Servicios REST

Sesión 3:
Manejadores de contenidos. Excepciones. Api cliente.

Índice

- Negociación del contenido
- Representaciones XML y JSON
- Respuestas
- Excepciones
- API Cliente

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"/>
```

Procesamiento de los datos del cuerpo

- El parámetro del método que procesa la petición se mapea con el dato enviado en el cuerpo de la petición
- Podemos definir distintos métodos con distintas anotaciones @Consume y se invoca el método correspondiente al tipo de dato recibido, es lo que en REST se llama *negociación del contenido*
- Si no hay ningún método que pueda consumer el tipo de dato recibido se devuelve una excepción `415 Unsupported Media Type`

Ejemplo

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

EJB 3.0; Bill Burke
```

```
@Consumes("text/*")
@Path("/library")
public class Library {

@POST
public String stringBook(String book) {...}


@Consumes("text/xml")
@POST
public String jaxbBook(Book book) {...}
```

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


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

Tipos aceptados según el content-type

Media Types	Java Type
application/*+xml, text/*+xml, application/*+json, application/*+fastinfoset, application/atom+*	JaxB annotated classes
application/*+xml, text/*+xml	org.w3c.dom.Document
/	java.lang.String
/	java.io.InputStream
text/plain	primtives, java.lang.String, or any type that has a String constructor, or static valueOf(String) method for input, toString() for output
/	javax.activation.DataSource
/	java.io.File
/	byte[]
application/x-www-form-urlencoded	javax.ws.rs.core.MultivaluedMap

Documentación de RESTEasy JAX-RS

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

```
// Leemos los datos del cliente del body del mensaje HTTP
// La cabecera Content-Type tiene que ser text/plain
// 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;
    }
```

Negociación de contenido en peticiones GET

- En las peticiones `GET`, el cliente puede definir el formato esperado con la cabecera `Accept`
- Los métodos definen el tipo de dato devuelto con la anotación `@Produces`
- Distintos métodos pueden devolver distintos formatos
- Si no existe ningún formato compatible se devuelve `406 Not Acceptable`

```
@GET
    @Path("{id}")
    @Produces("application/octet-stream")
    public StreamingOutput recuperarBytesClienteId(@PathParam("id") int id) {
        final Cliente cliente = clienteDB.get(id);
        if (cliente == null) {
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        return new StreamingOutput() {
            public void write(OutputStream outputStream)
                    throws IOException, WebApplicationException {
                escribirCliente(outputStream, 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) {
        ...
     }
}
```

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

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

    @XmlElement
    protected 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;
    }
}
```

```xml
<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 distintos de null), independientemente de que se hayan etiquetado o no
- Si queremos evitar convertir algún atributo debemos marcarlo como `@XmlTransient`
- Con la anotación `@XmlAccessorType(XmlAccessType.NONE)` sólo se convierten a XML los campos anotados

Campos no primitivos

- En el caso de clases con campos de clases también anotadas 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 Customer() {}

    // getters
    ...
}
```

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

  @XmlElement
  protected String ciudad;

  @XmlElement
  protected String codPostal;

  // 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>
```

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

- Es posible convertir colecciones en XML

```java
@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>();
}
```

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

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

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

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

Códigos de respuesta con éxito 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 (por ejemplo un `GET` o un `POST` que devuelve el objeto creado)
  - `204 No Content` si el cuerpo no contiene nada

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

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

  @POST
  @Produces("application/xml")
  @Consumes("application/xml")
  public Cliente crearCliente(Cliente nuevoCli) {...}

  @PUT
  @Path("{id}")
  @Consumes("application/xml")
  public void updateCliente(@PathParam(“id") int id,
                                                Cliente cli) {...}

  @Path("{id}")
  @DELETE
  public void borrarCliente(@PathParam("id") int id) {...}
}
```

Códigos de respuesta de fallo

- Códigos HTTP entre 400 y 599
- Algunos códigos se generan automáticamente:
  - `404 Not Found` - error en la URL
  - `405 Method Not Allowed` - URL correcta, pero no soporta el método HTTP solicitado
  - `406 Not Acceptable` - error en la negociación de contenido

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

```java
@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)
}
```

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

Exepción capturada

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

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;
  }
}
```

API Cliente

- JAX-RS 2.0 incluye un API cliente de servicios REST que permite consultar otros servicios RESTful
- Muy útil para definir tests que prueben los propios servicios que estamos desarrollando
- Para acceder a un recurso REST mediante el API cliente es necesario seguir los siguientes pasos:
  - Obtener una instancia de la interfaz `Client`
  - Configurar la instancia `Client` a través de un target (instancia de `WebTarget`)
  - Crear una petición basada en el target anterior
  - Invocar la petición

Ejemplos (1)

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target(“http://localhost:8080/clientes");
Response response = target.request()
                          .post(Entity.xml(new Cliente("Alvaro", "Gomez")));
response.close();

Cliente cliente = target.queryParam("nombre", "Alvaro Gomez")
                        .request()
                        .get(Cliente.class);
client.close();
```

Ejemplos (2)

```
Client cliente = ClientBuilder.newClient();
PedidoAlmacen pedido = new PedidoAlmacen(...);
WebTarget miRecurso = client.target("http://ejemplo/webapi/escritura");
NumeroSeguimiento numSeg =
                    miRecurso.request(MediaType.APPLICATION_XML)
                             .post(Entity.xml(pedido), NumeroSeguimiento.class);
```

```
List<PedidoAlmacen> pedidos = client.target("http://ejemplo/webapi/lectura")
                                    .path("pedidos")
                                    .request(MediaType.APPLICATION_XML)
                                    .get(new GenericType<List<PedidoAlmacen>>() {});
```

```
Form form = new Form().param("nombre", "Pedro")
                      .param("apellido", "Garcia");
...
response = client.target("http://ejemplo/clientes")
                 .request().
                .post(Entity.form(form));
response.close();
```

Captura de respuestas de error

- Las respuestas de error HTTP se capturan como excepciones

```
try {
        Cliente cli = client.target("http://tienda.com/clientes/123")
                           .request("application/json")
                           .get(Cliente.class);
} catch (NotAcceptableException notAcceptable) {
        ...
} catch (NotFoundException notFound) {
        ...
}
```

Tests en Maven con el API cliente

- Hay que incluir las dependencias de RESTEasy para poder ejecutar los tests en local
- Es necesario desplegar el servicio primero y después lanzar los tests, el orden inverso al habitual de Maven
- En Maven se puede modificar el objetivo en el que se lanzan los tests

```
...
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-client</artifactId>
  <version>3.0.5.Final</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.resteasy</groupId>
  <artifactId>resteasy-jaxb-provider</artifactId>
  <version>2.3.3.Final</version>
</dependency>
...
```

Modificación del orden del lanzamiento de tests

```
<plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-surefire-plugin</artifactId>
   <version>2.18</version>
   <configuration>
        <skip>true</skip>
   </configuration>
   <executions>
      <execution>
        <id>surefire-it</id>
        <phase>integration-test</phase>
        <goals>
          <goal>test</goal>
        </goals>
        <configuration>
          <skip>false</skip>
        </configuration>
      </execution>
    </executions>
</plugin>
```

Referencias

- *Manuales de RESTEasy JAX-RS*
- Bill Burke, RESTfull Java with JAX-RS 2.0, O'Reilly 2013
- Leonard Richardson, Sam Ruby, RESTful Web Services, O'Reilly 2007