



Servicios REST

Sesión 2:

Anotaciones básicas JAX-RS. El modelo de despliegue



Índice

- **Más sobre la anotación @Path**
- Anotaciones @Produces y @Consumes
- Inyección de parámetros JAX-RS
- Configuración y despliegue de aplicaciones JAX-RS



Expresiones @Path: más de una variable

- Para que una clase Java sea identificada por JAX-RS como un recurso, tiene que estar anotada con @PATH. A estas clases se las denomina **recursos JAX-RS raíz**
- El valor de @PATH denota una URI, que puede tener **segmentos variables**

```
@Path("/{nombre1}/{nombre2}/")  
public class MiResource {  
    ...  
}
```

```
http://expertojava.org/REST/Pedro/Lopez
```

```
@Path("/")  
public class ClienteResource {  
    @GET  
    @Path("clientes/{apellido1}-{apellido2}")  
    public String getCliente(@PathParam("apellido1") String ape1,  
                             @PathParam("apellido2") String ape2) {  
        ...  
    }  
}
```

```
GET http://org.expertojava/contexto/clientes/Pedro-Lopez
```



Expresiones @Path: uso de expresiones regulares

- **Formato:** {" nombre-variable [":" expresion-regular] " }

```
@Path("/clientes")
public class ClienteResource {
    @GET
    @Path("{id : \\d+}")
    public String getCustomer(@PathParam("id") int id) {
        ...
    }
}
```

- `getCustomer` procesaría una petición dirigida a `/clientes/101`, pero no a `/clientes/Pepe`
- Ejemplos de expresiones regulares: (ver <https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>)
 - `\\d+` : solo puede contener uno o más dígitos
 - `.+` : se permite cualquier carácter
 - `[a-zA-Z][a-zA-Z]` : el primer carácter puede ser una letra del alfabeto en mayúsculas o minúsculas, y el segundo también (sólo puede haber dos caracteres)
 - `[^0-9][0-9]+` : el primer carácter NO puede ser un dígito, el segundo carácter y los siguientes tienen que ser dígitos
 - `*` denota cero o más veces, `?` una o ninguna vez, `+` una o más veces, `{n}` exactamente n veces



Expresiones @Path: ambigüedad

- ¿Qué ocurre si una URI encaja con más de un @Path?

```
@Path("/clientes")
public class ClienteResource {
    @GET
    @Path("{id : .+}")
    public String getCliente(@PathParam("id") String id) {...}

    @GET
    @Path("{id : .+}/direccion")
    public String getDireccion(@PathParam("id") String id) {...}
}
```

- **Reglas de prioridad:** (la idea es que el @Path más específico debe tener mayor prioridad)
 - Los @Path con mayor número de caracteres literales tienen mayor prioridad
 - A igualdad del anterior, menor número de variables
 - A igualdad del anterior, menor número de variables que tienen expresiones regulares asociadas



Expresiones @Path: matrix parameters

- Pares nombre-valor que aparecen al **final** de un **segmento** de la URI (segmento de ruta) y están delimitados por el carácter ;

```
http://ejemplo.coches.com/seat/ibiza;color=black/2006
```

- Los parámetros matrix representan atributos de ciertos segmentos de la URI y se utilizan para propósitos de identificación. Pensemos en ellos como **adjetivos**.
- Se ignoran para el *matching* de la URI. Se puede acceder a ellos con `@MatrixParam`

```
@Path("/seat")
public class SeatService {

    @GET
    @Path("/ibiza/{anyo}")
    @Produces("image/jpeg")
    public Jpeg getIbizaImagen(@PathParam("anyo"), @MatrixParam("color") String color) {...}
}
```



Subresource locators

- Son métodos Java anotados con `@Path`, pero sin anotaciones `@GET`, `@PUT`, ... Estos métodos no devuelven directamente la respuesta, sino un servicio JAX-RS (un subrecurso) que es el que procesa la petición y devuelve la respuesta.

Petición `GET /clientes/europa-db/130 HTTP/1.0`

Recurso Raiz

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

    @Path("{zona}-db")
    public IClienteResource getBaseDeDatos(@PathParam("zona") String db) {
        // devuelve una instancia de un recurso dependiendo del valor del parámetro db
        IClienteResource resource = localizaClienteResource(db);
        return resource;
    }

    protected ClienteEuropaResource localizaClienteResource(String db) {
        ...
    }
}
```

La petición se "delega" en el subrecurso *ClienteEuropaResource*



Subrecursos

- No atienden las peticiones HTTP directamente, por ello no necesitan `@Path` para responder a la petición (sí pueden llevarlo si necesitamos “complementar” la trayectoria)

Mapeado a `ClienteEuropaResource` por el subresource locator

Petición

`GET /clientes/europa-db /130 HTTP/1.0`

```
public class ClienteEuropaResource implements IClienteResource{
    ...

