

DAM  
Desarrollo de Aplicaciones Multiplataforma  
2º Curso

AD  
Acceso a Datos

UD 5  
Programación de componentes  
de acceso a datos  
(Parte 4)

IES BALMIS  
Dpto Informática  
Curso 2022-2023  
Versión 3 (01/2023)

## UD5 – Programación de componentes de acceso a datos

### ÍNDICE

#### 10. Crear componentes Servlets con Apache Tomcat

##### 10.1 Crear un Servlet de ejemplo

##### 10.2 Ampliar la funcionalidad del Servlet

##### 10.3 Servlet de acceso a datos

#### 11. Componentes para servidores de aplicaciones

#### 12. Crear componentes API Rest

##### 12.1 Asistente Restful Web Services from Patterns

##### 12.2 Creación de servicios sin acceso a datos

##### 12.3 Creación de servicios usando Bases de Datos

#### 13. Uso de JPA para componentes de servidores web

##### 13.1 API REST con JPA

##### 13.2 Añadir funcionalidad

##### 13.3 Mejorar el código

##### 13.4 Añadir métodos de actualización

##### 13.5 Añadir funcionalidad con XML

##### 13.6 Proyecto Web con JPA y BD con más de una tabla

##### 13.7 Proyecto Web con JPA y BD usando JPQL

##### 13.8 Proyecto Web con JPA y BD usando NamedQuery

##### 13.9 Proyecto Web con JPA y BD usando fechas

## 10. Crear componentes Servlets con Apache Tomcat

### 10.1 Crear un Servlet de ejemplo

Ya hemos visto los tipos de aplicaciones (escritorio y web), y algunos ejemplos de componentes de software como el de **Persona.jar** reutilizado en otras aplicaciones de escritorio y en aplicaciones web (**Web App**) con JSP. Tomcat permite también la ejecución de **Servlets** que atienden peticiones de tipo **GET, POST, PUT y DELETE**.

Veamos un ejemplo sencillo para crear un servicio Web (**Web Service**) de tipo **Servlet**.

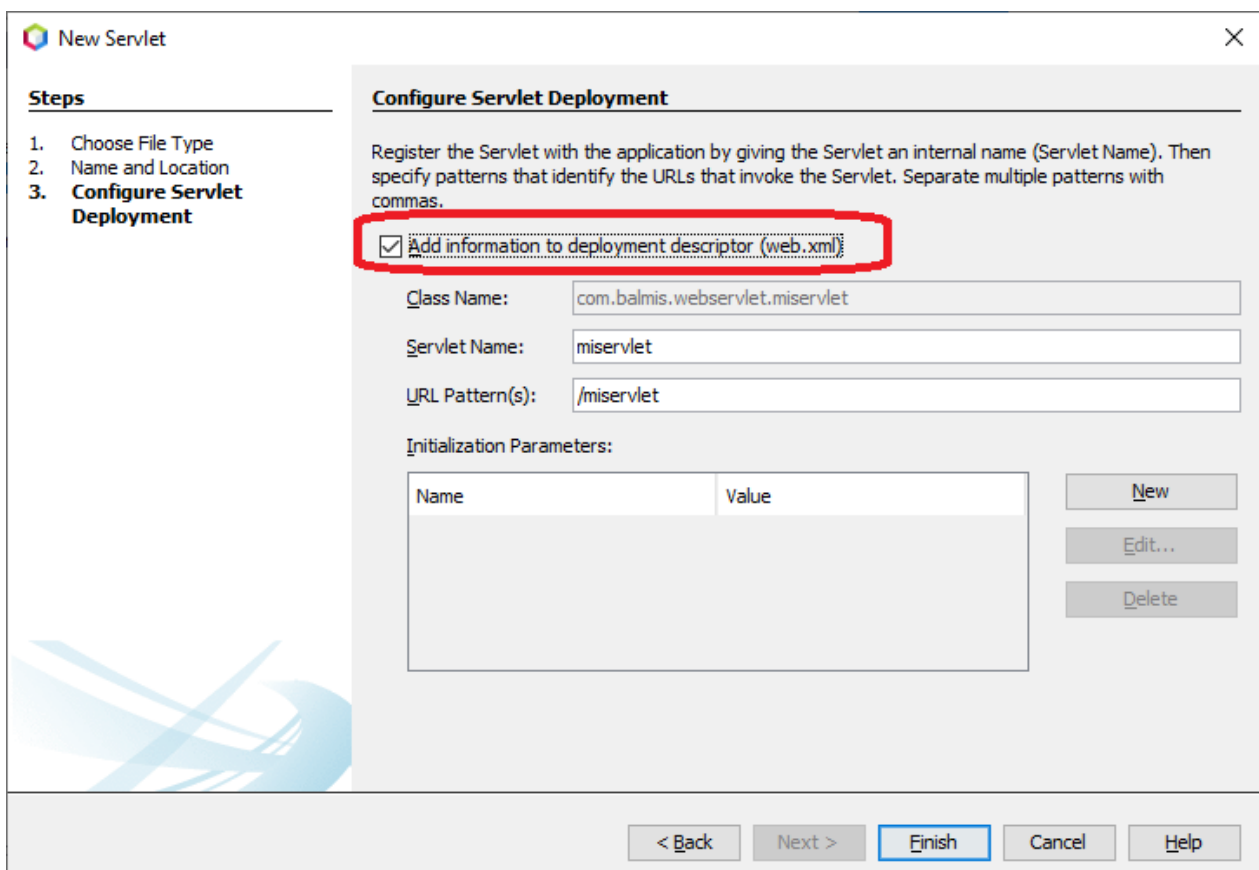
**Paso 1)** Creamos un proyecto de tipo

"**Java Ant** → **Java Web** → **Web Application**" denominado **WsServlet**  
con el servidor **Apache Tomcat**  
y Context Path **wsservlet**

**Paso 2)** Crear en **Source Package** el paquete **com.dam.webservlet**

**Paso 3)** Crear "**New** → **Other** → **Web** → **Servlet**" con el nombre **MiServlet.java**

En la última pantalla, incidir nombres en **minúsculas** y activar la casilla para crear el archivo de despliegue **web.xml**:



El asistente creará el método del servicio como **extends de HttpServlet** con un código de ejemplo que incluye los import de **javax.servlet** mientras que **Apache Tomcat 10** ya tiene los import de **jakarta.servlet**.

Además, en nuestro proyecto cambiaremos el método para mostrar **más información** disponible en el objeto **request**:

#### MiServlet.java

```
import java.io.IOException;
import java.io.PrintWriter;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class MiServlet extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                HttpServletResponse response) {
        response.setContentType("text/html;charset=UTF-8");
        try (PrintWriter out = response.getWriter()) {
            out.println("<!DOCTYPE html>");
            out.println("<html>");
            out.println("<head>");
            out.println("    <title>Servlet miservlet</title>");
            out.println("    <meta charset=\"UTF-8\">");
            out.println("</head>");
            out.println("<body>");
            out.println("<h1>Servlet SAMPLE</h1>");
            out.println("<p><b>URL:</b>" + request.getRequestURL() + "</p>");
            out.println("<p><b>URI:</b>" + request.getRequestURI() + "</p>");
            out.println("<p><b>MÉTODO:</b>" + request.getMethod() + "</p>");
            out.println("<p><b>Server Name:</b>" +
                request.getServerName() + "</p>");
            out.println("<p><b>Server Port:</b>" +
                request.getServerPort() + "</p>");
            out.println("<p><b>Context Path:</b>" + request.getProtocol() + "</p>");
            out.println("<p><b>Context Path:</b>" +
                request.getContextPath() + "</p>");
            out.println("<p><b>Servlet Path:</b>" +
                request.getServletPath() + "</p>");
            out.println("<p><b>Path Info:</b>");
            if (request.getPathInfo() != null) out.println(request.getPathInfo());
            out.println("</p>");

            out.println("<p><b>Query String:</b>");
            if (request.getQueryString() != null)
                out.println(request.getQueryString().replaceAll("&", "&amp;"));
            out.println("</p>");

            out.println("</body>");
            out.println("</html>");
        } catch (IOException ex) {
            System.out.println(ex.getMessage());
        }
    }
    [+] HttpServlet methods. Click on the + sign on the left to edit code
}
```

Pulsando en el [+] podemos añadir o quitar los métodos que permitimos que se atiendan.

**Paso 4)** Editar el archivo **index.html** para introducir un enlace a nuestro Servlet:

#### index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>SERVLET</title>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
  </head>
  <body>
    <div>Mi primer Servlet</div>
    <a href="http://localhost:8080/wsservlet/miservlet">
      http://localhost:8080/wsservlet/miservlet
    </a>
  </body>
</html>
```

**Paso 5)** Comprobar el archivo **web.xml** creado en **Configuration Files** del proyecto.

Editarlo y comprobar la información definida del **Servlet** que vamos a utilizar:

#### Web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd">

  <servlet>
    <servlet-name>miservlet</servlet-name>
    <servlet-class>com.dam.webservlet.MiServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>miservlet</servlet-name>
    <url-pattern>/miservlet</url-pattern>
  </servlet-mapping>

  <session-config>
    <session-timeout>
      30
    </session-timeout>
  </session-config>
</web-app>
```

**Paso 6)** Ejecutar y probar desde el navegador:

<http://localhost:8080/wsservlet/miservlet>

## 10.2 Ampliar la funcionalidad del Servlet

### URL

Si ampliamos la llamada a la URL siguiente con path ampliado y parámetros query:

<http://localhost:8080/wsservlet/miservlet/pathderecurso?param1=uno&param2=dos>

Comprobaremos que se muestra un error porque no hemos permitido en su definición que lo aceptemos. Para ello, cambiaremos el archivo **web.xml** en **Configuration Files** añadiendo en el patrón de url **/\***:

```
web.xml
...
<servlet-mapping>
  <servlet-name>miservlet</servlet-name>
  <url-pattern>/miservlet/*</url-pattern>
</servlet-mapping>
...
```

Si probamos de nuevo, ahora ya funcionará.

### Métodos

Además, si en el archivo **MiServlet.java** pulsamos al final en **[+]** podemos añadir o quitar los métodos que permitimos que se atiendan. Por defecto se atiende GET y POST.

Prueba su funcionamiento con **Postman** y comprueba que funciona correctamente en cualquiera de los métodos **GET** y **POST**, mientras que **PUT** o **DELETE** no funcionan.

Para que funcionen podemos añadir los métodos PUT y DELETE

```
MiServlet.java
...
@Override
protected void doPut(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}

@Override
protected void doDelete(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    processRequest(request, response);
}
...
```

Prueba de nuevo su funcionamiento con **Postman** y comprueba que funciona correctamente en cualquiera de los métodos **GET, POST, PUT** o **DELETE**.

## 10.3 Servlet de acceso a datos

Crear un **Servlet** de **API Rest** de acceso a datos implica.

- crear la estructura de **clases** (Frutas y Lista),
- implementar el **controlador** de métodos que realizan las funciones sobre los datos (**DAOFrutas**), y por último
- definir el **Servlet** que atiende los métodos REST (**ServletFrutas**) con el controlador que atiende cada método (**ControllerFrutas**):

En el siguiente proyecto ejemplo tenemos un **APIRest** que almacena la información en un **ArrayList** en RAM:

- **clases:** Frutas y Lista
- **controlador:** DAOFrutas
- **Servlet:** ControllerFrutas

Al recibir una petición Petición (Método REST + URL ) el flujo es:

- **ServletFrutas** recibe el método y llama al Controlador
- **ControllerFrutas** recibe los datos y solicita al DAO la acción
- **DAOFrutas** realiza la acción y devuelve el resultado (response) en cascada al Servlet para que la devuelva al cliente

### ServletFrutas

Comprobar el funcionamiento con **Postman** del API Rest de **ServletFrutas** y estudiar sus clases y métodos

<http://localhost:8080/servletfrutas/>

Muestra el index.html con la documentación del APIRest

<http://localhost:8080/servletfrutas/frutas> (GET)

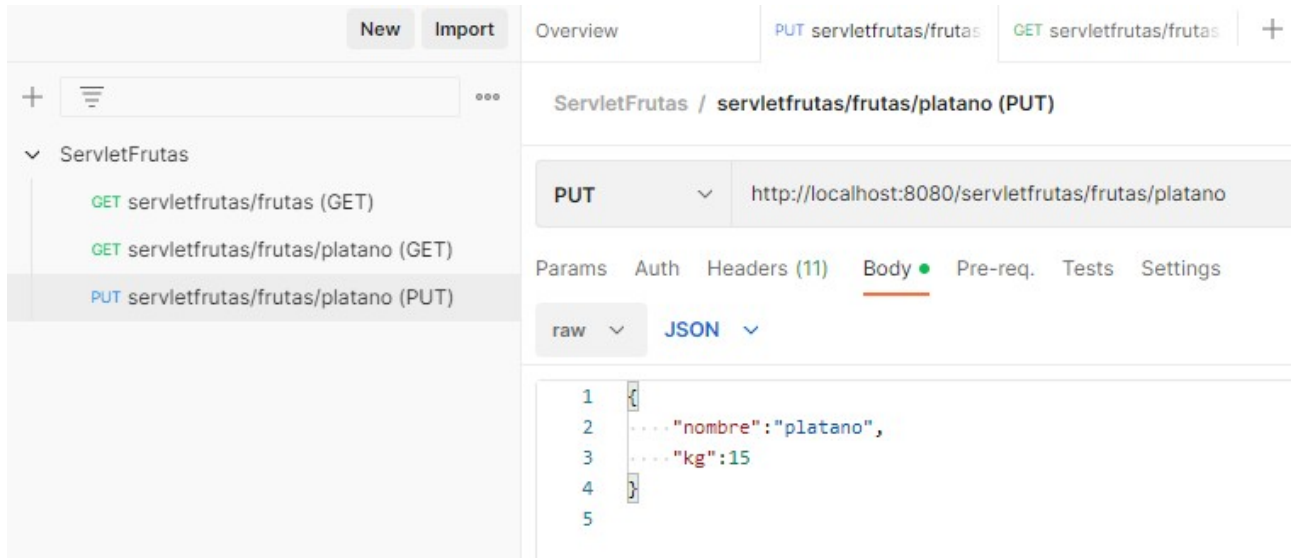
La primera vez que llamamos al APIRest con GET se llama a **DAOFrutas.cargaInicial** para cargar el ArrayList con algunas frutas, que son las que se muestran.

|                                     |   |
|-------------------------------------|---|
| <b>ServletFrutas.doGet</b>          | ServletFrutas.processRequest                    |
| <b>ServletFrutas.processRequest</b> | ControllerFrutas.atenderGET(request, response); |
| <b>ControllerFrutas.atenderGET</b>  | DAOFrutas.getAll()                              |
| <b>DAOFrutas.getAll()</b>           | DAOFrutas.cargaInicial                          |

<http://localhost:8080/servletfrutas/frutas/platano> (GET)

<http://localhost:8080/servletfrutas/frutas/platano> (PUT)

Se puede comprobar en el método **DAOFrutas.put** que solo se actualiza si los Kg son distintos



Los **Servlets** son la base del funcionamiento de un servidor web que utilice Java para generar contenido dinámico y atender algunos métodos definidos por la arquitectura **REST**.



