UNIVERSIDADE DA CORUÑA

Departamento de Ciencias de la Computación y Tecnologías de la Información

Tema 5. Tutorial de Jakarta RESTful Web Services (JAX-RS)

Integración de Aplicaciones

Índice

- Introducción a Jakarta RESTful Web Services(JAX-RS)
- Ejemplo **`rs-jaxrstutorial`**
  - Implementación de un servicio RESTful
  - Implementación de un cliente de un servicio RESTful

Introducción (1)

- Jakarta RESTful Web Services
  - https://jakarta.ee/specifications/restful-ws/
  - API java estándar para la creación de servicios web RESTful
    - Antes: JAX-RS: Java API for RESTful Web Services (~JSR 339, 370)

- Estudiaremos la versión 3.0
  - Es parte de Jakarta EE Platform 9
  - v1.0 - Septiembre de 2008
  - v1.1 - Septiembre de 2009
  - v2.0 – Mayo 2013
    - Incluye API cliente + hipermedia
  - v2.1 - Septiembre 2019 (Java EE 7 ~ Jakarta EE 8)
  - v3.0 - Noviembre 2020 (Jakarta EE 9)

    Mayo 2021 (Jakarta EE 9.1)
    - Renombrado de paquetes javax.* a jakarta.*
    - Compatible con java 11 (tb java 17)
  - V3.1 – Abril 2022
  - V4.0 – Mayo 2023? (Jakarta EE 10?)

- Java EE, desde 2018 es gestionada por la Eclipse Foundation y pasa a denominarse **Jakarta EE** https://jakarta.ee/
  - Oracle continua liderando la especificación de Java SE
  - Por motivos legales, Jakarta EE 9 renombra paquetería

Introducción (2)

- Típicamente, el programador crea un POJO (Plain Old Java Object) para cada tipo de recurso
- Se usan anotaciones para especificar cómo deben procesarse las peticiones sobre dicho recurso
- Abstrae al programador del servicio de complejidades como
  - Recepción y devolución de cabeceras HTTP
  - Creación de servlets
  - Parsing de URLs
  - Etc.

NOTA: Nos referiremos a JAX-RS o Jakarta RESTful Web Services de forma indistinta a partir de ahora

Introducción (y 3)

- Existen múltiples implementaciones
  - Jersey
    - https://eclipse-ee4j.github.io/jersey/
    - Implementación de referencia, apta para producción
    - Compatible con Jakarta RESTful Web Services 3.x
  - Apache CXF
    - https://cxf.apache.org/
      - Compatible con JAX-RS 2.1
  - RESTEasy
    - https://resteasy.jboss.org/
      - Compatible con Jakarta RESTful Web Services 3.1

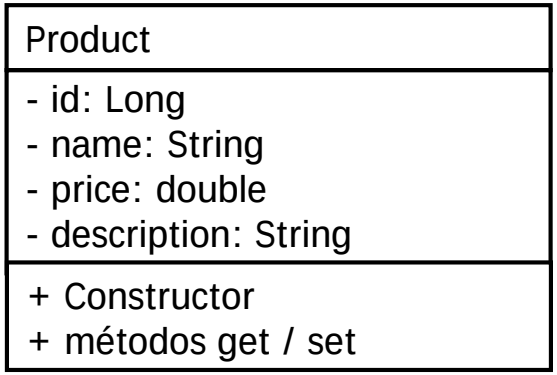
Ejemplo: **`ProductService`** (1)

- Estudiaremos la implementación de un servicio RESTful con JAX-RS a través de un ejemplo
  - Para implementar el servidor utilizaremos los siguientes paquetes
    - **`jakarta.ws.rs`**, **`jakarta.ws.rs.core`** y **`jakarta.ws.rs.ext`**
  - Para implementar el cliente utilizaremos adicionalmente el paquete
    - **`jakarta.ws.rs.client`**
- El ejemplo se encuentra en el módulo **`rs-jaxrstutorial`**
  - Tiene los siguientes submódulos
    - **`rs-jaxrstutorial-service`**
    - **`rs-jaxrstutorial-client`**
  - No se utiliza un diseño por capas
    - Pretende presentar JAX-RS con un ejemplo sencillo