    @POST
    @Consumes("application/xml")
    public Response crearCliente(InputStream is) { ... }

    @GET
    @Path("{id}")
    @Produces("application/xml")
    public StreamingOutput recuperarClienteId(@PathParam("id") int id) { ... }

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

Subrecurso



Otro de ejemplo de subrecurso

```
@Path("/alumnos")
public class AlumnosResource {

    @Context
    UriInfo uriInfo;

    @GET
    @Produces({MediaType.APPLICATION_XML,
              MediaType.APPLICATION_JSON})
    public List<AlumnoBean> getAlumnos() {
        return FactoriaDaos.getAlumnoDao().getAlumnos();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public void addAlumno(AlumnoBean alumno) {
        String dni = FactoriaDaos.getAlumnoDao()
            .addAlumno(alumno);
        URI uri = uriInfo.getAbsolutePathBuilder()
            .path("{dni}").build(dni);
        Response.created(uri).build();
    }

    @Path("{alumno}")
    public AlumnoResource getAlumno(
        @PathParam("alumno") String dni) {
        return new AlumnoResource(uriInfo, dni);
    }
}
```

Recurso raíz

```
public class AlumnoResource {
    UriInfo uriInfo;
    String dni;

    public AlumnoResource(UriInfo uriInfo, String dni) {
        this.uriInfo = uriInfo;
        this.dni = dni;
    }

    @GET
    @Produces({MediaType.APPLICATION_XML,
              MediaType.APPLICATION_JSON})
    public AlumnoBean getAlumno() {
        ...
        return alumno;
    }

    @PUT
    @Consumes(MediaType.APPLICATION_XML)
    public Response setAlumno(AlumnoBean alumno) {
        ...
    }

    @DELETE
    public void deleteAlumno() {
        FactoriaDaos.getAlumnoDao().deleteAlumno(dni);
    }
}
```

Subrecurso



Índice

- Más sobre la anotación @Path
- **Anotaciones @Produces y @Consumes**
- Inyección de parámetros JAX-RS
- Configuración y despliegue de aplicaciones JAX-RS



@Consumes

- Se puede asociar a `@POST` o `@PUT`
- Si `@Consumes` se aplica a la **clase**, se aplica por defecto a todos sus métodos. Si se aplica a nivel de **método**, se ignora la anotación `@Consumes` a nivel de clase para ese método
- Podemos tener `@Consumes` distintos en varios métodos de la misma clase
- El cliente debe enviar la cabecera **Content-Type** con el valor del tipo de dato enviado. La petición será dirigida al método con dicho valor en la anotación `@Consumes`
- Si no hay métodos que puedan "consumir" el tipo especificado por el cliente se enviará automáticamente un *status 415 Unsupported Media Type*

```
@Path("/pedidos")
public class PedidoResource {

    @PUT
    @Consumes("application/xml")
    public void modificarPedidoXML(InputStream pedido) {...}

    @PUT
    @Consumes("application/json")
    public void modificarPedidoJson(InputStream pedido) {...}

}
```

```
PUT /pedidos
content-type: application/xml

<pedido ... />
```

```
PUT /pedidos
content-type: application/json

{"pedido": ... }
```



@Produces

- Habitualmente asociada a @GET
- El cliente especifica el/los tipos de datos que puede aceptar con la cabecera **Accept**. Si el recurso no tiene un @Produces con un formato aceptable para el cliente, se genera automáticamente un *status* HTTP 406 Not Acceptable

```
@Path("/pedidos")
public class PedidoResource {

    @GET
    @Produces("application/xml")
    public String getPedidoXml() { }

    @GET
    @Produces("application/json")
    public String getPedidoJson() { }
}
```

GET /pedidos
Accept: application/xml

GET /pedidos
Accept: application/json



Índice

- Más sobre la anotación @Path
- Anotaciones @Produces y @Consumes
- **Inyección de parámetros JAX-RS**
- Configuración y despliegue de aplicaciones JAX-RS



Acceso a los datos de la petición

- Ya hemos visto cómo acceder a las partes variables de la URI a través de `@PathParam`
- También hay anotaciones para acceder a
 - Los parámetros de consulta HTTP (`@QueryParam`)
 - Si la petición ha sido consecuencia de un envío de formulario HTML, acceder a los datos enviados (`@FormParam`)
 - Las cabeceras HTTP (`@HeaderParam`)
 - Formularios html (`@FormParam`)
 - El contexto de los servlets (`@Context`)
 - La URI completa, para poder dividirla “manualmente” en partes (`@Context UriInfo`)
 - Inyección de Beans con anotaciones `@xxxParam` (`@BeanParam`)