## 11. Componentes para servidores de aplicaciones

### Plataformas Java

Hemos comprobado también que existe una equivalencia entre PHP y JSP para la creación de páginas web dinámicas, pero la realidad es que Java es mucho más potente.

Java tiene dos plataformas para ejecutar aplicaciones:

- **Java SE** (Java Standard Edition) o **Jakarta SE** (Jakarta Standard Edition)
- **Java EE** (Java Enterprise Edition) o **Jakarta EE** (Jakarta Enterprise Edition)

La plataforma **Java SE (Java Standard Edition)** es la base de la tecnología Java, incluyendo herramientas de desarrollo, la Máquina Virtual Java (JVM) y la documentación para la programación.

Está orientada a la creación de aplicaciones cliente pero no incluye soporte par tecnologías de Internet.

Esto implica que son J2SE no podemos desarrollar aplicaciones web de Java.

La plataforma **Java EE (Java Enterprise Edition)** añade a Java la funcionalidad necesaria para convertirse en un lenguaje orientado al desarrollo de aplicaciones y servicios en Internet.

Con Java EE se pueden desarrollar sitios web complejos bajo la tecnología Java mediante la utilización de **JSP** (lenguaje de script de servidor para crear páginas web dinámicas como las de PHP a ASP) y **Servlets** (scripts CGI en el servidor similares a los creados con PERL)

**Jakarta EE** es la nueva plataforma de código abierto (open source) de Java EE y cambia de nombre porque Oracle, a pesar de entregar el proyecto, no permite que lo usen.

### Tipos de Componentes Java

Vemos ahora una primera clasificación de los componentes

**JavaBean** es un componente de software para la plataforma Java SE.

**EJB (Enterprise JavaBean)** es un componente de software para la plataforma Java EE que se despliega sobre un contenedor de EJB, incluido en un servidor de aplicaciones.

Si desplegamos una aplicación con componentes **EJB** en Apache Tomcat veremos que no funciona.

**Apache Tomcat** es un servidor web, desarrollado bajo el proyecto Jakarta en la Apache Software Foundation, con soporte de **servlets** y **JSP** (JavaServer Pages), pero no es un servidor de aplicaciones, por tanto, habría que incluirle módulos adicionales para ampliar sus posibilidades. Se usa como servidor web autónomo en entornos con alto nivel de tráfico y alta disponibilidad.

**Apache Tomcat no es un Servidor Web de Aplicaciones Java EE**, pero como nosotros no crearemos EJBs, sino **WAR**, podremos usarlo sin problemas añadiendo las librerías necesarias al proyecto.

### **Servidor Web de Aplicaciones Java**

Los más usados son:

**WildFly**, anteriormente **JBoss**, es un servidor de aplicaciones J2EE de código abierto implementado en Java. Ofrece una plataforma de alto rendimiento para aplicaciones de e-business.

<https://www.wildfly.org/>

Sus características principales son:

- Es distribuido bajo licencia de código abierto GPL/LGPL.
- Proporciona un nivel de confianza suficiente para ser utilizado en entornos empresariales.
- Es un servicio incrustable, por ello está orientado a la arquitectura en servicios.
- Servicio del middleware para objetos Java.
- Soporte completo para JMX (Java Management eXtensions).

**GlassFish** es un servidor de aplicaciones de código abierto que implementa funcionalidades de Java EE.

<https://javaee.github.io/glassfish/>

Es gratuito y de código abierto, desarrollado por Sun Microsystems y tiene como base al servidor Sun Java Application Server de Oracle Corporation, un derivado de Apache Tomcat.

**Apache TomEE** es un servidor de aplicaciones de código abierto que implementa funcionalidades de Java EE.

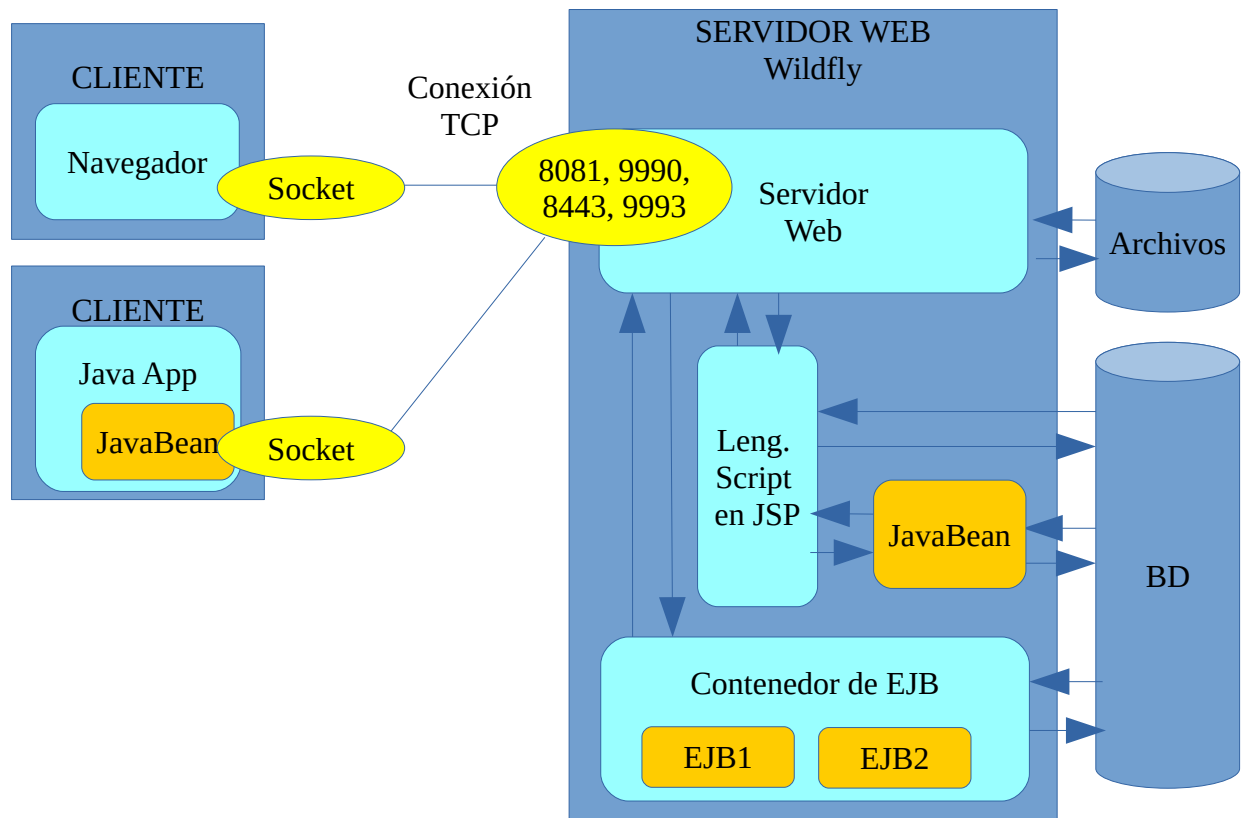
<https://tomee.apache.org/>

Es gratuito y de código abierto, desarrollado por Apache Software Foundation.

En principio daría igual usar cualquiera de ellos, porque tienen una instalación muy sencilla y se integran perfectamente en Netbeans (excepto Apache TomEE), permitiéndonos ejecutar el debugger durante la ejecución.

**Azure** nos facilita servidores virtualizados **JBoss EAP** (que es compatible con **Wildfly**) y **Apache Tomcat**.

De esta forma, el esquema para la programación en un servidor de aplicaciones Java como **Wildfly** quedaría:



Como podemos ver, el contenedor de aplicaciones EJB (Enterprise JavaBean) ejecutará los componentes desarrollados y generará una salida de datos para el servidor web.

## 12. Crear componentes API Rest

Los servicios de API Rest se pueden implementar con diferentes tipos de tecnologías (hemos visto que se puede incluso con PHP en un servidor web Apache), pero en los servidores de aplicaciones Java (como **Wildfly**, **Glassfish** o **Apache TomEE**) se suelen implementar con componentes de software de tipo **EJB**, lo que proporciona mayores prestaciones como mayor escalabilidad, mejor gestión de memoria, posibilidad de balanceo de carga, pool de conexiones, ...

Los componentes **EJB** se empaquetan en archivos **EAR**.

**EAR (Enterprise Application aRchive)** es un formato de archivo utilizado por Java EE para empaquetar uno o más módulos en un único archivo, de modo que la implementación de los distintos módulos en un servidor de aplicaciones se realice de manera simultánea y coherente. También contiene archivos XML llamados descriptores de implementación que describen cómo implementar los módulos.

En nuestro caso lo realizaremos un **APIRest** como una **Web App** en un servidor **Apache Tomcat**, igual que lo hace **Spring Boot**. Esta es la forma típica para implementar **microservicios**.

### 12.1 Asistente Restful Web Services from Patterns

Para crear un proyecto de APIRest simple, sin JPA de acceso a datos, podemos usar el asistente "**Restful Web Services from Patterns**".

Vamos a crear un proyecto "**Java Web**" denominado **ApiRestHolaMundo** que proporcione un API Rest de Java que se instalará (deploy) en un Servidor de Web de **Apache Tomcat** con el asistente de NetBeans.

La aplicación tendrá una URL para mostrar el contenido JSP:

<http://dominio/basico>

Y otra URL para activar el servicio API Rest:

<http://dominio/basico/recursos/holamundo>

## **Paso 1 – Crear proyecto de tipo Java Web**

Utilizaremos el asistente "**Java with Ant → Java Web → Web Application**":

The image displays three sequential screenshots of the NetBeans 'New Web Application' wizard, illustrating the steps to create a Java Web project.

**Screenshot 1: Name and Location**

The 'Pasos' (Steps) list on the left shows the current step is '2. Name and Location'. The main panel contains the following fields:

- Project Name:** ApiRestHolaMundo
- Project Location:** C:\Users\VICENTE\Documents\NetBeansProjects (with a 'Browse...' button)
- Project Folder:** ::\Users\VICENTE\Documents\NetBeansProjects\ApiRestHolaMundo
- ☐ Use Dedicated Folder for Storing Libraries
- Libraries Folder:** (with a 'Browse...' button)

A note at the bottom states: 'Different users and projects can share the same compilation libraries (see Help for details)'.

**Screenshot 2: Server and Settings**

The 'Steps' list on the left shows the current step is '3. Server and Settings'. The main panel contains the following fields:

- Add to Enterprise Application:** <None>
- Server:** Apache Tomcat (local) (with an 'Add...' button)
- Java EE Version:** Java EE 7 Web
- Note:** Source Level 7 will be set for Java EE 7 project.
- Context Path:** /basico

**Screenshot 3: Frameworks**

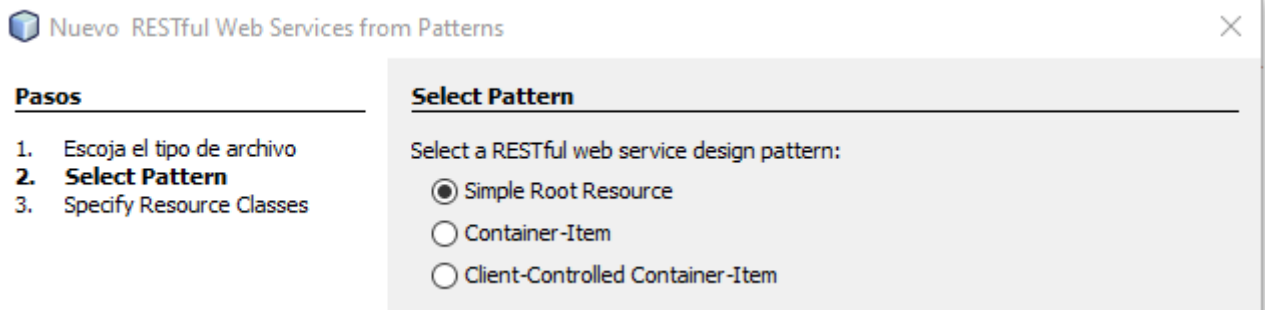
The 'Steps' list on the left shows the current step is '4. Frameworks'. The main panel contains the following text and options:

Select the frameworks you want to use in your web application.