Ejemplo: `ProductService` (2)

- El servicio proporciona operaciones para
  - Añadir un Producto
  - Encontrar un Producto a partir de su identificador
  - Actualizar un Producto
  - Eliminar un Producto
  - Encontrar todos los productos que contengan una cierta palabra clave en su nombre

- Un Producto tiene los siguientes atributos
  - Identificador
  - Nombre
  - Precio
  - Descripción



Ejemplo: **`ProductService`** (y 3)

- La lógica de negocio se simula a través de la clase **`MockProductService`**, que es invocada directamente desde la implementación del servicio web

MockProductService
- products: Map<Long, Product>
+ addProduct(p : Product): Product + findProductById(id : long) : Product + updateProduct(p : Product) : void + removeProduct(ig : long) : void + findProducstByName(keyword : String) : List<Product>

MockProductService.java (1)

```
public class MockProductService {

   private static Map<Long, Product> products =
      new HashMap<Long, Product>();
   private static long lastProductId = 0;

   static {
      try {
         addProduct(new Product(null, "Product 1", 10,
                                "Description of Product 1"));
         addProduct(new Product(null, "Product 2", 20,
                                "Description of Product 2"));
      } catch (InputValidationException e) {
         e.printStackTrace();
      }
   }

   private static synchronized long getNextProductId() {
      return ++lastProductId;
   }
```

MockProductService.java (2)

```
public static Product addProduct(Product p)
      throws InputValidationException {
   validateProduct(p);
   p.setId(getNextProductId());
   products.put(p.getId(), new Product(p));
   return p;
}

public static Product findProductById(long id)
       throws InstanceNotFoundException {
   ...
}

public static void updateProduct(Product p)
      throws InstanceNotFoundException, InputValidationException {
   ...
}
```

MockProductService.java (y 3)

```
    public static void removeProduct(long id)
          throws InstanceNotFoundException {
       ...
    }

    public static List<Product> findProductsByName(String keyword) {
       ...
    }

    private static void validateProduct(Product p)
          throws InputValidationException {
       if (p == null) {
         throw new InputValidationException("Product can't be null.");
       }
       PropertyValidator.validateMandatoryString("name", p.getName());
       PropertyValidator.validateMandatoryString("description",
          p.getDescription());
       PropertyValidator.validateDouble("price", p.getPrice(), 0, 1000);
    }

}
```

Comentarios

- Errores
  - Se reutilizan excepciones definidas en **`ws-util`**
    - Definidas en **`es.udc.ws.util.exceptions`**
  - Incluye dos excepciones para dos tipos de errores “lógicos” muy frecuentes en casi cualquier aplicación
    - **`InstanceNotFoundException`**: indica que se intenta hacer algo sobre un objeto que no existe
      - E.g. buscar producto por id, actualizar o eliminar producto no existente
    - **`InputValidationException`**: indica que se ha proporcionado una información de entrada en formato erróneo
      - E.g. añadir o actualizar producto con datos no válidos (nombre y o descripción vacía, o precio no comprendido entre 0 y 1000)

Protocolo REST (1)

- Seguiremos la misma aproximación que la mayoría de servicios web REST “recientes”
  - URLs únicos y globales para cada recurso
  - Uso consistente de **GET**, **PUT**, **POST** y **DELETE**
  - Uso consistente de los códigos de respuesta HTTP
  - Uso de algunas cabeceras estándar HTTP

Protocolo REST (2)

