Servicios REST

Sesión 3:
Manejadores de contenidos. Respuestas del servidor y Manejo de Excepciones

Índice

Tipos de datos del cuerpo de la petición

- Recordemos que en las peticiones `POST` y `PUT` se suelen definir los datos en el cuerpo de la petición
- La cabecera `content-type` en la petición HTTP define el tipo de dato que se envía en el cuerpo de la petición
- Por ejemplo, para añadir un libro nuevo, haciendo un `POST` al recurso `/library`:

```
POST /library
content-type: text/plain

EJB 3.0; Bill Burke
```

```
POST /library
content-type: text/xml

<book name="EJB 3.0" author="Bill Burke"/>
```

Tipos de datos del cuerpo de la respuesta

- Recordemos que en las peticiones `GET` se reciben los datos solicitados en el cuerpo de la respuesta
- La cabecera `Accept` en la petición HTTP define el tipo de dato solicitado
- La cabecera `Content-Type` en la respuesta HTTP define el tipo de dato que se envía en el cuerpo de la respuesta
- Por ejemplo, para consultar un libro, haciendo un `GET` al recurso `/library`:

Petición

```
GET /library/111111111
Accept: text/xml
```

Respuesta

```
HTTP/1.1 200 OK
Content-Type: text/xml

<book name="EJB 3.0" author="Bill Burke"/>
```

Mapeado de los datos del cuerpo de la petición y respuesta

- El parámetro del método que procesa la petición (objeto Java) se **mapea** con el dato enviado en el cuerpo de la petición (formato texto)
- Cuando el servicio REST devuelve los datos en el cuerpo de la respuesta, se **mapea** el objeto java con el dato enviado en el cuerpo de la respuesta (formato texto)
- Las clases que se encargan de **mapear** las representaciones del cuerpo del mensaje a entidades (objetos java) y viceversa, se denominan **proveedores de entidades**
- Dichas clases se anotan con **@provider**
- JAX-RX contempla un conjunto de proveedores de entidades estándar
- Podemos definir nuestros propios proveedores de entidades implementando las interfaces `MessageBodyReader` y `MessageBodyWriter`

Secuencia de acciones JAX-RS para mapear los datos

mapeado cuerpo petición a entidad

- Se obtiene el *media type* de la petición (valor de la cabecera HTTP Content-Type).
- Si la petición no contiene una cabecera Content-Type se usará "application/octet-stream"
- Se identifica el tipo java del parámetro cuyo valor será mapeado desde el cuerpo del mensaje
- Se localiza la clase **MessageBodyReader** que soporta el *media type* de la petición y se usa su método ***readFrom()*** para mapear el contenido del cuerpo del mensaje HTTP en el tipo Java que corresponda
- Si no es posible encontrar el MessageBodyReader adecuado se genera la excepción **NotSupportedException** con el código **405**

mapeado entidad a cuerpo respuesta

- Se obtiene el objeto que será mapeado a la entidad del cuerpo del mensaje
- Se determina el *media type* de la respuesta
- Se localiza la clase **MessageBodyWriter** que soporta el objeto que será mapeado a la entidad del cuerpo del mensaje HTTP, y se utiliza su método ***writeTo()*** para realizar dicho mapeado
- Si no es posible encontrar el MessageBodyWriter adecuado se genera la excepción **InternalServerErrorException** (que es una subclase de WebApplicationException) con el código **500**

Proveedores de entidades estándar en JAX-RS

Tipo Java	Media Type
byte[]	*/* (Cualquier *media type*)
java.lang.String	*/* (Cualquier *media type*)
java.io.InputStream	*/* (Cualquier *media type*)
java.io.Reader	*/* (Cualquier *media type*)
java.io.File	*/* (Cualquier *media type*)
javax.activation.DataSource	*/* (Cualquier *media type*)
javax.xml.transform.Source	text/xml, application/xml, application/*+xml (tipos basados en xml)
javax.xml.bind.JAXBElement and application-supplied JAXB classes	text/xml, application/xml, application/*+xml (tipos basados en xml)
MultivaluedMap<String,String>	application/x-www-form-urlencoded (Contenido de formularios)
StreamingOutput	*/* (Cualquier *media type*) (Sólo `MessageBodyWriter`)
java.lang.Boolean, java.lang.Character, java.lang.Number	text/plain