- ☐ Spring Web MVC
- ☐ JavaServer Faces
- ☐ Struts 1.3.10

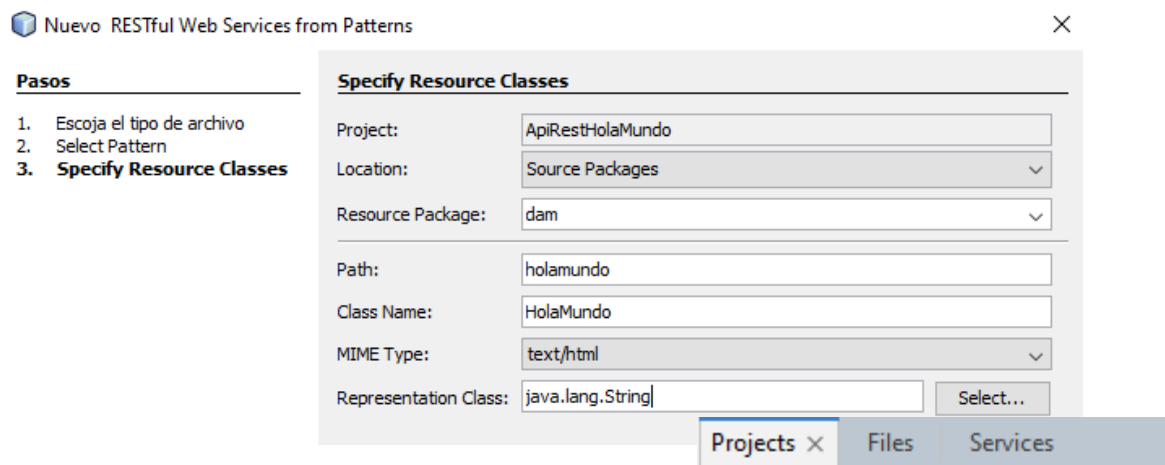
## Paso 2 – Añadir archivo "Web Services"

Añadiremos un archivo nuevo en "Source Packages" de la categoría "Web Services" del tipo de archivo "Restful Web Services from Patterns" con el patrón "Simple Root Resource".



A continuación indicaremos:

- **Resource Package:** dam
- **Path:** holamundo
- **Class Name:** HolaMundo
- **MIME Type:** text/html
- **Representation Class:** java.lang.String



El asistente nos crea el siguiente proyecto con:

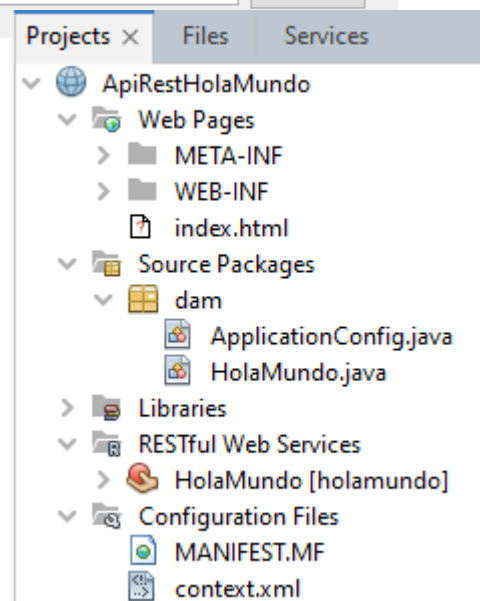
**ApplicationConfig.java**

**HolaMundo.java**

Cuando lo ejecutemos, además tendremos:

**RESTful Web Services => HolaMundo**

**Configuration Files**



### Paso 3 – Libraries

Como **Apache Tomcat** ya trabaja con **jakarta.ws** en vez de **javax.ws**, pero NetBeans tiene en sus Libraries todavía **JAX-RS 2.1.6** y **Jersey 2.34**, por lo que **debemos eliminarlos y añadir manualmente los jar de jakarta** mediante la carpeta **lib** del proyecto.

Otra opción sería cambiar los jar en **Tools** → **Libraries** pero haremos el proyecto más portable si dependemos solo de la carpeta **lib** del proyecto.

Luego, cambiaremos:

- todos los import **javax.ws** por **jakarta.ws** y
- todos los **@javax.ws** por **@jakarta.ws**

### Paso 4 – Preparar código

Veamos el contenido de los archivos principales obtenidos:

|                               |   |
|-------------------------------|---|
| <b>ApplicationConfig.java</b> | Prepara el servicio de API Rest añadiendo el <b>ApplicationPath</b> al <b>target</b> a la URL base. Por defecto es <b>webresources</b> pero puede cambiarse por <b>recursos</b> . |
| <b>HolaMundo.java</b>         | Es el interfaz de métodos que se implementan y estarán disponibles para el servicio: GET  |

En **HolaMundo.java** eliminaremos el PUT y modificaremos el GET:

| <b>HolaMundo.java</b>  |  |
|--|--|
| <pre>@GET @Produces(MediaType.TEXT_HTML) public String getHtml() {     return "&lt;html&gt;&lt;body&gt;&lt;h1&gt;Hola Mundo!!&lt;/h1&gt;&lt;/body&gt;&lt;/html&gt;"; }</pre> |  |

Ya está terminado y podemos ejecutarlo. El interfaz (API) que nos ofrece es:

|            |                            |                                   |
|------------|----------------------------|-----------------------------------|
| <b>GET</b> | <b>/recursos/holamundo</b> | Obtiene una texto en formato html |
|------------|----------------------------|-----------------------------------|

Donde podemos localizar:

|                   |  |
|-------------------|--|
| <b>/basico</b>    | en Configuration Files → context.xml       |
| <b>/recursos</b>  | en Source Packages → dam.ApplicationConfig |
| <b>/holamundo</b> | en Source Packages → dam.HolaMundo         |

Por último, cambiaremos el **body** de **index.html** par tener el enlace al servicio:

**index.html – Cambiamos el body**

```
<body>
  <div>ApiRestHolaMundo</div>
  <div>
    <a href="/basico/recursos/holamundo">/basico/recursos/holamundo</a>
  </div>
</body>
```

Para probarlo, lo ejecutaremos y accederemos desde el navegador a:

<http://localhost:8080/basico/>  
<http://localhost:8080/basico/recursos/holamundo>

**12.2 Creación de servicios sin acceso a datos**

Para crear servicios REST utilizaremos anotaciones en los métodos:

|                    |   |
|--------------------|---|
| <b>@MÉTODO</b>     | Indica la operación que ejecutará el método: GET, POST, PUT DELETE.       |
| <b>@Consumes</b>   | Son los formatos que soporta el método en los datos recibidos             |
| <b>@Produces</b>   | Son los formatos que soporta el método en los datos enviados              |
| <b>@Path</b>       | Es el Path que se añade al Target en la URL                               |
| <b>@PathParam</b>  | Es la anotación que asocia un parámetro del método a un elemento del Path |
| <b>@QueryParam</b> | Es la anotación que asocia un parámetro después del carácter ?            |

**API Rest – Anotaciones**

<https://docs.oracle.com/cd/E19798-01/821-1841/6nmq2cp1v/index.html>

**ApiRestMath**

Como ejemplo, podemos crear un proyecto "Java Web" con **Apache Tomcat** denominado **ApiRestMath** que admita el método **GET** en la URL:

<http://localhost:8080/math/apirest/aritmetica/sumar/3/5>  
<http://localhost:8080/math/apirest/aritmetica/dividir/8/3?decimales=2>

El método tendrá la anotación:

```
@Path("/sumar/{operando1}/{operando2}")
@Path("/dividir/{operando1}/{operando2}")
```

|   |             |                       |  |
|---|-------------|-----------------------|--|
| 1 | /math       | Al crear proyecto     | en Configuration Files → context.xml   |
| 3 | /apirest    | Editando archivo      | en Source Packages → ApplicationConfig |
| 2 | /aritmetica | Al ejecutar asistente | en Source Packages → MathResource      |



## Paso 1 – Crear proyecto de tipo Java Web

Utilizaremos el asistente "**Java with Ant** → **Java Web** → **Web Application**":

The figure shows three sequential screenshots of the 'New Web Application' wizard in NetBeans:

- Step 1: Name and Location**
  - Project Name: ApiRestMath
  - Project Location: C:\Users\VICENTE\Documents\NetBeansProjects (with a 'Browse...' button)
  - Project Folder: C:\Users\VICENTE\Documents\NetBeansProjects\ApiRestMath
  - ☐ Use Dedicated Folder for Storing Libraries
  - Libraries Folder: (empty field with a 'Browse...' button)
  - Note: Different users and projects can share the same compilation libraries (see Help for details).
- Step 2: Server and Settings**
  - Add to Enterprise Application: <None>
  - Server: Apache Tomcat (local) (with an 'Add...' button)
  - Java EE Version: Java EE 7 Web
  - Note: Source Level 7 will be set for Java EE 7 project.
  - Context Path: /math
- Step 3: Frameworks**
  - Select the frameworks you want to use in your web application.
  - Options:
    - ☐ Spring Web MVC
    - ☐ JavaServer Faces
    - ☐ Struts 1.3.10

## Paso 2 – Añadir archivo "Web Services"

Ahora añadiremos "**File New**" de la categoría "**Web Services**" del tipo de archivo "**Restful Web Services from Patterns**" con el patrón "**Simple Root Resource**".

The figure shows the 'New RESTful Web Services from Patterns' wizard in NetBeans:

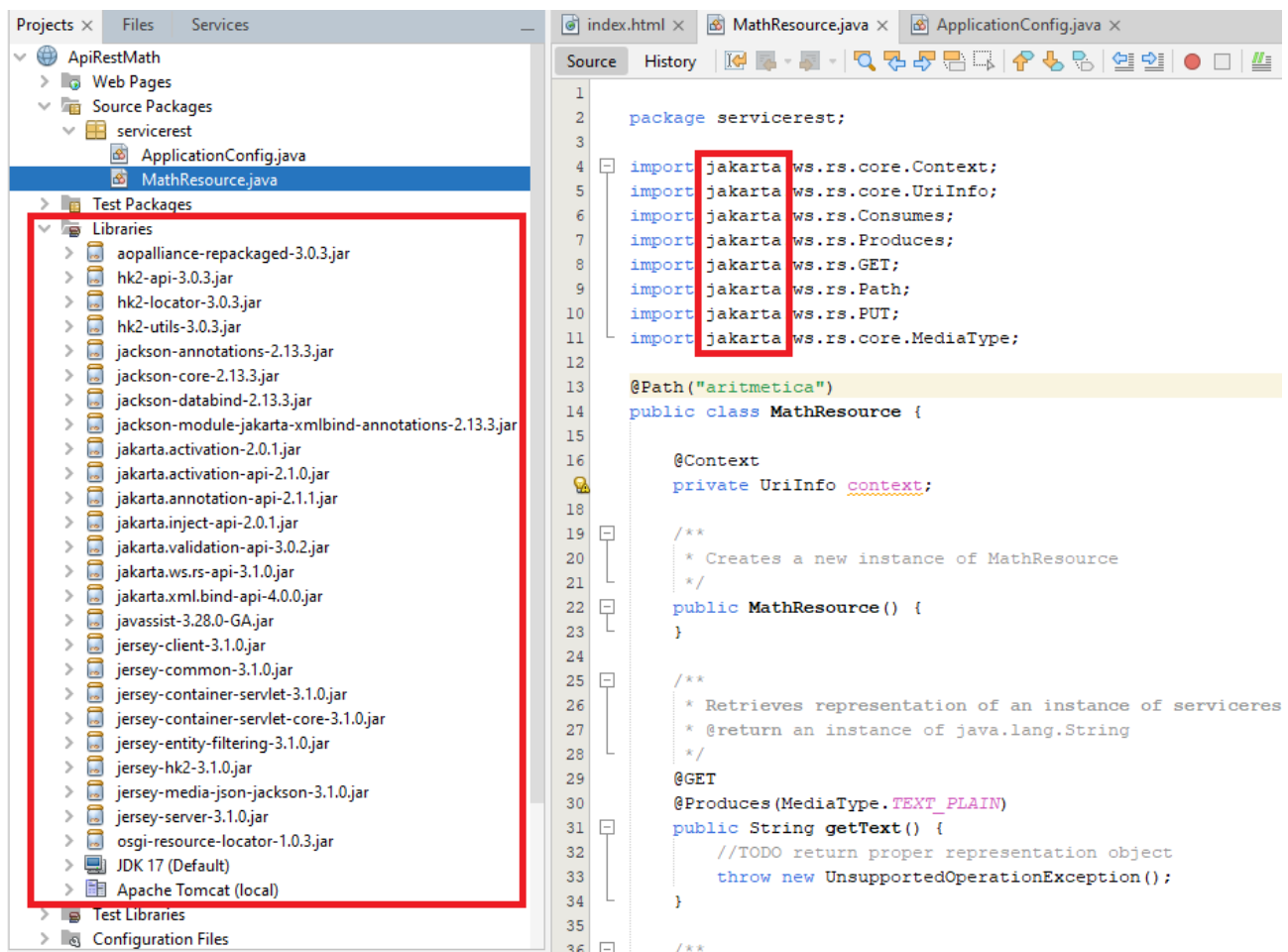
- Steps:**
  - Choose File Type
  - Select Pattern
  - Specify Resource Classes**
- Specify Resource Classes**
  - Project: ApiRestMath
  - Location: Source Packages
  - Resource Package: servicereast
  - Path: aritmetica
  - Class Name: MathResource
  - MIME Type: text/plain
  - Representation Class: java.lang.String (with a 'Select...' button)

A continuación indicaremos:

- **Resource Package:** servicere
- **Path:** **aritmetica**
- **Class Name:** MathResource
- **MIME Type:** text/plain
- **Representation Class:** java.lang.String

Al terminar de generar el código el asistente:

- eliminar de Libraries **JAX-RS-2.1.6** y **Jersey 2.34**,
- después añadir **lib** con **jakarta-ws.rs-3.1.0** y
- sustituir en el código **javax** por **jakarta** en los import y anotaciones.