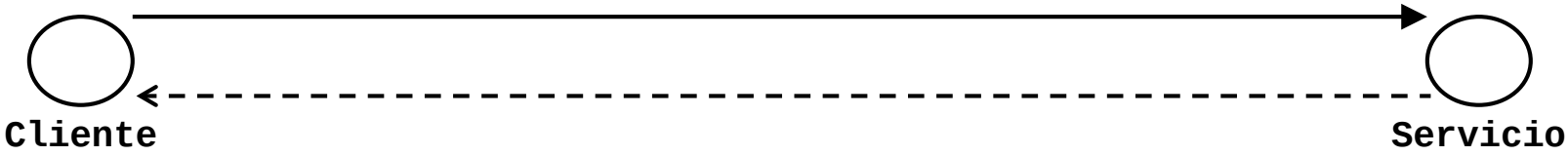
- Recursos
  - **`/products`** Recurso colección
    - **GET**
      - Lista todos los productos
      - El parámetro **`keyword`** permite filtrarlos por palabra clave contenida en el título: **`/products?keyword=ProductName`**
      - La información de los productos se representa en XML en el siguiente formato

```
<?xml version="1.0" encoding="UTF-8"?>
<products xmlns="http://ws.udc.es/products/xml">
  <product>
    <id>1</id>
    <name>Product 1</name>
    <price>10.0</price>
    <description>Description of Product 1</description>
  </product>
  <product>
  ...
  </product>
</products>
```

Protocolo REST (3)

- Listar productos por palabra clave

Petición GET a http://XXX/rs-jaxrstutorial-service/products?keywords=Product




```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<products xmlns="http://ws.udc.es/products/xml">
  <product>
    <id>1</id>
    <name>Product 1</name>
    <price>10.0</price>
    <description>Description of Product 1</description>
  </product>
  <product>
    <id>2</id>
    <name>Product 2</name>
    <price>20.0</price>
    <description>Description of Product 2</description>
  </product>
</products>
```

Protocolo REST (4)

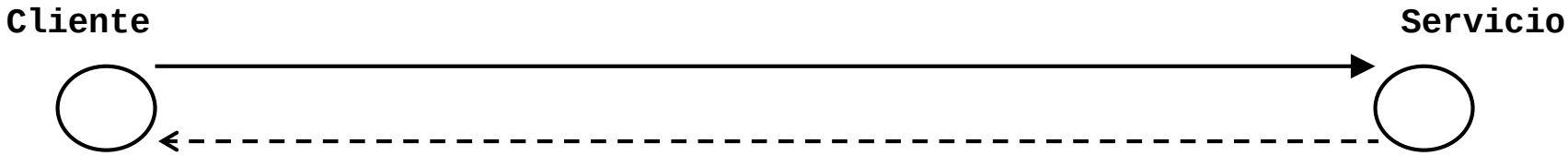
- Recursos
  - **`/products`** Recurso colección
    - **`POST`**
      - Añade un nuevo producto
      - El nuevo producto se envía en el cuerpo de la petición en el formato XML especificado anteriormente (sin identificador)
      - Devuelve el código de respuesta HTTP: **`201 Created`**
      - Devuelve el URL del nuevo producto usando la cabecera estándar HTTP **`Location`**
      - El cuerpo de la respuesta devuelve el nuevo producto creado

Protocolo REST (5)

- Crear un producto

Petición POST a http://XXX/rs-jaxrstutorial-service/products

```
<?xml version="1.0" encoding="UTF-8"?>
<product xmlns="http://ws.udc.es/products/xml">
  <name>Product 3</name>
  <price>30.0</price>
  <description>Description of Product 3</description>
</product>
```




```
HTTP/1.1 201 Created
…
Location: http://XXX/rs-jaxrstutorial-service/products/3

<?xml version="1.0" encoding="UTF-8"?>
<product xmlns="http://ws.udc.es/products/xml">
  <id>3</id>
  <name>Product 3</name>
  ...
</product>
```