Acceso a los parámetros de consulta HTTP: @QueryParam

- Lo que en un *servlet* haríamos con `request.getParameter()`
- Se convierten automáticamente los tipos primitivos

```
GET /clientes?inicio=0&total=10
```

```
@Path("/clientes")
public class ClienteResource {
    @GET
    @Produces("application/xml")
    public String getClientes(@QueryParam("inicio") int inicio,
                              @QueryParam("total") int total)
    ... }
}
```

- Los parámetros de consulta **siempre** aparecen al **final** de la URI, y **siempre** pertenecen al recurso **completo** que estemos referenciando (a diferencia de los parámetros Matrix)



Acceso a datos de formulario: @FormParam

- Para datos enviados a través de un formulario, que por defecto usan el Content-Type `application/x-www-form-urlencoded`

```
<form action="http://ejemplo.com/clientes" method="post">
  Nombre: <input type="text" name="nombre"><br>
  Apellido: <input type="text" name="apellido"><br>
  <input type="submit" value="Send">
</form>
```

```
@Path("/clientes")
public class ClienteResource {
    @POST
    public void crearCliente(@FormParam("nombre") String nom,
                             @FormParam("apellido") String ape) {
        ... }
}
```



Acceso a cabeceras HTTP: @HeaderParam

- Como valor de la anotación ponemos el nombre de la cabecera de la petición a la que queremos acceder

```
@Path("/miservicio")
public class MiServicio {
    @GET
    @Produces("text/html")
    public String get(@HeaderParam("Referer") String referer) {
        ... }
}
```





Acceso al contexto de la petición: @Context

- Nos sirve para acceder a diversos elementos del contexto de la petición o del propio JAX-RS que no tienen "anotación propia": `ServletContext`, `request`, la URI completa (`UriInfo`), ...

- Acceso al contexto de servlets

```
@GET
@Produces("image/jpeg")
public InputStream getImagen(@Context ServletContext sc) {
    return sc.getResourceAsStream("/fotos/" + nif + ".jpg");
}
```

- Acceso a todas las cabeceras HTTP de la petición

```
@Path("/miservicio")
public class MiServicio {
    @GET
    @Produces("text/html")
    public String get(@Context HttpHeaders cabeceras) {
        String referer = headers.getRequestHeader("Referer").get(0);
        for (String header : headers.getRequestHeaders().keySet()) {
            System.out.println("Se ha utilizado esta cabecera : " + header);
        }
    }
}
```



Acceso a la URI de la petición (UriInfo)

- La interfaz `javax.ws.rs.core.UriInfo` nos proporciona un API para consultar y extraer información sobre las peticiones URI de entrada:

```
public interface UriInfo {  
    public java.net.URI getAbsolutePath();  
    public UriBuilder getAbsolutePathBuilder();  
  
    public java.net.URI getBaseUri();  
    public UriBuilder getBaseUriBuilder();  
  
    public String getPath();  
    public List<PathSegment> getPathSegments();  
    public MultivaluedMap<String, String> getPathParameters();  
    ...  
}
```

Ruta absoluta de la petición

Ruta base de la petición

Ruta relativa de nuestros servicios REST

- Ejemplo: la petición sobre `http://localhost:8080/contexto/rest/clientes/2`
 - `uri.getAbsolutePath()` = `http://localhost:8080/contexto/rest/clientes/2`
 - `uri.getBaseUri()` = `http://localhost:8080/contexto/rest/`
 - `uri.getPath()` = `/clientes/2`
 - `uri.getPathSegments()` = `{ clientes, 2 }`

`/contexto` es donde desplegamos el `.war`

`/rest` es el valor de la anotación `@Path` de la clase del recurso



Empaquetar los datos en un @BeanParam

- Podemos “empaquetar” en un solo objeto diversos datos que nos interese obtener con inyección JAX-RS
 - Creamos una clase y le ponemos las anotaciones ya vistas

```
public class ClienteInput {  
    @FormParam("nombre")  
    String nombre;  
  
    @HeaderParam("Content-Type")  
    String contentType;  
  
    public String getFirstName() {...}  
    ...  
}
```