### **Paso 3 – Preparar código**

Veamos el contenido de los archivos principales:

|                               |  |
|-------------------------------|--|
| <b>ApplicationConfig.java</b> | Prepara el servicio de API Rest añadiendo el <b>ApplicationPath</b> al <b>target</b> a la URL base. Por defecto es <b>webresources</b> pero puede cambiarse por <b>apiREST</b> . |
| <b>MathResource.java</b>      | Es el interfaz de métodos que se implementan y estarán disponibles para el servicio: GET   |

En **MathResource.java** eliminaremos el PUT y el GET, y añadiremos:

| <b>MathResource.java</b>  |
|---|
| <pre> ...     @GET     @Path("/sumar/{operando1}/{operando2}")     @Produces(MediaType.TEXT_PLAIN)     public int sumar(@PathParam("operando1") int op1,                     @PathParam("operando2") int op2) {         return op1+op2;     }      @GET     @Path("/dividir/{operando1}/{operando2}")     @Produces(MediaType.TEXT_PLAIN)     public double dividir(@PathParam("operando1") double op1,                         @PathParam("operando2") double op2,                         @QueryParam("decimales") int decimales) {         double resultado = op1/op2;         resultado = Math.round( resultado * Math.pow(10,decimales) ) /                         Math.pow(10,decimales);         return resultado;     } ... </pre> |

Aunque ya está terminado, podemos crear la documentación en el **index.html**.

Una vez terminado, podemos ejecutarlo. El interfaz (API) que nos ofrece es:

|     |   |                            |
|-----|---|----------------------------|
| GET | /apiREST/aritmetica/sumar/{op1}/{op2}   | Obtiene la suma de op1+op2 |
| GET | /apiREST/aritmetica/dividir/{op1}/{op2} | Obtiene la suma de op1/op2 |

Para probarlo, lo ejecutaremos y accederemos desde el navegador a:

<http://localhost:8080/math/apiREST/aritmetica/sumar/3/5>

<http://localhost:8080/math/apiREST/aritmetica/dividir/8/3?decimales=2>

## 12.3 Creación de servicios usando Bases de Datos

Para diseñar un servicio de API Rest con acceso a datos utilizaremos varias clases:

- 1) **ApplicationConfig**: clase que define el path de la aplicación y añade las clases que van a responder como servicios en el API Rest
- 2) **ServiceREST**: clases que definen los métodos con la interfaz del servicio de API Rest
- 3) **DAO**: Objeto de Acceso a Datos que define las funciones que mapean los datos almacenados en las Bases de Datos con los objetos de clases de datos del API Rest

Un **Objeto de Acceso a Datos** (en inglés, **Data Access Object**, abreviado **DAO**) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una Base de datos o un archivo. El término se aplica frecuentemente al Patrón de diseño Object.

Los Objetos de Acceso a Datos son un Patrón de los subordinados de Diseño Core J2EE y considerados una buena práctica. La ventaja de usar objetos de acceso a datos es que cualquier objeto de negocio (aquel que contiene detalles específicos de operación o aplicación) no requiere conocimiento directo del destino final de la información que manipula.

Además, también usaremos datos de respuesta que se representan por unos códigos denominados "**Response Status Code**".

### API Rest – Response.status

<https://docs.oracle.com/javaee/7/api/javax/ws/rs/core/Response.Status.html>

Si seguimos la estructura, lo primero que deberemos desarrollar es la clase DAO de nuestro nuevo API Rest **DAO**.

### Paso 1) Crear un DAO

Veamos un **DAO** que utiliza **MySQL** con una base de datos denominada **biblioteca** con una tabla **libros** que contiene 3 campos (**id**, **titulo** y **autor**).

En esta clase crearemos los siguientes métodos que permitirán a la clase **ServiceREST** acceder, obtener y actualizar datos de la tabla **libros**.

### DAOLibros.java

```
public static boolean librosExiste(int id)
public static ArrayList<Libros> librosGetAll()
public static Libros librosGet(int id)
public static boolean librosPost(Libros libro)
public static boolean librosPut(Libros libro)
```

La BD es la misma que ya hemos utilizado en la UD3:

```
CREATE SCHEMA biblioteca DEFAULT CHARACTER SET utf8mb4
                                COLLATE utf8mb4_general_ci;
USE biblioteca ;

CREATE TABLE libros (
    id          INT PRIMARY KEY AUTO_INCREMENT,
    titulo      VARCHAR(60),
    autor       VARCHAR(60),
    UNIQUE KEY titulo_UNIQUE (titulo)
);

INSERT INTO libros ( titulo, autor ) VALUES
('Macbeth', 'William Shakespeare'),
('La Celestina (Tragicomedia de Calisto y Melibea)', 'Fernando de Rojas'),
('El Lazarillo de Tormes', 'Anónimo'),
('20.000 Leguas de Viaje Submarino', 'Julio Verne'),
('Alicia en el País de las Maravillas', 'Lewis Carrol'),
('Cien Años de Soledad', 'Gabriel García Márquez'),
('La tempestad', 'William Shakespeare');
```

Para crear y probar esta clase **DAOLibros** puede crearse un proyecto de tipo **Java Application** separado y probarla.

#### AppDAOLibros (proporcionado por el profesor para las pruebas)

Crear un proyecto **Java Application** denominado **AppDAOLibros** que acceda a datos de la tabla **libros** de la BD de MySQL denominada **biblioteca** con los métodos definidos en la clase **DAOLibros**.

En esta aplicación la estructura de dependencia de clases es:

Inicio → Menu → **ControllerLibros** → **DAOLibros** → (Libros + MySQL)

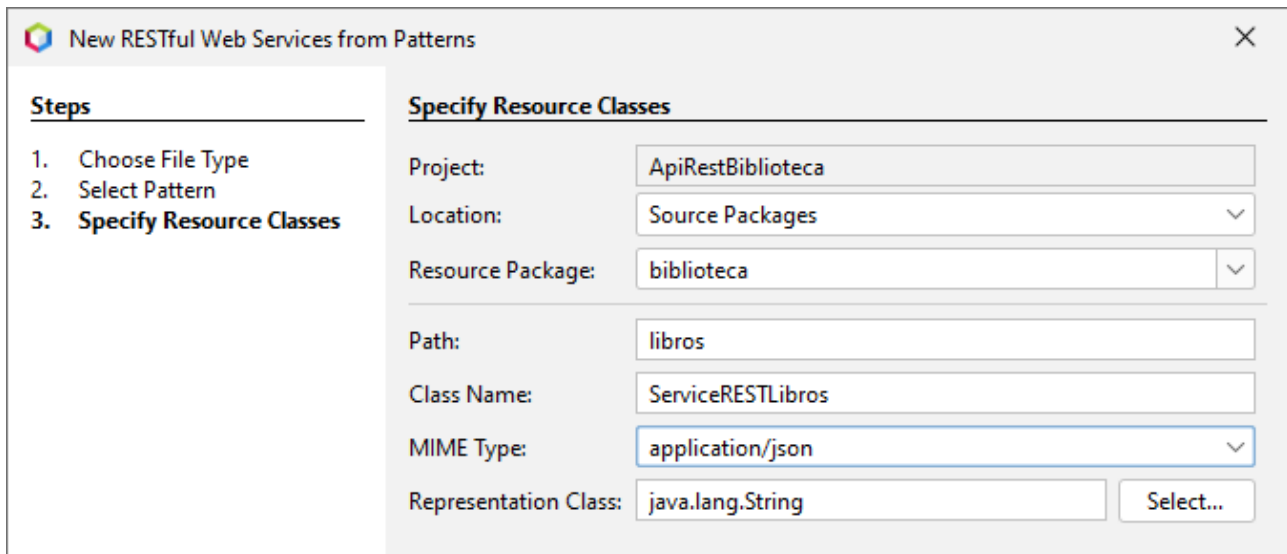
#### Paso 2) Crear ServiceREST

##### ApiRestBiblioteca

Crear un proyecto "Java Web" para crear un **API Rest** con **Apache Tomcat** denominado **ApiRestBiblioteca** que acceda a datos de las tablas **libros** de la BD de MySQL denominada **biblioteca** utilizando las clases definidas en la clase **DAOLibros**.

La URL será: <http://localhost:8080/apirest1/biblioteca/libros>

Creamos un java package **biblioteca** y generamos con el asistente "RESTful Web Services from Patterns" una clase **ServiceRESTLibros**:



### Paso 3) Configurar ApplicationConfig

Prepara el servicio de API Rest añadiendo el **ApplicationPath** al target a la URL base. Por defecto es **webresources** pero puede cambiarse por **biblioteca**.

Al terminar de generar el código el asistente:

- eliminar de Libraries **JAX-RS-2.1.6** y **Jersey 2.34**,
- después añadir **lib** con **jakarta-ws.rs-3.1.0** y
- sustituir en el código **javax** por **jakarta** en los import y anotaciones.

### Paso 4) Crear clases

Creamos un package denominado **clases** y crearemos las **clases serializables** para el DAO de **Libros** (con la estructura de la tabla libros de MySQL), **ListaLibros** (que contendrá un ArrayList de Libros).

### Paso 4) Establecer el DAO

Copiaremos nuestro **DAOLibros** en el java package **biblioteca**, y deberemos cambiar la línea **package** para evitar errores.

También copiaremos el java package **clases** con la clase **Libros**.

### Paso 5) Driver MySQL para el DAO

Copiaremos al proyecto la carpeta **lib** con el driver de MySQL para la clase **DAOLibros**.

## Paso 6) Preparar código del Service REST

Eliminaremos los métodos GET y PUT de **ServiceRESTLibros** generados por el asistente y añadiremos:

### ServiceRESTLibros.java

|      |              |                              |
|------|--------------|------------------------------|
| GET  | /libros      | Obtiene todos los libros     |
| GET  | /libros/{id} | Obtiene un libro             |
| POST | /libros      | Inserta un libro nuevo       |
| PUT  | /libros      | Actualiza un libro existente |

### ServiceRESTLibros → GET /libros

```
@GET
@Produces({MediaType.APPLICATION_JSON})
public Response getAll() {
    Response response;

    response = Response
        .status(Response.Status.OK)
        .entity(DAOLibros.librosGetAll())
        .build();

    return response;
}
```

### ServiceRESTLibros → GET /libros/{id}

```
@GET
@Path("/{id}")
@Produces({MediaType.APPLICATION_JSON})
public Response getOne(@PathParam("id") int id) {
    Response response;
    HashMap<String,String> mensaje = new HashMap<>();
    Libros libro = DAOLibros.librosGet(id);

    if (libro!=null) {
        response = Response
            .status(Response.Status.OK)
            .entity(libro)
            .build();
    } else {
        mensaje.put("mensaje", "No existe libro con ID " + id);
        response = Response
            .status(Response.Status.NOT_FOUND)
            .entity(mensaje)
            .build();
    }

    return response;
}
```

## ServiceRESTLibros → POST /libros

```

@POST
@Consumes({MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_JSON})
public Response post(Libros libro) {
    Response response;
    HashMap<String,String> mensaje = new HashMap<>();

    if (DAOLibros.librosPost(libro)) {
        mensaje.put("mensaje", "Registro insertado");
        response = Response
            .status(Response.Status.CREATED)
            .entity(mensaje)
            .build();
    } else {
        mensaje.put("mensaje", "Error al insertar");

        response = Response
            .status(Response.Status.BAD_REQUEST)
            .entity(mensaje)
            .build();
    }

    return response;
}

```

## ServiceRESTLibros → PUT /libros

```

@PUT
@Consumes({MediaType.APPLICATION_JSON})
@Produces({MediaType.APPLICATION_JSON})
public Response put(Libros libro) {
    Response response;
    HashMap<String,String> mensaje = new HashMap<>();

    if (DAOLibros.librosExiste(libro.getId())) {
        if (DAOLibros.librosPut(libro)) {
            mensaje.put("mensaje", "Registro actualizado");
            response = Response
                .status(Response.Status.OK)
                .entity(mensaje)
                .build();
        } else {
            mensaje.put("mensaje", "Error al actualizar");
            response = Response
                .status(Response.Status.CONFLICT)
                .entity(mensaje)
                .build();
        }
    } else {
        mensaje.put("mensaje", "No existe libro con ID "+libro.getId());
        response = Response
            .status(Response.Status.NOT_FOUND)
            .entity(mensaje)
            .build();
    }

    return response;
}

```



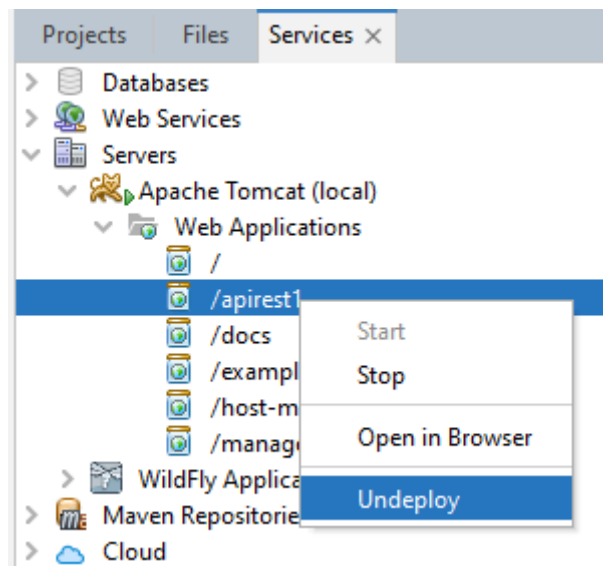
### **Paso 7) Modificar index.html**

Para mostrar información de nuestro apirest, cambiaremos el contenido de **index.html**, indicando el interfaz de nuestro API Rest.

Cuando ejecutamos la aplicación web, se realiza el "**despliegue**" en el servidor web.

En algunos casos da error porque no puede "**replegar**" la aplicación existente.

Para solucionar este problema, deberemos hacerlo manualmente accediendo a **Services** y pulsando sobre la aplicación con botón derecho para seleccionar "**Undeploy**":



## APIRest trabajando con XML

Ahora realizaremos una copia del proyecto **ApiRestBiblioteca** como **ApiRestBibliotecaXML**.