Ejemplo con InputStream

El ejemplo es sólo una ilustración. En el caso concreto presentado no sería necesario usar un InputStream. Al ser datos con formato de texto sería mejor usar `String`

```
POST /pedidos
content-type: text/plain

Antonio;Muñoz Molina;Nueva York
```

```java
// Leemos los datos del cliente del body del mensaje HTTP
// La cabecera Content-Type tiene que ser text/plain
// El formato es: Nombre, Apellidos y Ciudad separados por ;
// Ejemplo: Antonio;Muñoz Molina;Nueva York
@POST
@Consumes("text/plain")
public Response crearCliente(InputStream is) {
   try {
     Cliente cliente = leerCliente(is);
     cliente.setId(idContador.addAndGet(1));
     clienteDB.put(cliente.getId(), cliente);
     System.out.println("Cliente creado "
                        + cliente.getId());
     return Response.created(
             URI.create("/clientes/"
                 + cliente.getId())).build();
    } catch (IOException e) {
        e.printStackTrace();
        throw new WebApplicationException(
                    Response.Status.BAD_REQUEST);
   }
}
```

```java
// Leemos del InputStrem
// El formato es: Nombre, Apellidos y Ciudad separados por ;
// Ejemplo: Antonio;Muñoz Molina;Nueva York
private Cliente leerCliente(InputStream stream)
                                        throws IOException {
  ByteArrayOutputStream baos = new ByteArrayOutputStream();
  byte[] buffer = new byte[1000];
  int wasRead = 0;
  do {
     wasRead = stream.read(buffer);
     if (wasRead > 0) {
        baos.write(buffer, 0, wasRead);
     }
  } while (wasRead > -1);
  String datos[] = new String(baos.toByteArray()).split(";");
  Cliente cliente = new Cliente();
  cliente.setNombre(datos[0]);
  cliente.setApellidos(datos[1]);
  cliente.setCiudad(datos[2]);
  return cliente;
}
```

Formularios de entrada

- Los datos de los formularios HTML son codificados en el cuerpo de la petición con el media-type `application/x-www-form-urlencoded`
- Se mapean en un parámetro de tipo MultivaluedMap<String, String>

```
@Path("/")
public class MiServicio {
     @POST
     @Consumes("application/x-www-form-urlencoded")
     public Response procesarPedido(
                         MultivaluedMap<String, String> form) {
        ...
      }
}
```

Índice

- Manejadores de contenidos
- **Representaciones XML y JSON**
- Respuestas
- Excepciones

JAXB

- Java for XML Binding (*JSR 222*)
- Especificación antigua, orientada al mapeo de objetos Java con representaciones XML
- Muy útil en JAX-RS, donde se adapta también al mapeo con representaciones JSON

propiedades de la clase (se asocian con pares getters/setters): id, nombre

```
@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    @XmlAttribute
    private int id;

    @XmlElement
    private String nombreCompleto;

    public Customer() {}

    public int getId() { return this.id; }
    public void setId(int id) { this.id = id; }

    public String getNombre() {
            return this.nombre;
    }
    public void setNombre(String nombre} {
            this.nombre = nombre;
    }
}
```

Campos (privados) de la clase

Al proceso de **serializar** (convertir) un objeto Java en un documento XML se denomina *marshalling*
El proceso inverso se denomina *unmarshalling*

```
<cliente id="42">
    <nombre>Pablo Martinez</nombre>
</cliente>
```

Atributo XML

Elemento XML

Conversión a XML

- Al anotar la clase como `@XmlAccessorType(XmlAccessType.FIELD)` se convierten a XML todos los campos del objeto (los que tengan un valor distinto de null, ya sean públicos o privados), independientemente de que se hayan anotado o no
- Con la anotación `@XmlAccessorType(XmlAccessType.NONE)` sólo se convierten a XML los campos anotados con JAXB
- Al anotar la clase como `@XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)` se convierten a XML todos los campos públicos del objeto y las propiedades, a menos que se hayan anotado con `@XmlTransient`. Es el XmlAccessType por defecto, si no se especifica ninguno
- Al anotar la clase como `@XmlAccessorType(XmlAccessType.PROPERTY)` se convierten a XML todas las propiedades del objeto, a menos que se hayan anotado con `@XmlTransient`
- Si queremos evitar convertir algún atributo (campo o propiedad) debemos marcarlo como `@XmlTransient`