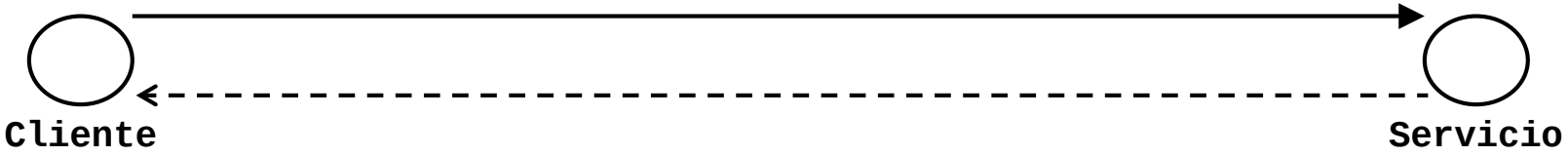
Protocolo REST (6)

- Recursos
  - **`/products/{id}`** Recurso individual por producto
    - **`GET`**
      - Obtiene la información del producto
      - La información de cada producto se representa en XML en el formato mostrado previamente
    - **`PUT`**
      - Modifica el producto
      - El producto se envía en el cuerpo de la petición en el formato XML mostrado previamente
      - El cuerpo de la respuesta va vacío
      - Devuelve el código **`204 No Content`**
    - **`DELETE`**
      - Borra el producto
      - El cuerpo de la respuesta va vacío
      - Devuelve el código **`204 No Content`**

Protocolo REST (7)

- Obtener información de un producto concreto

Petición GET a http://XXX/rs-jaxrstutorial-service/products/3



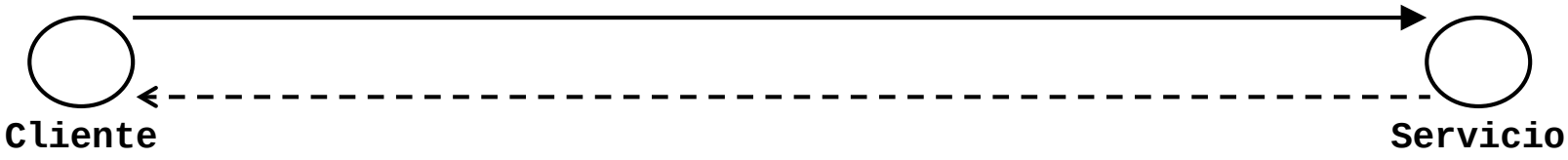

```
HTTP/1.1 200 OK
…

<?xml version="1.0" encoding="UTF-8"?>
<product xmlns="http://ws.udc.es/products/xml">
  <id>3</id>
  <name>Product 3</name>
  ...
</product>
```

Protocolo REST (8)

- Modificar información de un producto

Petición PUT a http://XXX/rs-jaxrstutorial-service/products/3

```
<?xml version="1.0" encoding="UTF-8"?>
<product xmlns="http://ws.udc.es/products/xml">
  <name>New Product Name</name>
  <price>10.0</price>
  <description>Description of Product 3</description>
</product>
```

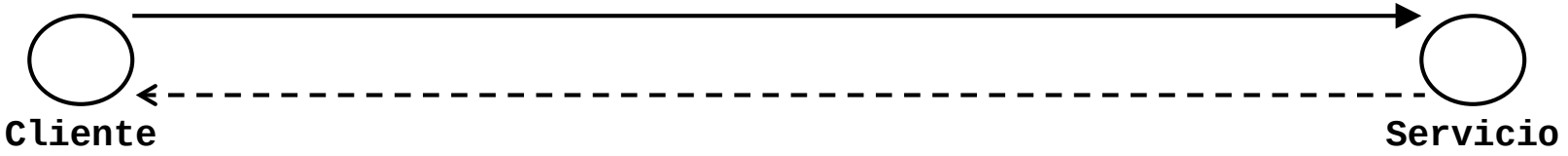



```
HTTP/1.1 204 No Content
…
```

Protocolo REST (9)

- Borrar un producto

Petición DELETE a http://XXX/rs-jaxrstutorial-service/products/3




```
HTTP/1.1 204 No Content
…
```

Protocolo REST (y 10)