Cuando queramos devolver el resultado en **XML** en vez de **JSON**, deberemos seguir los siguientes pasos:

- 1) En **context.xml** de **Configuration Files** cambiar el context path a **/apirestxml**
- 2) Crear la clase **ListaLibros** con un **ArrayList<Libros>** y añadir a las clases de datos (**Libros**, **ListaLibros**) las anotaciones **JAXB**
- 3) Añadir el **MediaType** de XML en las anotaciones de Consumes y Produces que haya **ServiceREST**:

```
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
```

- 4) Cuando devolvemos una lista debemos diferenciar entre JSON y XML.

Este es el caso del método **getAll()**, donde comprobaremos el **header** recibido y dependiendo del valor de **Accept** devolveremos:

- **para JSON:** un **ArrayList** de **Libros**, que es lo que devuelve el método **getAll()**
- **para XML:** un objeto que contenga el **ArrayList** con las anotaciones, es decir, en nuestro caso un objeto de **ListaLibros**

### ServiceRESTLibros → GET /libros

```
@GET
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getAll(@HeaderParam("Accept") String header) {
    Response response;

    if (header.equals("application/json")) {
        response = Response
            .status(Response.Status.OK)
            .entity(DAOLibros.librosGetAll())
            .build();
    } else {
        ListaLibros lista = new ListaLibros();
        lista.setLista(DAOLibros.librosGetAll());

        response = Response
            .status(Response.Status.OK)
            .entity(lista)
            .build();
    }

    return response;
}
```

### 5) Crear clase **Mensaje** para las etiquetas del mensaje a mostrar

Crearemos la clase **Mensaje** para poder configurar las etiquetas a mostrar, ya que con **HashMap<String,String>** no podríamos definirlas.

```
Mensaje
@XmlRootElement(name="datos")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(propOrder={"mensaje"})
public class Mensaje implements Serializable {
    @XmlElement(name="mensaje") private String mensaje;

    public Mensaje() {
    }

    public Mensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    @Override
    public String toString() {
        return "Mensaje{" + "mensaje=" + mensaje + '}';
    }
}
```

### 6) Sustituir los **mensaje.put**

En todos los métodos, cambiaremos la definición del objeto mensaje de:

```
HashMap<String, String> mensaje = new HashMap<>();
```

a

```
Mensaje mensaje = new Mensaje();
```

Y todos los **mensaje.put** de:

```
mensaje.put("mensaje", "...");
```

a

```
mensaje.setMensaje("...");
```

Todos estos cambios extra en el código son el motivo de que se use más JSON que XML para el acceso a datos.

## 13. Uso de JPA para componentes de servidores web

Si trabajamos con un Servidor de Base de Datos Relacional como MySQL, y deseamos realizar un mapping entre las tablas/registro de MySQL y los objetos de Java, podremos usar **JPA**.

**Java Persistence API**, más conocida por sus siglas **JPA**, es la API de persistencia desarrollada para la plataforma Java EE. Es un framework del lenguaje de programación Java que maneja datos relacionales en aplicaciones usando la Plataforma Java en sus ediciones Standard y Enterprise

En API Rest también suele usarse JPA, igual que lo usamos en la UD de ORM.

### 13.1 API REST con JPA

Como ejemplo, realizaremos un proyecto que acceda a la BD biblioteca pero en vez de crear nuestro DAOLibros, dejaremos que JPA realice el acceso y el mapeo.

#### ApiRestBiblioJPA

Vamos a crear un proyecto "**Java Web**" denominado **ApiRestBiblioJPA** que proporcione un API Rest de Java que se instalará (deploy) en un Servidor Web **Apache Tomcat** con el asistente de NetBeans.

La aplicación tendrá una URL para mostrar el contenido de HTML y JSP:

<http://dominio/apirestjpa>

Y otra URL para activar el servicio API Rest:

<http://dominio/apirestjpa/biblioteca/libros>

## Paso 1 – Crear proyecto de tipo Java Web

The first screenshot shows the 'Name and Location' step. The project name is 'ApiRestBiblioJPA', the location is 'C:\Users\VICENTE\Documents\NetBeansProjects', and the folder is 'C:\Users\VICENTE\Documents\NetBeansProjects\ApiRestBiblioJPA'. The 'Use Dedicated Folder for Storing Libraries' checkbox is unchecked.

The second screenshot shows the 'Server and Settings' step. The 'Add to Enterprise Application' dropdown is set to '<None>'. The server is 'Apache Tomcat (local)', the Java EE version is 'Java EE 7 Web', and the context path is '/apirestjpa'. A note states: 'Note: Source Level 7 will be set for Java EE 7 project.'

The third screenshot shows the 'Frameworks' step. The frameworks to be used are 'Spring Web MVC', 'JavaServer Faces', and 'Struts 1.3.10', all of which are selected with checkboxes.

## Paso 2 – Añadir Persistence Unit

Añadiremos el Persistence Unit:

The 'New Persistence Unit' dialog shows the 'Provider and Database' step. The persistence unit name is 'ApiRestBiblioJPAPU'. The persistence library is 'EclipseLink (JPA 2.1)' and the database connection is 'MySQL biblioteca'. The table generation strategy is set to 'None'.

Si creamos la conexión "MySQL biblioteca" recuerda que completaremos la URL:  
**&autoReconnect=true&useSSL=false&serverTimezone=UTC**

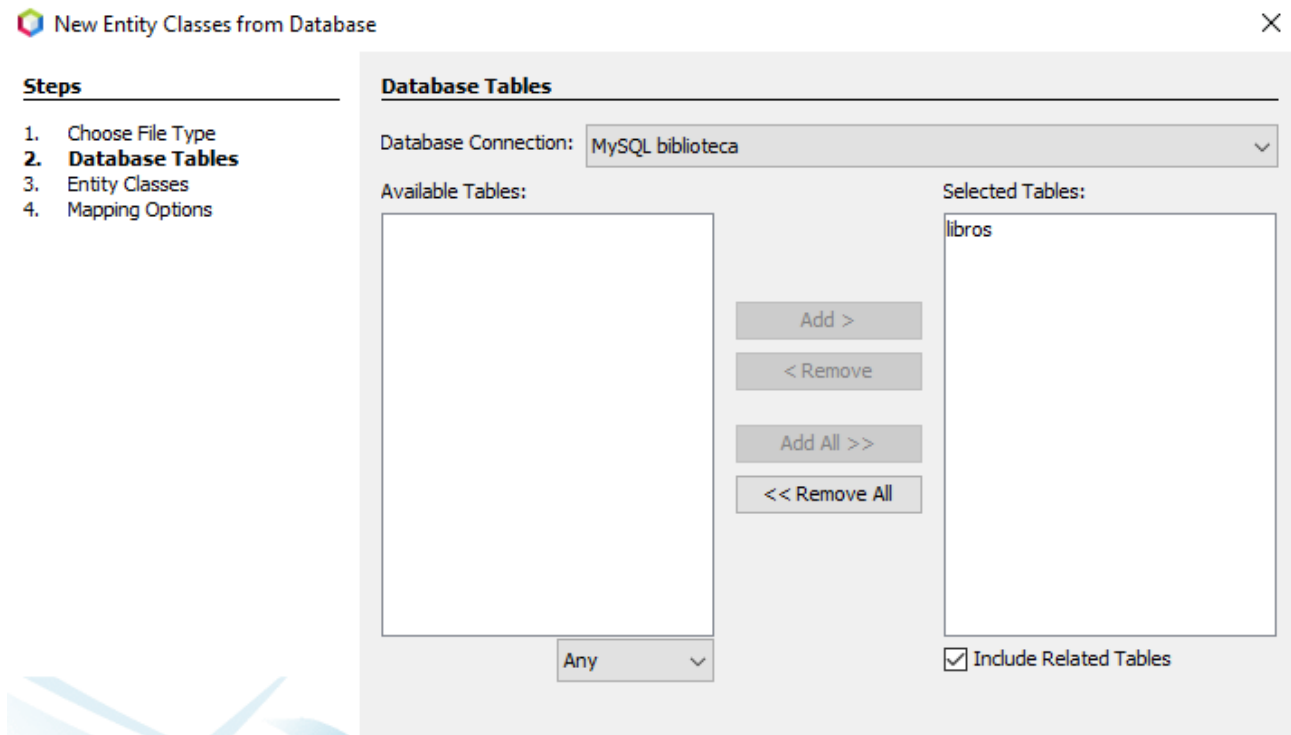
Una vez generado el archivo **persistence.xml**, le añadiremos:

```
<property name="eclipselink.logging.level" value="OFF"/>
```

### Paso 3 – Añadir Clases

Al igual que ya vimos en la UD de ORM, crearemos un package denominado **biblioteca** y añadiremos todos los archivos generados por los asistentes de JPA.

Comenzaremos con el asistente de "**Entity Classes from Database**" para crear la clase **Libros**:



**New Entity Classes from Database**

**Steps**

1. Choose File Type
2. **Database Tables**
3. Entity Classes
4. Mapping Options

**Database Tables**

Database Connection: MySQL biblioteca

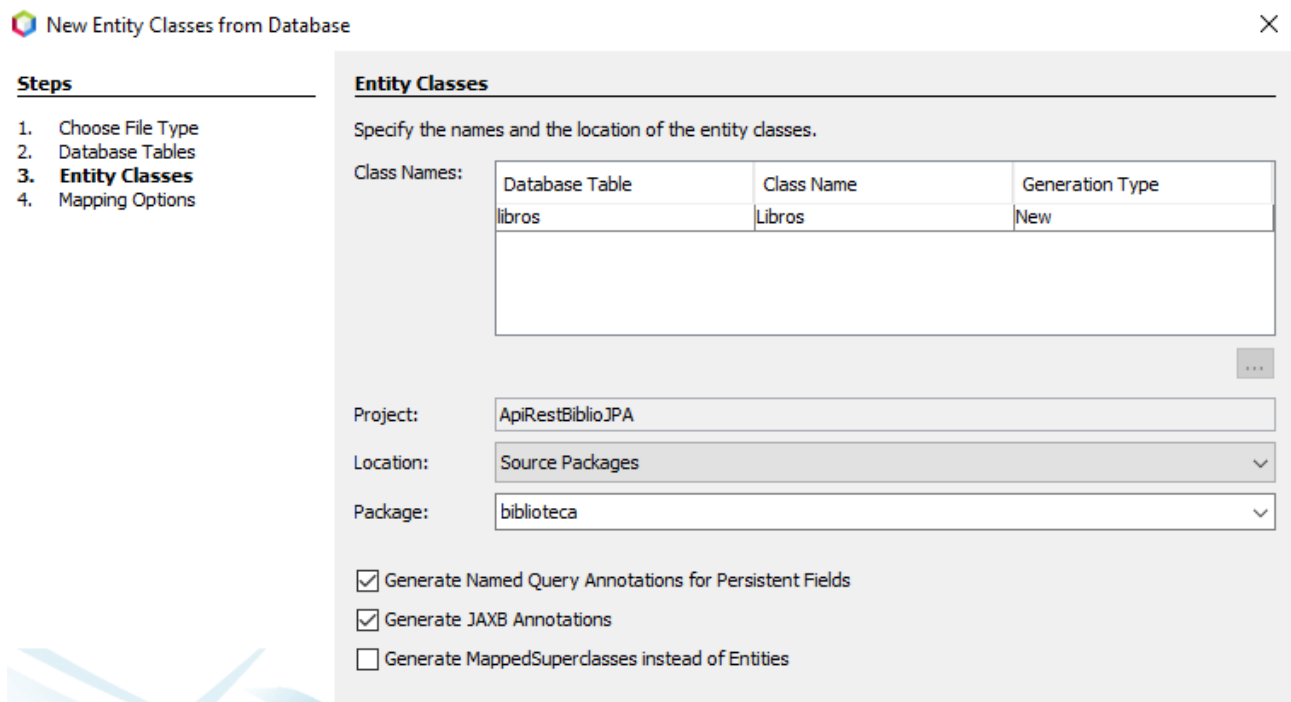
Available Tables:

Selected Tables: libros

Add > < Remove Add All >> << Remove All

Any

☒ Include Related Tables



**New Entity Classes from Database**

**Entity Classes**

Specify the names and the location of the entity classes.

Class Names:

| Database Table | Class Name | Generation Type |
|----------------|------------|-----------------|
| libros         | Libros     | New             |

Project: ApiRestBiblioJPA


Location: Source Packages

Package: biblioteca

☒ Generate Named Query Annotations for Persistent Fields

☒ Generate JAXB Annotations

☐ Generate MappedSuperclasses instead of Entities

 New Entity Classes from Database

**Steps**

1. Choose File Type
2. Database Tables
3. Entity Classes
- 4. Mapping Options**

**Mapping Options**

Specify the default mapping options.

Association Fetch:

Collection Type:

☐ Fully Qualified Database Table Names

☐ Attributes for Regenerating Tables


☒ Use Column Names in Relationships

☐ Use Defaults if Possible

☐ Generate Fields for Unresolved Relationships

### Paso 4 – Añadir Controlador

Continuaremos con el asistente "JPA Controller Classes from Entity Classes" para crear **LibrosJpaController**:

 New JPA Controller Classes from Entity Classes

**Steps**

1. Choose File Type
- 2. Entity Classes**
3. Generate JPA Controller Classes

**Entity Classes**

Available Entity Classes:

Selected Entity Classes:

biblioteca.Libros


Add >

< Remove

Add All >>

<< Remove All

☒ Include Referenced Classes

 New JPA Controller Classes from Entity Classes

**Steps**

1. Choose File Type
2. Entity Classes
- 3. Generate JPA Controller Classes**

**Generate JPA Controller Classes**

Specify the location of the JPA controller classes and related classes.