Campos no primitivos

- En el caso de clases con campos de tipos anotados con `@XmlRootElement` se genera un elemento XML anidado con el principal

```
@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    @XmlAttribute
    protected int id;

    @XmlElement
    protected String nombreCompleto;

    @XmlElement
    protected Direccion direccion;

    public Cliente() {}

    // getters y setters
    ...
}
```

```
@XmlRootElement(name="direccion")
@XmlAccessorType(XmlAccessType.FIELD)
public class Direccion {
  @XmlElement
  protected String calle;

  @XmlElement
  protected String cludad;

  @XmlElement
  protected String codPostal;

  public Direccioon() {}

  // getters y setters
  ...
}
```

```
<cliente id="56">
    <nombre>Ricardo Lopez</nombre>
    <direccion>
       <calle>calle del oso, 35</calle>
       <ciudad>Alicante</ciudad>
       <codPostal>01010</codPostal>
    </direccion>
 </cliente>
```

Para serializar la clase, es necesario que dicha clase tenga un *constructor sin parámetros*

Producción y consumo

- Es muy sencillo generar y consumir clases en formato XML usando las anotaciones JAXB y el tipo de medio `application/xml` o `text/xml`
- Para convertir un objeto XML en el cuerpo de una petición `POST` hay que anotar la petición con `content-type: application/xml` o `text/xml` y hay que declarar el mismo tipo en la anotación `@Consumes` del método `POST`

```
POST http://localhost:8080/ejemplobase-rest/rest/clientes/
content-type: application/xml
<cliente>
   <nombre>Antonio</nombre>
   <apellidos>Muñoz Molina</apellidos>
</cliente>
```

```
@POST
@Consumes("application/xml")
public Response crearCliente(Cliente cliente) {
   ...
}
```

Producción y consumo

- Para producir un objeto XML basta con anotar el método `GET` con `@Produces("application/xml")` y devolver un objeto del tipo Java
- La petición `GET` debe tener la cabecera `Accept: application/xml` o `text/xml`

```
@GET
@Path("{id}")
@Produces("application/xml")
public Cliente recuperarClienteIdXML(@PathParam("id") int id) {
   Cliente cliente = clienteDB.get(id);
   if (cliente == null) {
        throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   return cliente;
}
```

Mapeado de colecciones (a una serie de elementos)

- Es posible convertir colecciones en XML

```
@XmlRootElement(name="estado")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
              new ArrayList<TareaBean>();
}
```

```
@XmlRootElement(name="tarea")
public class TareaBean {

   ...
}
```

Cada objeto de la lista se serializa como un elemento XML

```
<estado valor="Idle" toner="25">
    <tarea>
        <nombre>texto.doc</nombre>
        <estado>imprimiendo...</estado>
        <paginas>13</paginas>
    </tarea>
    <tarea>
        <nombre>texto2.doc</nombre>
        <estado>en espera...</estado>
        <paginas>5</paginas>
    </tarea>
</estado>
```

Mapeado de colecciones (a un elemento que contiene otros)

- Podemos incluir un elemento que "envuelva" la colección de elementos serializados

```java
@XmlRootElement(name="estado")
public class EstadoBean {

   @XmlAttribute(name="valor")
   public String estado = "Idle";

   @XmlAttribute(name="toner")
   public int tonerRestante = 25;

   @XmlElementWrapper(name="tareas")
   @XmlElement(name="tarea")
   public List<TareaBean> tareas =
              new ArrayList<TareaBean>();
}
```

```xml
<estado valor="Idle" toner="25">
  <tareas>
    <tarea>
        <nombre>texto.doc</nombre>
        <estado>imprimiendo...</estado>
        <paginas>13</paginas>
    </tarea>
    <tarea>
        <nombre>texto2.doc</nombre>
        <estado>en espera...</estado>
        <paginas>5</paginas>
    </tarea>
  </tareas>
</estado>
```

```java
@XmlRootElement(name="tarea")
public class TareaBean {

   ...
}
```

Cada objeto de la lista se serializa como un elemento XML, anidado en una etiqueta que denota la colección y agrupa a todos ellos