- Para los errores generados por la lógica de la aplicación
  - Se utilizan los códigos HTTP más próximos a la semántica de la respuesta
    - Parámetros incorrectos: **`400 Bad Request`**
      - Similar a **`InputValidationException`**
    - Recurso no existe: **`404 Not Found`**
      - Similar a **`InstanceNotFoundException`**
  - El cuerpo del mensaje lleva información adicional
    - Representación en XML de los datos de la excepción
    - E.g. para InstanceNotFoundException

```
HTTP/1.1 404 Not Found
...
<?xml version="1.0" encoding="UTF-8"?>
<instanceNotFoundException xmlns="http://ws.udc.es/products/xml">
  <instanceId>3</instanceId>
  <instanceType>Product</instanceType>
</instanceNotFoundException>
```

Consideraciones de Diseño REST (1)

- Creación de recursos con POST
  - El URL en la cabecera **`Location`** permite al cliente conocer el identificador del nuevo recurso creado para referirse a él más tarde (en nuestro caso, además, tiene existencia real y permite acceder a la representación del recurso)
    - Usar una cabecera estándar permite proporcionar **semántica** para cualquier intermediario y cliente, aunque no conozcan los formatos de nuestro servicio
  - Devolver el nuevo recurso creado en el cuerpo es útil si creemos que el cliente va a utilizarlo de inmediato (ahorra al cliente una petición HTTP)
    - ... pero si la representación puede ser grande y no es seguro que el cliente la necesite inmediatamente, puede ser mejor enviar sólo el URL

Consideraciones de Diseño REST (2)

- Códigos de respuesta de éxito y de error
  - La ventaja de usar códigos estándar es que cualquier cliente o intermediario conoce la semántica de la respuesta sin conocer nuestros formatos específicos
  - Ejemplos según las convenciones de la asignatura
    - Respuesta **`400`** es cacheable, 404 no, **`500`** no es cacheable
      - No tiene sentido reintentar una petición que ha devuelto 400, pero sí una que ha devuelto 500
  - El cuerpo del mensaje puede llevar información adicional no especificable en HTTP para los clientes que sí conozcan nuestros formatos

Consideraciones de Diseño REST (y 3)

- ¿Es totalmente RESTful?
  - No se siguen los principios HATEOAS: no usa hipermedia
  - Representación no autodescriptiva: no usa representaciones en formatos estandarizados como e.g. ATOM

- Sí son características RESTful
  - Cada producto tiene su propio identificador global
  - Uso consistente de la interfaz uniforme para HTTP: **GET**, **POST**, **PUT**, **DELETE** y de los códigos de respuesta HTTP

Aplicación JAX-RS

- Una aplicación JAX-RS está compuesta por **uno o más recursos** y **cero o más proveedores** (providers de ahora en adelante)
  - Un recurso, en el contexto de JAX-RS, es una clase Java que usa anotaciones JAX-RS para implementar un recurso Web
  - Los providers son clases anotadas con **`@Provider`** que implementan una o más interfaces definidas en JAX-RS, y sirven como punto de extensión del entorno de ejecución de JAX-RS
    - Entity providers. Utilizados para mapear entre diferentes representaciones (e.g. XML, JSON) y los tipos Java correspondientes
    - Context providers. Utilizados para proporcionar información de contexto a los recursos u otros providers
    - Exception Mapping providers. Utilizados para realizar mapeos entre excepciones y respuestas

Recursos

- Clases recurso
  - Son POJOs
  - Deben tener un constructor público que el entorno de ejecución de JAX-RS sea capaz de utilizar para instanciar la clase
    - e.g. constructor sin argumentos
  - La anotación **`@Path`**, a nivel de clase, permite especificar patrones de URI a los que se asociará el recurso
- Los métodos de la clase recurso son utilizados para responder a las peticiones realizadas sobre el recurso
  - Deben ser públicos
  - Pueden extender la ruta asociada al recurso utilizando también la anotación **`@Path`**
    - La URI asociada al método es la concatenación de la especificada en el recurso más la especificada en el propio método
  - Las anotaciones **`@GET`**, **`@POST`**, **`@PUT`**, **`@DELETE`** identifican con qué método HTTP se asocia cada método de la clase recurso
    - Solamente es posible aplicar una anotación de este tipo a cada método