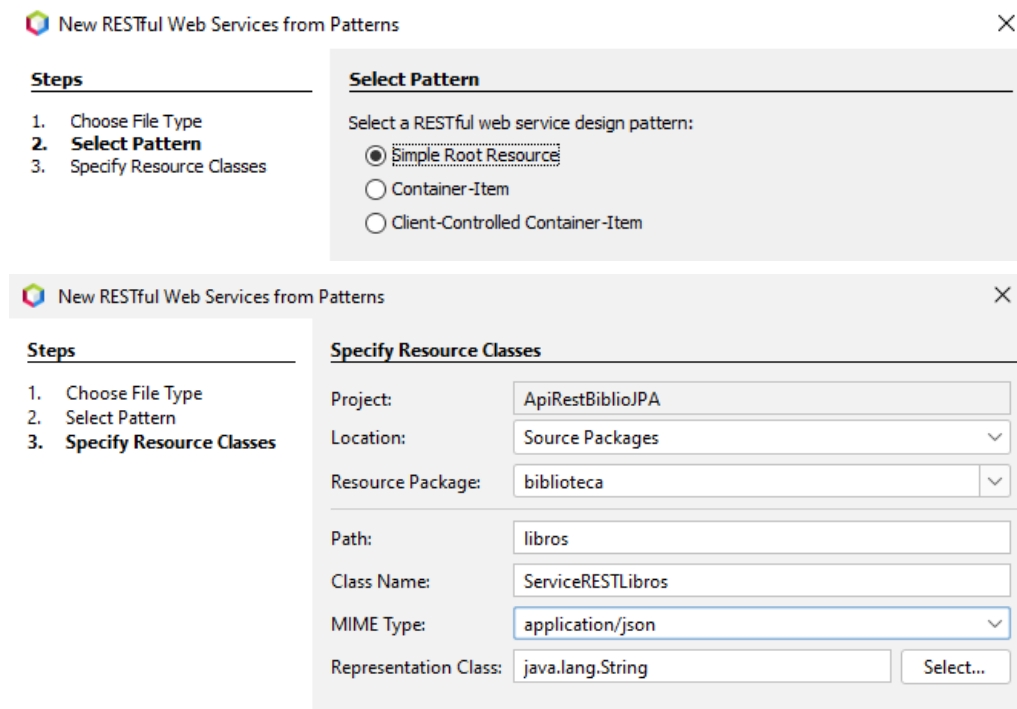
Project:

Location:

Package:

## Paso 5 – Añadir ServiceREST

Ahora continuaremos añadiendo la clase **ServiceRESTLibros** del servicio que atenderá los métodos ofrecidos por el API REST, igual que en los ejemplos anteriores. Para ello usaremos el asistente "**RESTful Web Services from Patterns**"



**New RESTful Web Services from Patterns**

**Steps**

1. Choose File Type
2. **Select Pattern**
3. Specify Resource Classes

**Select Pattern**

Select a RESTful web service design pattern:

- ☒ Simple Root Resource
- ☐ Container-Item
- ☐ Client-Controlled Container-Item

---

**New RESTful Web Services from Patterns**

**Specify Resource Classes**

Project: ApiRestBiblioJPA

Location: Source Packages

Resource Package: biblioteca

Path: libros

Class Name: ServiceRESTLibros

MIME Type: application/json

Representation Class: java.lang.String Select...

## Paso 6 – Eliminar librerías y añadir archivos JAR de lib

Eliminamos las librerías añadidas por el asistente de **JAS-RS 2.1.6**, **Jersey 2.34** y **EclipseLink (JPA 2.1)** ya que utilizaremos versiones más actualizadas.

Copiaremos en el proyecto la carpeta **lib** con el driver de **MySQL** y **JAX-RS**, y luego añadiremos las librerías en NetBeans,

Recuerda cambiar:

- **javax.ws** por **jakarta.ws** en **ApplicationConfig** y en **ServiceRESTLibros**
- **javax.xml** por **jakarta.xml** en **Libros** y otras clases de datos que creamos

## Paso 7) Configurar ApplicationConfig

Prepara el servicio de API Rest añadiendo el **ApplicationPath** al target a la URL base. Por defecto es **webresources** pero puede cambiarse por **biblioteca**.

Comprobar que en **ApplicationConfig** se ha añadido el ServiceREST creado:

### **ApplicationConfig**

```
private void addRestResourceClasses(Set<Class<?>> resources) {
    resources.add(biblioteca.ServiceRESTLibros.class);
}
```



## **Paso 8 – Preparar ServiceREST**

Ahora continuaremos añadiendo el servicio que atenderá los métodos ofrecidos por el API REST, igual que en ejemplos anteriores.

Eliminaremos los métodos de ejemplo GET y PUT y añadiremos nuestro método **getAll** y **getOne**:

### **ServiceRESTLibros.java → GET /libros**

```
...
@GET
@Produces(MediaType.APPLICATION_JSON)
public List<Libros> getAll() {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("ApiRestBiblioJPAPU");

    LibrosJpaController dao=new LibrosJpaController(emf);
    List<Libros> lista = dao.findLibrosEntities();

    emf.close();
    return lista;
}
...
```

## **Paso 9) Modificar index.html**

Para mostrar información de nuestro APIRest, cambiaremos el contenido de **index.html**, indicando el interfaz de nuestro API Rest.

Probar accediendo a:

<http://localhost:8080/apirestjpa/biblioteca/libros>

## 13.2 Añadir funcionalidad

Por ejemplo, vamos a añadir una nueva funcionalidad al servicio API Rest que nos permita **recuperar un registro concreto de libros (getOne)**. Lo realizaremos añadiendo métodos a **ServiceRESTLibros**, de la misma forma que se hizo en las aplicaciones de escritorio en la UD de ORM.

Añadiremos a **ServiceRESTLibros** el método GET indicando el path con el ID:

**ServiceRESTLibros.java** → **GET /libros/{id}**

```
...
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Libros getOne(@PathParam("id") int id) {
    EntityManagerFactory emf =
        Persistence.createEntityManagerFactory("ApiRestBiblioJPAPU");

    LibrosJpaController dao=new LibrosJpaController(emf);
    Libros libro = dao.findLibros(id);

    emf.close();
    return libro;
}
...
```

Probar accediendo a:

<http://localhost:8080/apirestjpa/biblioteca/libros/2>

## 13.3 Mejorar el código

Copiamos el proyecto **ApiRestBiblioJPA** a **ApiRestBiblioJPA2** y añadiremos:

- crear constante con el nombre del **Persistence Unit**
- **control de errores**
- devolución de objeto **response**

Para ello habrá que realizar los siguientes cambios previos:

**context.xml**

En la carpeta **Configuration Files** del proyecto, editaremos **context.xml** para cambiar el **context path**:

**context.xml**

```
...
<context-root>/apirestjpa2</context-root>
...
```

### **persistence.xml**

En la carpeta **Configuration Files** del proyecto, editaremos **persistence.xml** para cambiar el nombre de **persistence-unit**:

#### **persistence.xml**

```
<persistence-unit name="ApiRestBiblioJPA2PU" transaction-type="RESOURCE_LOCAL">
```

### **ServiceRESTLibros.java**

En la carpeta "Source Packages → biblioteca" del proyecto, editaremos **ServiceRESTLibros.java** para cambiar el nombre de **persistence-unit** en todas sus llamadas:

#### **ServiceRESTLibros (getAll y getOne)**

```
EntityManagerFactory emf =
    Persistence.createEntityManagerFactory("ApiRestBiblioJPA2PU");
```

### **index.html**

En la carpeta "Web Pages" del proyecto, editaremos **index.html** para cambiar el interfaz:

#### **index.html**

```
<td>
  <a href="/apiRESTJPA2/biblioteca/libros">/apiRESTJPA2/biblioteca/libros</a>
</td>
```

Ahora ya funcionará con los cambios realizados como el proyecto inicial.

### **Crear constante con el nombre del Persistence Unit**

#### **ServiceRESTLibros**

```
@Path("libros")
public class ServiceRESTLibros {

    @Context
    private UriInfo context;

    private static final String PERSISTENCE_UNIT="ApiRestBiblioJPA2PU";
```

Cambiar todas las llamadas en **ServiceRESTLibros.java** con la constante:

#### **ServiceRESTLibros (getAll y getOne)**

```
...
    emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
...
```

Ahora gestionaremos los errores y devolveremos un objeto tipo **Response**.

Por ejemplo, el método **getAll** podría quedar:

#### ServiceRESTLibros → GET /libros

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAll() {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    List<Libros> lista;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        LibrosJpaController dao=new LibrosJpaController(emf);
        lista=dao.findLibrosEntities();
        if (lista == null) {
            statusResul=Response.Status.NO_CONTENT;
            response = Response
                .status(statusResul)
                .build();
        } else {
            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(lista)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

De la misma forma, el método **getOne** quedaría:

#### ServiceRESTLibros → GET /libros/{id}

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getOne(@PathParam("id") int id) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;
    Libros libro;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        LibrosJpaController dao=new LibrosJpaController(emf);
        libro=dao.findLibros(id);

        if (libro == null) {
            statusResul=Response.Status.NOT_FOUND;
            mensaje.put("mensaje", "No existe libro con ID " + id);
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(libro)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

## 13.4 Añadir métodos de actualización

Ahora añadiremos los métodos **POST**, **PUT** y **DELETE** utilizando los métodos del controlador de **JPA**.

### ServiceRESTLibros → PUT /libros

```
@PUT
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response put(Libros libro) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        LibrosJpaController dao = new LibrosJpaController(emf);
        Libros libroFound = dao.findLibros(libro.getId());

        if (libroFound == null) {
            statusResul = Response.Status.NOT_FOUND;
            mensaje.put("mensaje", "No existe libro con ID " + libro.getId());
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            dao.edit(libro);
            statusResul = Response.Status.OK;
            mensaje.put("mensaje", "Libro con ID " + libro.getId()+" actualizado");
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

## ServiceRESTLibros → POST /libros

```

@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response post(Libros libro) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        LibrosJpaController dao = new LibrosJpaController(emf);

        Libros libroFound=null;
        if ( (libro.getId()!=0) && (libro.getId()!=null) ) {
            libroFound = dao.findLibros(libro.getId());
        }

        if (libroFound != null) {
            statusResul = Response.Status.FOUND;
            mensaje.put("mensaje", "Ya existe libro con ID " + libro.getId());
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            dao.create(libro);
            statusResul = Response.Status.CREATED;
            mensaje.put("mensaje", "Libro " + libro.getTitulo()+" grabado");
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}

```

**ServiceRESTLibros → DELETE /libros/{id}**

```
@DELETE
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response delete(@PathParam("id") int id) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        LibrosJpaController dao = new LibrosJpaController(emf);
        Libros libroFound = dao.findLibros(id);

        if (libroFound == null) {
            statusResul = Response.Status.NOT_FOUND;
            mensaje.put("mensaje", "No existe libro con ID " + id);
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            dao.destroy(id);
            statusResul = Response.Status.OK;
            mensaje.put("mensaje", "Libro con ID " + id + " eliminado");
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        }
    } catch (Exception ex) {
        statusResul = Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```



## 13.5 Añadir funcionalidad con XML

Realizaremos un nuevo proyecto copiando el proyecto **ApiRestBiblioJPA2** a **ApiRestBiblioJPAXML**.

Para dar soporte también a XML:

- Cambiar **context.xml**
- Cambiaremos el nombre del **Persistence Unit** en:
  - **persistence.xml**
  - **ServiceRESTLibros**
- Modificaremos todos los **Produces** y **Consumes**
- Añadiremos la clase **ListaLibros** para el método **getAll** como se hizo en ejemplos anteriores.
- Modificaremos el método **getAll** diferenciando la salida para JSON o XML según el valor de **Accept**
- Añadir la clase **Mensaje** para sustituir los HashMap que no tienen etiquetas XML
- Sustituir todos los **mensaje.put** por **mensaje.setMensaje**
- Actualizar el **index.html**

**JPA** ya genera las clases con las anotaciones de **JAXB**.

### context.xml

En la carpeta **Configuration Files** del proyecto, editaremos **context.xml**:

#### **context.xml**

```
<?xml version="1.0" encoding="UTF-8"?>
<Context path="/apirestjpaxml"/>
```

### persistence.xml

En la carpeta **Configuration Files** del proyecto, editaremos **persistence.xml** para cambiar el nombre de **persistence-unit**:

#### **persistence.xml**

```
...
<persistence-unit name="ApiRestBiblioJPA3PU" transaction-type="RESOURCE_LOCAL">
...
```

## ServiceRESTLibros.java

En la carpeta "Source Packages → biblioteca" del proyecto, editaremos **ServiceRESTLibros.java** para cambiar el nombre de la constante de Persistence Unit

### ServiceRESTLibros.java

```
...
@Path("libros")
public class ServiceRESTLibros {

    @Context
    private UriInfo context;

    private static final String PERSISTENCE_UNIT="ApiRestBiblioJPA3PU";
    ...
```

## Produces y Consume

### ServiceRESTLibros.java

```
...
    @Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    @Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
    ...
```

## ListaLibros

### ListaLibros.java

```
@XmlElement(name="libros")
@XmlAccessorType(XmlAccessType.FIELD)
public class ListaLibros implements Serializable {
    @XmlElement(name="libro") private ArrayList<Libros> lista = new ArrayList<>();
    ...
```

getAll

## ServiceRESTLibros

```

@GET
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getAll(@HeaderParam("Accept") String header) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    List<Libros> lista;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        LibrosJpaController dao=new LibrosJpaController(emf);
        lista=dao.findLibrosEntities();
        if (lista == null) {
            statusResul=Response.Status.NO_CONTENT;
            response = Response
                .status(statusResul)
                .build();
        } else {
            statusResul = Response.Status.OK;

            if (header.equals("application/json")) {
                response = Response
                    .status(statusResul)
                    .entity(lista)
                    .build();
            } else {
                ArrayList<Libros> librosArray = new ArrayList<>();
                librosArray.addAll(lista);
                ListaLibros listalib = new ListaLibros();
                listalib.setLista(librosArray);

                response = Response
                    .status(statusResul)
                    .entity(listalib)
                    .build();
            }
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}

```

## Clase Mensaje

Crearemos la clase **Mensaje** para poder configurar las etiquetas a mostrar, ya que con **HashMap<String,String>** no podríamos definirlas.

```
Mensaje
@XmlRootElement(name="datos")
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(propOrder={"mensaje"})
public class Mensaje implements Serializable {
    @XmlElement(name="mensaje") private String mensaje;

    public Mensaje() {
    }

    public Mensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    public String getMensaje() {
        return mensaje;
    }

    public void setMensaje(String mensaje) {
        this.mensaje = mensaje;
    }

    @Override
    public String toString() {
        return "Mensaje{" + "mensaje=" + mensaje + '}';
    }
}
```

## Sustituir los mensaje.put

En los métodos del **ServiceREST**, cambiaremos la definición del objeto mensaje de:

```
HashMap<String, String> mensaje = new HashMap<>();
```

a

```
Mensaje mensaje = new Mensaje();
```

Y todos los **mensaje.put** de:

```
mensaje.put("mensaje", "...");
```

a

```
mensaje.setMensaje("...");
```

## index.html

Ahora solo que da actualizar el **index.html** y probar el proyecto.