Mapeado en JSON

- Las clases anotadas con JAXB se pueden mapear automáticamente con representaciones JSON
- Basta con definir el tipo `application/json` o `text/json` en las anotaciones `@Consumes` y/o `@Produces`
- Es posible anotar un método con `text/json` y `text/xml` y dejar que la petición elija el tipo de formato requerido. Se devolverá la representación solicitada en la cabecera `Accept` o `Content-type`

```
@GET
@Path("{id}")
@Produces({"application/xml","application/json"})
public Cliente recuperarClienteIdXML(@PathParam("id") int id) {
   final Cliente cliente = clienteDB.get(id);
   if (cliente == null) {
       throw new WebApplicationException(Response.Status.NOT_FOUND);
   }
   return cliente;
}
```

Representación XML y JSON

xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<producto>
   <id>1</id>
   <nombre>iPad</nombre>
   <descripcion>Dispositivo móvil</descripcion>
   <precio>500</precio>
</producto>
```

json

```json
{
  "id":"1",
  "nombre":"iPad",
  "descripcion":"Dispositivo móvil",
  "precio":500
}
```

xml

```xml
<cliente id="56">
    <nombre>Ricardo Lopez</nombre>
    <direccion>
       <calle>calle del oso, 35</calle>
       <ciudad>Alicante</ciudad>
       <codPostal>01010</codPostal>
    </direccion>
 </cliente>
```

json

```json
{
  "id": "56",
  "nombre": "Ricardo Lopez",
  "direccion": { "calle": "calle del oso, 35",
                 "ciudad": "Alicante",
                 "codPostal": "01010"
               }
}
```

Índice

- Manejadores de contenidos
- Representaciones XML y JSON
- **Respuestas**
- Excepciones

Respuestas del servidor: códigos de respuesta por defecto

- Los métodos `GET`, `POST`, `PUT` y `DELETE` devuelven por defecto un código de respuesta de éxito
  - `200 OK` si el cuerpo contiene algún contenido
  - `201 Created` si se ha creado con éxito un nuevo recurso
  - `204 No Content` si el cuerpo no contiene nada

Petición

```
GET /productos HTTP/1.1
```

```
HTTP/1.1 200 OK
Content-Type: application/xml

<productos>
   <producto ... />
   ...
</productos>
```

Respuesta

Petición

```
POST /pedidos HTTP/1.1
Content-Type: application/xml

<pedido>
   <total>199.02</total>
     ...
</pedido>
```

```
HTTP/1.1 201 Created
Content-Type: application/xml
Location: http://pedidos/233

<pedido id="233">
   <total>199.02</total>
     ...
</pedido>
```

Respuesta

Petición

```
PUT /productos/233 HTTP/1.1
Content-Type: application/xml

<producto id="111">
   <nombre>iPhone</nombre>
   <precio>649.99</precio>
</producto>
```

```
HTTP/1.1 200 OK
<producto id="111">
   <nombre>iPhone</nombre>
   <precio>649.99</precio>
</producto>
```

Respuesta

Petición

```
DELETE /pedidos/233/
```

```
HTTP/1.1 204 No
content
```

Respuesta

Códigos de respuesta de fallo

- Códigos HTTP entre `400` y `599`
  - Los códigos `4xx` son errores del cliente
  - Los códigos `5xx` son errores del servidor
- Algunos códigos se generan automáticamente:
  - `401 Unauthorized` - la petición requiere que el usuario se autentifique
  - `404 Not Found` - no se ha encontrado ningún método que atienda peticiones en esa URI
  - `405 Method Not Allowed` - URL correcta, pero no soporta el método HTTP solicitado
  - `406 Not Acceptable` - no se pueden generar respuestas con el media type solicitado (cabecera Accept de la petición) (el media type de la anotación @Produces y la cabecera Accept no coinciden)
  - `415 Unsupported Media Type` - el media type de la petición (cabecera Content Type de la petición) no está soportado por el método que sirve el recurso solicitado (media type especificado en la anotación @Consumes)
  - `500 Internal Server Error` - error genérico en el lado del servidor

Clase Response

- Se utiliza para construir la respuesta que nos interesa
- Se define el código de respuesta y el contenido del cuerpo
- Se termina llamando al método `build()` que construye la respuesta
- Consultar en los apuntes el API completo

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Response getClientes() {
  ClientesBean clientes = obtenerClientes();
  return Response.ok(clientes).build();
}

@POST
@Consumes(MediaType.APPLICATION_XML)
public Response addCliente(ClienteBean cliente,
                           @Context UriInfo uriInfo) {
  String id = insertarCliente(cliente);
  URI uri = uriInfo.getAbsolutePathBuilder()
                           .path("{id}").build(id);
  return Response.created(uri).build();
}
```