ProductResource.java (1)

```
package es.udc.rs.jaxrstutorial.service;
...

@Path("products")
public class ProductResource {

   @POST
   @Consumes(MediaType.APPLICATION_XML)
   @Produces(MediaType.APPLICATION_XML)
   public Response addProduct(Product product, @Context UriInfo ui)
         throws InputValidationException {

      Product addedProduct = MockProductService.addProduct(product);

      return Response.created(
            URI.create(ui.getRequestUri().toString() + "/"
               + addedProduct.getId()))
         .entity(addedProduct)
         .build();
   }
```

ProductResource.java (2)

```
@PUT
@Consumes(MediaType.APPLICATION_XML)
@Produces(MediaType.APPLICATION_XML)
@Path("/{id : \\d+}")
public void updateProduct(Product product, @PathParam("id") long id)
      throws InputValidationException, InstanceNotFoundException {

   product.setId(id);
   MockProductService.updateProduct(product);
}

@DELETE
@Produces(MediaType.APPLICATION_XML)
@Path("/{id : \\d+}")
public void deleteProduct(@PathParam("id") long id)
      throws InstanceNotFoundException {

   MockProductService.removeProduct(id);
}
```

ProductResource.java (y 3)

```
@GET
@Path("/{id : \\d+}")
@Produces(MediaType.APPLICATION_XML)
public Product findProductById(@PathParam("id") long id)
      throws InstanceNotFoundException {

   return MockProductService.findProductById(id);
}

@GET
@Produces(MediaType.APPLICATION_XML)
public List<Product> findProductsByName(
      @DefaultValue("") @QueryParam("keyword") String keyword) {
   return MockProductService.findProducstByName(keyword);
}
```

Recursos – Rutas con plantillas (1)

- En los URIs asociados a un recurso u operación (mediante la anotación **@Path**) es posible utilizar parámetros plantilla
  - E.g. **@Path("/{id}")**
    - La expresión **{id}** es el parámetro plantilla y representa a un comodín con nombre
    - Siempre va entre **{** y **}** y el nombre puede ser cualquier secuencia de caracteres alfanuméricos
    - Encaja con cualquier secuencia de caracteres diferentes del carácter '**/**'
    - Pueden ir en cualquier lugar dentro de la declaración del URI (e.g. **customers/{firstname}-{lastname}**)

Recursos – Rutas con plantillas (y 2)

- Es posible asignarles una expresión regular para que encajen solamente contra cadenas que cumplan esa expresión
  - La expresión regular se especifica después del nombre del parámetro plantilla separada por **“:”**
  - Es el caso de los métodos **`findProductById`**, **`updateProduct`** y **`deleteProduct`**: **`@Path("/{id : \\d+}")`**
    - La expresión **`\\d+`** encaja con una secuencia de números
  - Se pueden utilizar expresiones regulares soportadas por **`java.util.regex.Pattern`**
  - La expresión regular **`.+`** sirve para encajar cualquier secuencia de caracteres (incluido el carácter **‘/’**)
  - Un mismo URI puede encajar con varios patrones de URI declarados en diferentes métodos de una clase recurso

Recursos – Precedencia de rutas (1)

- Reglas para decidir qué método atiende una petición cuando un URI encaja con varios patrones
  - El patrón que tenga más caracteres literales (constantes)
  - El patrón que tenga más parámetros plantilla
  - El patrón que tenga más parámetros plantilla con una expresión regular asociada

  - Por ejemplo, los siguientes patrones están ordenados por orden de precedencia
    - **`/customers/{id}/{name}/address`**
    - **`/customers/{id : .+}/address`**
    - **`/customers/{id}/address`**
    - **`/customers/{id : .+}`**

Recursos – Precedencia de rutas (y 2)