## 13.6 Proyecto Web con JPA y BD con más de una tabla

Para crear un API REST mediante un proyecto Java Web con acceso a Bases de Datos MySQL que utilice JPA pero que acceda a múltiples tablas seguiremos inicialmente los mismos pasos que en el ejemplo de una tabla.

Para simplificar, en proyectos profesionales generalmente no ofrecen la posibilidad de usar XML ya que se evita tener que añadir las anotaciones de JAXB. Por consiguiente, tomaremos como modelo el proyecto **ApiRestBiblioJPA2**.

En nuestro siguiente ejemplo, crearemos una BD denominada **bdempresa** con las siguientes instrucciones:

```
CREATE SCHEMA bdempresa DEFAULT CHARACTER SET utf8mb4 COLLATE
utf8mb4_general_ci;

USE bdempresa;

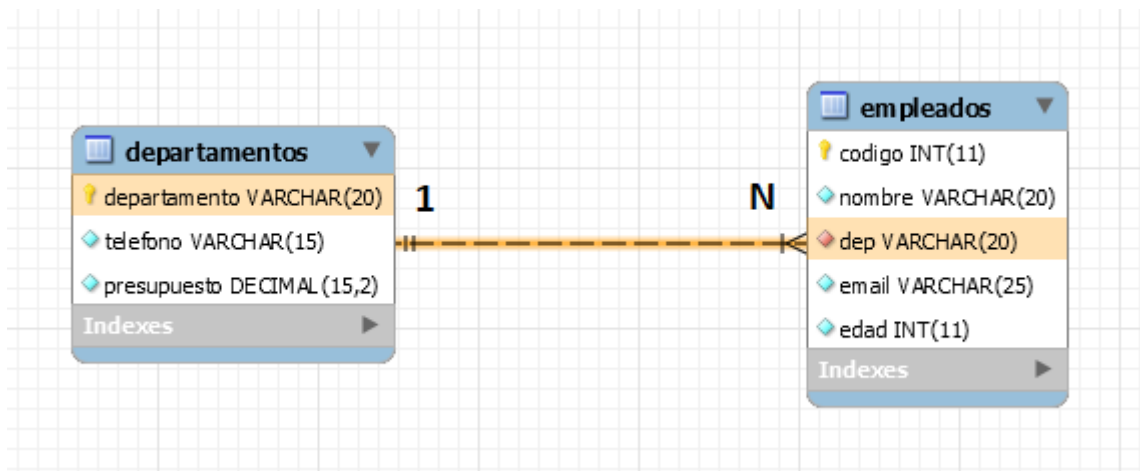
--
-- TABLA departamentos
--
CREATE TABLE departamentos (
  departamento VARCHAR(20) NOT NULL PRIMARY KEY,
  telefono VARCHAR(15) NOT NULL,
  presupuesto DECIMAL(15,2) NOT NULL
);

INSERT INTO departamentos (departamento, telefono, presupuesto) VALUES
('Compras', '965452315', 1980500),
('Facturación', '965452300', 560000),
('Ventas', '965452314', 2450950);

--
-- TABLA empleados
--
CREATE TABLE empleados (
  codigo INT(11) NOT NULL PRIMARY KEY AUTO_INCREMENT,
  nombre VARCHAR(20) NOT NULL,
  dep VARCHAR(20) NOT NULL,
  email VARCHAR(25) NOT NULL,
  edad INT(11) NOT NULL,
  FOREIGN KEY (dep) REFERENCES departamentos(departamento)
);

INSERT INTO empleados (codigo, nombre, dep, email, edad) VALUES
(1, 'Claresta', 'Ventas', 'cwiles0@ning.com', 38),
(2, 'Malinde', 'Ventas', 'mabrahamowicz1@paypal.or', 30),
(3, 'Lauri', 'Compras', 'lharwood2@bluehost.com', 47),
(4, 'Richie', 'Facturación', 'roxlee3@chronoengine.org', 48),
(5, 'Monti', 'Compras', 'mcraigheid4@bandcamp.com', 31),
(6, 'Rosabel', 'Ventas', 'rverrills5@java.com', 26),
(7, 'Caresa', 'Ventas', 'cterrans6@bravesites.com', 53),
(8, 'Brendis', 'Facturación', 'bburchfield7@cnbc.com', 37),
(9, 'Almeta', 'Compras', 'alow8@dyndns.org', 52),
(10, 'Berkeley', 'Ventas', 'bdacey9@wiley.com', 33);
```

Se puede observar que mediante el **FOREIGN KEY** se crea una relación entre **departamentos** y **empleados de 1 a N**, guardando el código del departamento en la tabla de empleados.



Se pueden comprobar los datos realizando un SELECT con JOIN entre las dos tablas

```
SELECT * FROM empleados, departamentos
WHERE empleados.dep=departamentos.departamento;
```

Crearemos un proyecto ejemplo que puede servir para el PROYECTO DE CICLO

### ApiRestEmpreJPA

Vamos a crear un proyecto "**Java Web**" denominado **ApiRestEmpreJPA** que proporcione un API Rest de Java que se instalará (deploy) en un Servidor Web de **Apache Tomcat** con el asistente de NetBeans.

La aplicación tendrá una URL para mostrar el contenido HTML y JSP:

<http://dominio/bdempresa>

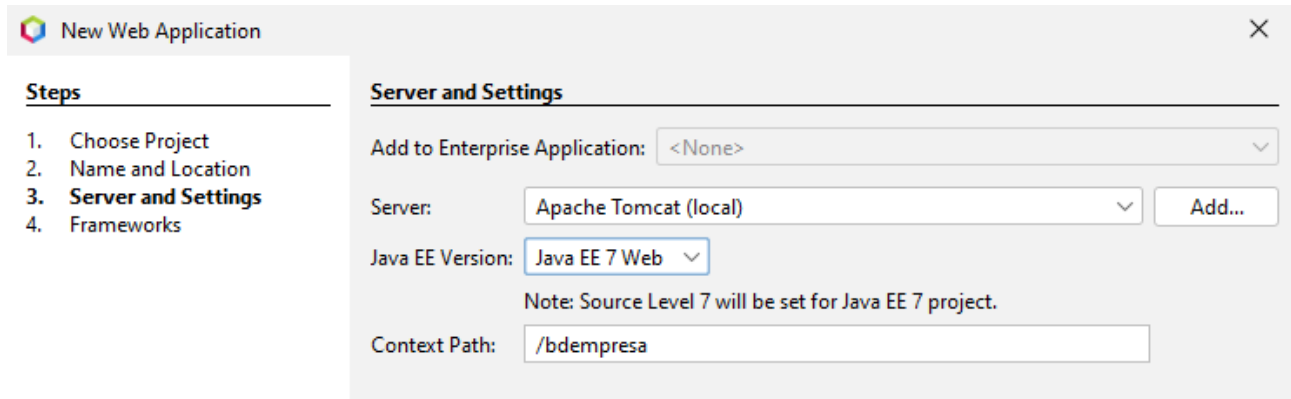
Y otras URL para activar el servicio API Rest:

<http://dominio/bdempresa/datos/empleados>  
<http://dominio/bdempresa/datos/departamentos>

Realizaremos los siguientes pasos:

### **Paso 1 – Crear proyecto de tipo Java Web**

Indicaremos **bdempresa** como Context Path

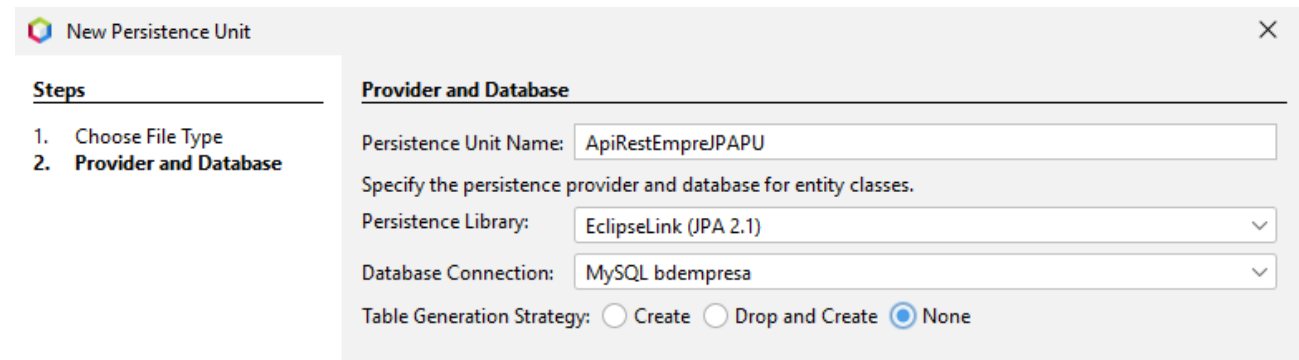


The screenshot shows the 'New Web Application' dialog box. On the left, under 'Steps', step 3 'Server and Settings' is selected. The main area, titled 'Server and Settings', contains the following fields: 'Add to Enterprise Application' set to '<None>', 'Server' set to 'Apache Tomcat (local)', 'Java EE Version' set to 'Java EE 7 Web', and 'Context Path' set to '/bdempresa'. A note states: 'Note: Source Level 7 will be set for Java EE 7 project.'

Crear el archivo **index.html** que mostrará la información del API REST

### **Paso 2 – JPA**

- Añadir Persistence Unit



The screenshot shows the 'New Persistence Unit' dialog box. On the left, under 'Steps', step 2 'Provider and Database' is selected. The main area, titled 'Provider and Database', contains the following fields: 'Persistence Unit Name' set to 'ApiRestEmpreJPAPU', 'Persistence Library' set to 'EclipseLink (JPA 2.1)', and 'Database Connection' set to 'MySQL bdempresa'. The 'Table Generation Strategy' section has three radio buttons: 'Create', 'Drop and Create', and 'None', with 'None' selected.

- Crear Java Package **jpaempresa**

- Crear clases (Entity Classes from Database)
- Crear controladores (JPA Controller Classes from Entity Classes)

### **Paso 3 – Service REST**

- Crear Java Package **serempresa**
- Crear un **ServiceREST** para cada tabla (RESTful Web Services from Patterns)
- Configurar **ApplicationConfig** (**datos**)

Comprobar el contenido de todas las clases creadas.

**Si aparecen vacías eliminarlas y volverlas a crear.**

Comprobar que en **ApplicationConfig** se han añadido los **ServiceREST** creados:

#### **ApplicationConfig**

```
private void addRestResourceClasses(Set<Class<?>> resources) {
    resources.add(biblioteca.ServiceRESTLibros.class);
    resources.add(biblioteca.ServiceRESTDepartamentos.class);
}
```



### **Paso 4 – Eliminar librerías y añadir archivos JAR de lib**

Eliminamos las librerías añadidas por el asistente de **JAS-RS 2.1.6, Jersey 2.34 y EclipseLink (JPA 2.1)** ya que utilizaremos versiones más actualizadas.

Copiaremos en el proyecto la carpeta **lib** con el driver de **MySQL** y **JAX-RS**, y luego añadiremos las librerías en NetBeans,

Recuerda cambiar:

- **javax.ws** por **jakarta.ws** en **ApplicationConfig** y en **ServiceRESTLibros**
- **javax.xml** por **jakarta.xml** en **Libros** y otras clases de datos que creamos

### **Paso 5 - Limitar la recursividad en relaciones**

Para evitar errores de recursividad debemos indicar qué campo no mostrar al serializar nuestras clases a JSON.

#### **Opción 1) Mostrar relación muchos**

Mostrar los empleados de cada departamento

NO mostrar el departamento de un empleado

#### **Opción 2) Mostrar relación uno**

Mostrar el departamento de un empleado

NO Mostrar los empleados de cada departamento

Para evitar mostrar un campo utilizaremos la anotación **@JsonbTransient**

En la **Opción 1** añadiríamos **@JsonbTransient** en la clase **Empleados**

```
@JoinColumn(name = "dep", referencedColumnName = "departamento")
@ManyToOne(optional = false)
@JsonbTransient
private Departamentos dep;
```

En algunas ocasiones preferiríamos al menos ver el valor del campo relacionado. De forma opcional podemos añadirlo con información de solo lectura.