Un ejemplo de respuesta con cabeceras

```
@Path("/libro")
public class LibroServicio {
  @GET
  @Path("/restfuljava")
  @Produces("text/plain")
  public Response getLibro() {
    String libro = ...;
    ResponseBuilder builder = Response.ok(libro);
    builder.language("fr").header("Some-Header", "some value");
    return builder.build();
  }
}
```

Códigos de estado de la respuesta

- Los códigos en el rango del 100 se consideran informacionales
- Los códigos en el rango del 200 se consideran exitosos
- Los códigos de error pertenecen a los ragos 400 y 500. En el rango de 400 se consideran errores del cliente y en el rango de 500 son errores del servidor

```
public enum Status {
  OK(200, "OK"),
  CREATED(201, "Created"),
  ACCEPTED(202, "Accepted"),
  NO_CONTENT(204, "No Content"),
  MOVED_PERMANENTLY(301, "Moved Permanently"),
  SEE_OTHER(303, "See Other"),
  NOT_MODIFIED(304, "Not Modified"),
  TEMPORARY_REDIRECT(307, "Temporary Redirect"),
  BAD_REQUEST(400, "Bad Request"),
  UNAUTHORIZED(401, "Unauthorized"),
  FORBIDDEN(403, "Forbidden"),
  NOT_FOUND(404, "Not Found"),
  NOT_ACCEPTABLE(406, "Not Acceptable"),
  CONFLICT(409, "Conflict"),
  GONE(410, "Gone"),
  PRECONDITION_FAILED(412, "Precondition Failed"),
  UNSUPPORTED_MEDIA_TYPE(415, "Unsupported Media Type"),
  INTERNAL_SERVER_ERROR(500, "Internal Server Error"),
  SERVICE_UNAVAILABLE(503, "Service Unavailable");
  public enum Family {
     INFORMATIONAL, SUCCESSFUL, REDIRECTION,
     CLIENT_ERROR, SERVER_ERROR, OTHER
  }

  public Family getFamily()
  public int getStatusCode()
  public static Status fromStatusCode(final int
statusCode)
}
```

Índice

- Manejadores de contenidos
- Representaciones XML y JSON
- Respuestas
- **Excepciones**

Excepciones WebApplicationException

- JAX-RS incluye una excepción unchecked que podemos lanzar desde nuestra aplicación RESTful. Esta excepción se puede pre-inicializar con un objeto Response, o con un código de estado particular.

```
@Path("/clientes")
public class ClienteResource {
   @GET
   @Path("{id}")
   @Produces("application/xml")
   public Cliente getCliente(@PathParam("id") int id) {
      Cliente cli = recuperarCliente(id);
      if (cli == null) {
           throw new WebApplicationException(Response.Status.NOT_FOUND);
      }
      return cli;
   }
}
```

Mapeado de excepciones

- Es muy normal que las capas internas de la aplicación lancen excepciones de distintos tipos
- JAX-RS permite capturar estas excepciones y generar una respuesta de error HTTP

```
@Provider
public class EntityNotFoundMapper
          implements ExceptionMapper<EntityNotFoundException> {
   public Response toResponse(EntityNotFoundException e) {
      return Response.status(Response.Status.NOT_FOUND).build();
   }
}
```

Exepción capturada

Método toResponse que devuelve un objeto Respuesta con el código de error HTTP

Excepciones que heredan de WebApplicationException

- En lugar de crear una instancia de `WebApplicationException` e inicializarla con un código de estado específico, podemos utilizar una de las excepciones que heredan de ella
- Por ejemplo, podemos cambiar el ejemplo anterior que utilizaba `WebApplicationException`, y en su lugar, usar `javax.ws.rs.NotFoundException`

```
@Path("/clientes")
public class ClienteResource {
  @GET
  @Path("{id}")
  @Produces("application/xml")
  public Cliente getCliente(@PathParam("id") int id) {
     Cliente cli = recuperarCliente(id);
     if (cli == null) {
          throw new NotFoundException();
     }
     return cli;
  }
}
```

¿Alguna duda, pregunta…?