- En el método que procesa la petición anotamos con @BeanParam un parámetro de esa clase

```
@Path("/clientes")  
public class ClienteResource {  
    @POST  
    public void crearCliente(@BeanParam ClienteInput nuevoCliente) {  
        ...  
    }  
}
```



Conversión automática de tipos en los datos inyectados

- Los datos se obtienen de la petición HTTP como un String, pero se convierten automáticamente en los siguientes casos
 - Tipos primitivos (int, short, float, double, byte, char, y boolean)
 - Clases con un constructor con un único parámetro de tipo String
 - Clases con un método estático denominado `valueOf()`, que toma un único String como argumento, y devuelve una instancia de la clase.
 - Clases de tipo `java.util.List<T>`, `java.util.Set<T>`, o `java.util.SortedSet<T>`, en donde T es un tipo que satisface los criterios 2 ó 3, o es un String. Por ejemplo, `List<Double>`, `Set<String>`, o `SortedSet<Integer>`.
- Si se falla al intentar hacer una conversión:
 - Si es un `@MatrixParam`, `@QueryParam`, o `@PathParam`, se genera automáticamente un 404 Not Found
 - En otro caso, se genera un 400 Bad Request



Valores por defecto

- Algunos de los datos inyectados pueden ser opcionales y por tanto no estar presentes en la petición actual. Podemos darles un valor por defecto con `@DefaultValue`

```
@Path("/clientes")
public class ClienteResource {
    @GET
    @Produces("application/xml")
    public String getClientes(
        @DefaultValue("0") @QueryParam("inicio") int inicio,
        @DefaultValue("10") @QueryParam("total") int total)
        ... }
}
```



Índice

- Más sobre la anotación @Path
- Anotaciones @Produces y @Consumes
- Inyección de parámetros JAX-RS
- **Configuración y despliegue de aplicaciones JAX-RS**



Algo más sobre la configuración de JAX-RS

- Los **recursos JAX-RS** pueden seguir el modelo
 - **Per request**: con cada petición se crea una nueva instancia del recurso. Por tanto no hay estado (es la opción por defecto)
 - **Singleton**: hay una única instancia del recurso que sirve todas las peticiones. En ella podemos guardar estado
- Además en una aplicación JAX-RS podemos tener también **proveedores (providers)**, usados para funcionalidades como
 - Serialización/deserialización de objetos
 - Mapeado de excepciones a códigos de estado HTTP



Configuración mediante la clase Application

- Creamos una clase que herede de Application y que devuelva un conjunto (Set) de
 - Proveedores y recursos per-request: método getClasses()
 - Proveedores y recursos singleton: método getSingletons()

```
package org.expertojava;
import javax.ws.rs.core.Application;
import javax.ws.rs.ApplicationPath;

@ApplicationPath("/rest")
public class ComercioApplication extends Application {

    public Set<Class<?>> getClasses() {
        HashSet<Class<?>> set = new HashSet<Class<?>>();
        set.add(ClienteResource.class);
        set.add(PedidoResource.class);
        return set;
    }

    public Set<Object> getSingletons() {
        JsonWriter json = new JsonWriter();
        TarjetaCreditoResource servicio = new TarjetaCreditoResource();
        HashSet<Object> set = new HashSet();
        set.add(json);
        set.add(servicio);
        return set;
    }
}
```



Usar la implementación “por defecto” de Application

- Automáticamente escanea las clases en busca de recursos y proveedores

```
package org.expertojava;

import javax.ws.rs.ApplicationPath;
import javax.ws.rs.core.Application;

//Únicamente configuramos la raíz del servicio
@Path("/rest")
public class ComercioApplication extends Application {
    //y usamos la implementación “por defecto”
}
```



Configurar Application a través del web.xml

- Equivalente a lo anterior, pero la "raíz" del servicio se configura en el web.xml
- Es la forma que vimos en la sesión anterior

```
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet-mapping>
    <servlet-name>javax.ws.rs.core.Application</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```



Configuración en un servidor que no implemente JAX-RS

- Los servidores con una versión <3.0 del API de servlets no tienen por qué implementar JAX-RS
- Habría que añadir la implementación de JAX-RS a nuestro proyecto, y en el web.xml:

```
<?xml version="1.0"?>
<!-- Usamos RESTEasy, la implementación de JBoss, valdría cualquier otra, p.ej. Jersey -->
<web-app>
  <servlet>
    <servlet-name>JAXRS</servlet-name>
    <servlet-class>
      org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher
    </servlet-class>
    <init-param>
      <param-name> javax.ws.rs.Application</param-name>
      <param-value>org.expertoJava.ComercioApplication</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>JAXRS</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```



¿Alguna duda, pregunta...?