- Cuando no se encuentra ningún método de un recurso con un patrón de URI que encaje con el URI solicitado, se devuelve un error **`404 (Not Found)`** sin cuerpo
  - Por ejemplo si se hace un GET sobre nuestro recurso Products con el URI **`/products/abc`**

- Si se encuentra un método con un patrón de URI que encaje con la petición, pero no está anotado para responder al método HTTP solicitado, entonces se devuelve un error **`405 (Not Allowed)`** sin cuerpo

Recursos – Ciclo de vida

- Ciclo de vida de una clase recurso
  - Por defecto debe crearse una instancia de la clase por cada petición que deba atender
    - Válido para la mayoría de casos
  - La especificación de JAX-RS deja abierta la posibilidad de que las implementaciones ofrezcan otros ciclos de vida alternativos
    - En Jersey es posible especificar ciclos de vida alternativos al por defecto, utilizando anotaciones propietarias
      - **`@Singleton`**. Solamente se crea una instancia de cada clase por aplicación web
      - **`@PerSession`**. Se crea una instancia de cada clase por sesión web y se guarda como un atributo de la sesión

Parámetros – Anotaciones (1)

- Los parámetros de los métodos de una clase recurso pueden estar anotados
  - **@PathParam** permite mapear parámetros plantilla del URI con parámetros del método

```
@Path("/{id}")
public void deleteProduct(@PathParam("id") String id)...
```

  - **@QueryParam** permite mapear parámetros HTTP del URI con parámetros del método

```
public List<Product> findProductsByName(
    @QueryParam("keyword") String keyword) ...
```

  - **@FormParam** permite mapear valores de parámetros codificados en el cuerpo de una petición POST con parámetros del método
  - **@CookieParam** permite mapear valores de cookies con parámetros del método

Parámetros – Anotaciones (y 2)

- Los parámetros de los métodos de un recurso pueden estar anotados (cont.)
  - **`@HeaderParam`** permite mapear valores de cabeceras con parámetros del método
  - **`@MatrixParam`** permite mapear valores de parámetros especificados dentro del path del URI con parámetros del método

```
http://ex.com/rest/categories;name=foo/objects;color=green/?page=1
```

- El valor de un parámetro de un método no anotado se obtiene del cuerpo de la petición correspondiente
  - La conversión entre el cuerpo de la petición y el tipo Java correspondiente es responsabilidad de un "**Entity provider**"

Parámetros – Constructores de recursos

- Constructores
  - Si se utiliza el ciclo de vida por defecto de las clases recurso, entonces los constructores también pueden recibir parámetros anotados con **`@PathParam`**, **`@QueryParam`**, **`@CookieParam`**, **`@HeaderParam`** y **`@MatrixParam`**
    - Los valores de esos parámetros se inyectan cuando se crea una instancia de la clase
      - Con el ciclo de vida por defecto se crea una instancia de la clase para atender a cada petición
      - Si se utiliza otro ciclo de vida, la implementación de JAX-RS debe advertir del uso de estas anotaciones a nivel de constructor

Parámetros – Conversión de tipos (1)

- Conversión de tipos en parámetros anotados
  - Todas las anotaciones comentadas referencian partes de una petición HTTP: se representan como cadenas de caracteres en la petición HTTP
  - JAX-RS se encarga de convertir la cadena de caracteres al tipo Java adecuado
  - El tipo Java debe cumplir alguna de las siguientes restricciones
    - Ser un tipo primitivo
    - Ser un tipo que tenga un constructor que acepte un único argumento de tipo String
    - Ser un tipo que tenga un método estático llamado **`valueOf`** o **`fromString`** y que reciba un único argumento de tipo String
    - Ser **`List<T>`**, **`Set<T>`**, o **`SortedSet<T>`**, siendo **`T`** un tipo que satisface alguno de los dos puntos anteriores

Parámetros – Conversión de tipos (2)

