hemos accedido a los servicios que presta nuestro proyecto RESTful, escribiendo directamente la URL en el navegador.

Pero lo interesante es poder consumir estos servicios desde cualquier programa de cualquier tipo. Y en eso consisten los **sistemas distribuidos**, donde los programas que los componen hacen uso de servicios distribuidos en cualquier lugar de la red.

Nuestros ejemplos son muy simples, y es posible que no entiendas todavía el verdadero potencial de lo que estamos hablando. Nuestros servicios web, en lugar de devolver un simple valor, podrían estar consultando datos en una base de datos y devolviendo el resultado de una búsqueda, en función de unos parámetros suministrados por el cliente.

El siguiente código muestra un programa Java ejecutable, de consola, que consumirá el servicio web y mostrará la respuesta del servidor en pantalla.

```
import java.io.IOException;
import java.net.HttpURLConnection;
import java.net.URL;
import java.util.Scanner;

public class Principal {

    public static void main(String[] args) throws IOException {
        URL objURL = new URL("http://localhost:8080/rest/datos/xml?nombre=Carlos");
        HttpURLConnection conexion = (HttpURLConnection) objURL.openConnection();
        conexion.setRequestMethod("GET");

        int codigoHTTP = conexion.getResponseCode();
        System.out.println("Código: " + codigoHTTP);

        Scanner lector = new Scanner (conexion.getInputStream());
        String respuesta = "";
        while (lector.hasNextLine()) {
            respuesta = respuesta.concat(lector.nextLine());
        }

        lector.close();
        System.out.println(respuesta);
    }
}
```

Vamos a analizarlo por partes:

```
URL objURL = new URL("http://localhost:8080/rest/datos/xml?nombre=Carlos");
HttpURLConnection conexion = (HttpURLConnection) objURL.openConnection();
conexion.setRequestMethod("GET");
```

Aquí, estamos solicitando el servicio a través de una URL y un objeto *HttpURLConnection*. La petición se envía a través del método *GET*.

```
int codigoHTTP = conexion.getResponseCode();
System.out.println("Código: " + codigoHTTP);
```

Obtenemos el código HTTP de la petición y lo mostramos en pantalla. Salvo que ocurra algún error, el código será 200 (OK).

```
Scanner lector = new Scanner (conexion.getInputStream());
String respuesta = "";
while (lector.hasNextLine()) {
    respuesta = respuesta.concat(lector.nextLine());
}
```

El método `conexion.getInputStream()` nos devuelve el flujo de datos que necesitaremos para leer el mensaje del servidor. Dicho flujo de datos se lo pasamos a un objeto `Scanner`, que facilitará enormemente la lectura.

---

## Resumen

---

Has terminado la lección, repasemos los puntos más importantes que hemos tratado.

- Los **servicios web REST** utilizan el protocolo HTTP para la comunicación entre máquinas. Se caracterizan por no tener estado y corresponden al tipo de servicios stateless, es decir, que en cada solicitud que llega de un cliente, el servidor no tiene información de las solicitudes anteriores del mismo cliente.
- El cliente de servicios REST solicita recursos al servidor a través de una **URI** (*Uniform Resource Identifier*), que sirve para identificar inequívocamente un recurso dentro de la red.
- Los datos que transmite un servicio REST van en un formato específico, identificado mediante la nomenclatura **MIME** (image/jpeg, text/html, text/xml, video/mpeg, etc.).
- REST utiliza las acciones estándar definidas por el protocolo HTTP: **GET, POST, PUT, DELETE**, etc.
- Una característica propia del protocolo HTTP es que, para cada solicitud que realiza un cliente, se obtiene un **código de respuesta**.



**PROEDUCA**