En nuestro caso, si optamos por la opción 1. Además, mostraremos el campo **dep** con el nombre de **departamento** de la clase **Empleados** indicando como **read-only**, es decir, que ni se inserte ni se actualice y añadiendo solo el método **get**:

```
@Column(name = "dep", updatable = false, insertable = false)
private String departamento;

public String getDepartamento() {
    return departamento;
}
```

Otra opción (dentro de la opción 1) sería utilizar el nuevo campo para realizar **update** e **insert** si el campo es **NOT NULL** en la Base de Datos:

```
@JoinColumn(name = "dep", referencedColumnName = "departamento",
            updatable = false, insertable = false)
@ManyToOne(optional = false)
@JsonbTransient
private Departamentos dep;

@Column(name = "dep", updatable = true, insertable = true)
private String departamento;

public String getDepartamento() {
    return departamento;
}

public void setDepartamento(String departamento) {
    this.departamento = departamento;
}
```

En la **Opción 2** añadiríamos **@JsonbTransient** en la clase Departamentos

```
@OneToMany(cascade = CascadeType.ALL, mappedBy = "dep")
@JsonbTransient
private Collection<Empleados> empleadosCollection;
```

## Paso 6 - Añadir método GET de ejemplo en cada ServiceREST

Lo primero es añadir en cada ServiceREST el valor de la constante PERSISTENCE\_UNIT:

### ServiceRESTEmpleados y ServiceRESTDepartamentos

```
...

@Context
private UriInfo context;

private static final String PERSISTENCE_UNIT="ApiRestEmpreJPAPU";

...
```

Ahora los métodos de prueba:

### ServiceRESTEmpleados → GET /empleados

```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Response getAll() {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    List<Empleados> lista;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);

        EmpleadosJpaController dao=new EmpleadosJpaController(emf);
        lista=dao.findEmpleadosEntities();
        if (lista == null) {
            statusResul=Response.Status.NO_CONTENT;
            response = Response
                .status(statusResul)
                .build();
        } else {
            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(lista)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

## ServiceRESTDepartamentos → GET /departamentos/{id}

```

@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response getOne(@PathParam("id") String id) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    Departamentos dep;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        DepartamentosJpaController dao=new DepartamentosJpaController(emf);
        dep=dao.findDepartamentos(id);

        if ( dep == null) {
            statusResul=Response.Status.NOT_FOUND;
            mensaje.put("mensaje", "No existe departamento con ID " + id);
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(dep)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}

```

## ServiceRESTEmpleados → POST /empleados

```

@POST
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response post(Empleados emp) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

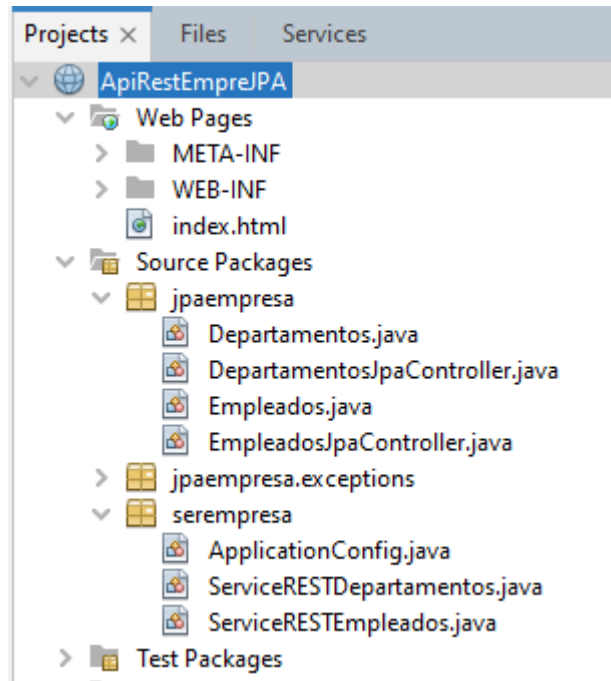
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        EmpleadosJpaController dao = new EmpleadosJpaController(emf);

        Empleados empFound=null;
        if ( (emp.getCodigo()!=0) && (emp.getCodigo()!=null) ) {
            empFound = dao.findEmpleados(emp.getCodigo());
        }

        if (empFound != null) {
            statusResul = Response.Status.FOUND;
            mensaje.put("mensaje", "Ya existe empleado con ID "+emp.getCodigo());
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        } else {
            dao.create(emp);
            statusResul = Response.Status.CREATED;
            mensaje.put("mensaje", "Empleado " + emp.getNombre()+" grabado");
            response = Response
                .status(statusResul)
                .entity(mensaje)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}

```

La estructura del proyecto quedaría:



Algunas URL de prueba serían:

<http://localhost:8080/bdempresa/datos/empleados>

<http://localhost:8080/bdempresa/datos/departamentos/ventas>

Y con **Postman** podríamos probar el POST.

Ahora podríamos seguir añadiendo los diferentes métodos que deseemos ofrecer en el API REST.

## 13.7 Proyecto Web con JPA y BD usando JPQL

Para crear una respuesta API REST como resultado de un JPQL deberemos crear un método que ejecute la consulta y, si no es una objeto de nuestras clases de datos, cree el resultado en un **JsonObject** o **JsonArray** según queramos obtener un objeto o un array de objetos.

Como ejemplo realizaremos un método para obtener la edad más alta de entre todos los empleados de cada departamento. La consulta **JQPL** sería:

```
SELECT emp.departamento, MAX(emp.edad)
FROM Empleados emp
GROUP BY emp.departamento
```

Para que esta consulta funcione, debemos haber añadido el campo **departamento** en la clase **Empleados** como se ha explicado anteriormente, ya que si se ha puesto **@JsonTransient** en **dep**, se pierde este campo. El código que debe estar es:

```
@Column(name = "dep", updatable = false, insertable = false)
private String departamento;

public String getDepartamento() {
    return departamento;
}
```

Para ello crearemos un nuevo API con la url:

|     |                                    |
|-----|------------------------------------|
| GET | /bdempresa/datos/empleados/mayores |
|-----|------------------------------------|

El código del método sería:

### ServiceRESTEmpleados → GET /empleados/mayores

```
@GET
@Path("/mayores")
@Produces(MediaType.APPLICATION_JSON)
public Response getMayores() {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    String resultado="{ }";
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        EmpleadosJpaController dao=new EmpleadosJpaController(emf);
        EntityManager em = dao.getEntityManager();

        Query query =
            em.createQuery("SELECT emp.departamento, MAX(emp.edad) "+
                " FROM Empleados emp GROUP BY emp.departamento");
        List<Object[]> lista = query.getResultList();

        if ( (lista!=null) && (!lista.isEmpty())) {
            JsonArrayBuilder jsonArrayB = Json.createArrayBuilder();
            for (Object[] obj : lista) {
                JsonObjectBuilder jsonOB = Json.createObjectBuilder();
                jsonOB.add("departamento", (String) obj[0] );
                jsonOB.add("maxedad", (Integer) obj[1] );
                jsonArrayB.add(jsonOB);
            }
            JsonArray arrayJson = jsonArrayB.build();

            resultado = arrayJson.toString();

            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(resultado)
                .build();
        } else {
            statusResul = Response.Status.NO_CONTENT;
            response = Response
                .status(statusResul)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

<http://localhost:8080/bdempresa/datos/empleados/mayores>



El código de creación de `JsonArray` usado en el método anterior a partir de la **lista** es:

```
JsonArrayBuilder jsonArrayB = Json.createArrayBuilder();
for (Object[] obj : lista) {
    JsonObjectBuilder jsonOB = Json.createObjectBuilder();
    jsonOB.add("departamento", (String) obj[0] );
    jsonOB.add("maxedad", (Integer) obj[1] );
    jsonArrayB.add(jsonOB);
}
JsonArray arrayJson = jsonArrayB.build();
resultado = arrayJson.toString();
```

Si optamos por generar el JSON sin los nombres de campos (**departamento**, **maxedad**), es decir, tal cual lo recibimos, se puede sustituir por el procesador de JSON-B que usa JAX-RS y hacer:

```
resultado = new JsonBindingBuilder().build().toJson(lista);
```

O también por el Mapeador de Objetos de Jackson y hacer:

```
resultado = new ObjectMapper(new JsonFactory()).writeValueAsString(lista);
```

## 13.8 Proyecto Web con JPA y BD usando NamedQuery

Para crear una respuesta API REST como resultado de usar un **NamedQuery** con una consulta JPQL deberemos crear un método que ejecute la llamada y devuelva el resultado.

En nuestro ejemplo, la consulta **NamedQuery** que almacenaremos en la clase `Empleados` sería:

```
@NamedQuery(name = "Empleados.findByDepartamento",
            query = "SELECT e FROM Empleados e WHERE e.departamento = :dep")
```

Para que esta consulta funcione, debemos haber añadido el campo **departamento** en la clase **Empleados** como se ha explicado anteriormente, ya que si se ha puesto **@JsonTransient** en **dep**, se pierde este campo. El código que debe estar es:

```
@Column(name = "dep", updatable = false, insertable = false)
private String departamento;

public String getDepartamento() {
    return departamento;
}
```

Para ello crearemos un nuevo API con la url:

|     |  |
|-----|--|
| GET | /bdempresa/datos/empleados/departamento/{codigo} |
|-----|--|

El código del método sería:

**ServiceRESTEmpleados → GET /empleados/departamento/{codigo}**

```
@GET
@Path("/departamento/{dep}")
@Produces(MediaType.APPLICATION_JSON)
public Response getEmpleadosDep(@PathParam("dep") String dep) {
    EntityManagerFactory emf = null;
    HashMap<String, String> mensaje = new HashMap<>();
    Response response;
    Status statusResul;

    List<Empleados> lista = null;
    try {
        emf = Persistence.createEntityManagerFactory(PERSISTENCE_UNIT);
        EmpleadosJpaController dao=new EmpleadosJpaController(emf);
        EntityManager em = dao.getEntityManager();

        TypedQuery<Empleados> consultaRegistros =
            em.createNamedQuery("Empleados.findByDepartamento", Empleados.class);
        lista = consultaRegistros.setParameter("dep", dep).getResultList();

        if ( (lista!=null) && (!lista.isEmpty())) {
            statusResul = Response.Status.OK;
            response = Response
                .status(statusResul)
                .entity(lista)
                .build();
        } else {
            statusResul = Response.Status.NO_CONTENT;
            response = Response
                .status(statusResul)
                .build();
        }
    } catch (Exception ex) {
        statusResul=Response.Status.BAD_REQUEST;
        mensaje.put("mensaje", "Error al procesar la petición");
        response = Response
            .status(statusResul)
            .entity(mensaje)
            .build();
    } finally {
        if (emf != null) {
            emf.close();
        }
    }
    return response;
}
```

<http://localhost:8080/bdempresa/datos/empleados/departamento/ventas>

## 13.9 Proyecto Web con JPA y BD usando fechas

En los campos de tipo Date, aunque tenemos la anotación **@Temporal** que debería formatear los datos quedando solo la fecha y eliminando la hora, al mapear nos devuelve una fecha en formato **ISO\_INSTANT**:

```
'2011-12-03T10:15:30Z'
```

Pueden consultarse las siguientes páginas:

- <https://javabydeveloper.com/temporal/>
- <https://docs.oracle.com/javase/8/docs/api/java/time/format/DateTimeFormatter.html>

Cuando intentamos mapear de la BD un campo de tipo **DATE** (no DATETIME O TIMESTAMP), el asistente nos crea un atributo en la clase con un conjunto de anotaciones. Por ejemplo para un campo **fecha DATE** en MySQL con valores que tienen el formato **'YYYY-MM-DD'** obtendríamos:

```
@Basic(optional = false)
@Column(name = "fecha")
@Temporal(TemporalType.DATE)
private Date fecha;
```

Para que funcione correctamente debemos cambiar en la clase el tipo del atributo de Java a **LocalDate** y eliminar la anotación de **@Temporal**

```
@Basic(optional = false)
@Column(name = "fecha")
private LocalDate fecha;
```

Esta modificación nos permitirá trabajar con el formato **'AAAA-MM-DD'** ya que **LocalDate** ya trabaja con formato **ISO\_LOCAL\_DATE**. De esta forma, la siguiente consulta **JSQL** funcionará correctamente:

```
SELECT alu FROM Alumnos alu WHERE alu.fechaNac = '2011-12-03'
```

Esto es así porque **@Temporal** trabaja correctamente para **java.sql.Date** pero no para **java.util.Date**, y **JPA** no trabaja correctamente con **java.sql.Date**.

Si hiciéramos un método para que reciba en la URL la fecha en String, sería:

```
@GET
@Path("/fechanac/{fecha}")
@Produces(MediaType.APPLICATION_JSON)
public Response getAlumnosFechaNac(@PathParam("fecha") String fechanac) {
    ...
}
```

### Trabajar con campo Date pero en formato DateTime

También podemos trabajar con **GET**, **POST**, **PUT** y **DELETE** sin cambiar nada utilizando el formato **ISO\_INSTANT**, es decir, que los valores serán del formato:

```
'2011-12-03T00:00:00Z'
```

El único problema es debemos sincronizar los horarios entre Java y MySQL. Para ello cambiaremos el constructor del **ServiceREST** (que está vacío) indicando la zona horaria:

```
public ServiceRESTAlumnos() {
    TimeZone.setDefault(TimeZone.getTimeZone("Etc/UTC"));
}
```

### Formato 'YYYY-MM-DD' manteniendo el tipo Date pero solo para lectura

Si no vamos a actualizar el atributo, es decir, que el atributo será de solo lectura, podemos usar la anotación **@JsonDateFormat**.

En una consulta **JPQL** que use fechas, la mejor forma de realizar la consulta es trabajar con un formato de tipo **String**. Por ejemplo, recordamos que si tenemos un objeto fecha de tipo **Date**, podemos convertirlo a **String** con:

```
Date fecha = new Date();
String strFecha = new SimpleDateFormat("YYYY-MM-DD").format(fecha)
```

Supongamos una clase **Alumnos** con un atributo para almacenar la fecha de nacimiento de tipo **Date** denominado **fechaNac**.

Podríamos preparar una consulta **JPQL** con el **String strFecha** en formato **'YYYY-MM-DD'**.

```
SELECT alu FROM Alumnos alu
WHERE function('date_format', alu.fechaNac, '%Y-%m-%d') = :strfecha
```

O incluso almacenarla en una **NamedQuery**.

```
@NamedQuery(
    name = "Empleados.findByFechaNac",
    query = "SELECT alu FROM Alumnos alu
            WHERE function('date_format', alu.fechaNac, '%Y-%m-%d') = :strfecha")
```