- Conversión de tipos en parámetros anotados (cont.)
  - Si se produce un error al realizar la conversión al tipo
    - Si el parámetro está anotado con **`@MatrixParam`**, **`@QueryParam`** o **`@PathParam`** se devuelve un error 404 (Not Found)
    - Si el parámetro está anotado con **`@HeaderParam`** o **`@CookieParam`** se devuelve un error 400 (Bad Request)

- Parámetros opcionales
  - Cuando la petición HTTP no proporciona la información necesaria para darle valor a un parámetro, JAX-RS le asigna valor **`null`** a los parámetros objetuales y cero a los de tipos primitivos
  - Es posible utilizar la anotación **`@DefaultValue`** para proporcionar un valor por defecto

Parámetros – Conversión de tipos (y 3)

- Si en los métodos **`findProductById`**, **`deleteProduct`** y **`updateProduct`**, quisiésemos devolver otro código de error (e.g. 400) cuando el parámetro **`id`** no está compuesto por números
  - No utilizaríamos ninguna expresión regular para encajar el parámetro **`{id}`**
  - Lo declararíamos de tipo String

```
@GET
@Path("/{id}")
@Produces(MediaType.APPLICATION_XML)
public Product findProductById(@PathParam("id") String id)
      throws InputValidationException, InstanceNotFoundException {
   Long productId;
   try {
      productId = Long.valueOf(id);
   } catch (NumberFormatException ex) {
      // Devolver error 400
      ...
   }
   return MockProductService.findProductById(productId);
}
```

Información de Contexto

- JAX-RS proporciona facilidades para acceder a información de contexto de la aplicación y de cada petición individual
  - Esa información puede ser accedida desde las clases recurso y desde los providers utilizando la anotación **`@Context`** sobre propiedades de clase o parámetros de métodos
  - Entre otros, es posible inyectar propiedades/parámetros de los siguientes tipos
    - **`UriInfo`**
      - Información del URI de la petición (URI completa, parámetros, etc.)
    - **`HttpHeaders`**
      - Acceso a las cabeceras de la petición
    - **`SecurityContext`**
      - Acceso a información de autenticación (e.g. esquema utilizado) y de si la petición ha llegado por una canal seguro
    - **`Request`**
      - Acceso a información de negociación de contenido (**`content-type`**, **`locale`**, **`encoding`**) y precondiciones de la petición (e.g. **`lastModified`**)
    - **`Providers`**
      - Permite inyectar un provider del tipo indicado

Campos y Propiedades

- Es posible utilizar las anotaciones **`@PathParam`**, **`@QueryParam`**, **`@CookieParam`**, **`@HeaderParam`** y **`@MatrixParam`** sobre campos o métodos get/set (propiedades), con la misma semántica y comportamiento que cuando se utilizan sobre parámetros de métodos, siempre y cuando se utilice el ciclo de vida por defecto para las clases recurso
  - Los valores de los campos o propiedades anotados con alguna de estas anotaciones se inyectan cuando se crea una instancia de la clase
  - Con el ciclo de vida por defecto se crea una instancia de la clase para atender a cada petición
- La anotación **`@Context`** siempre se puede utilizar sobre campos o propiedades independientemente del ciclo de vida de las clases recurso
  - La implementación de JAX-RS debe proporcionar los mecanismos necesarios para que la inyección funcione correctamente aunque no se cree una instancia del recurso por petición

Tipos de Contenido (1)

- Cuando una operación recibe contenido en el cuerpo de la petición utiliza la anotación **`@Consumes`** para indicar qué tipo de contenido acepta
  - Un cliente especifica el tipo del cuerpo de la petición mediante la cabecera **`Content-Type`**
  - Los métodos que consumen tipos más específicos (de entre todos los que encajen) tienen preferencia (e.g. **`text/xml`** tiene mayor preferencia que **`text/*`**)
  - Si el servidor no encuentra ningún método capaz de aceptar ese formato, devuelve un error 415 (Unsupported Media Type) sin cuerpo
- La anotación **`@Produces`** indica qué tipo de respuesta va a generar la operación
  - Un cliente especifica mediante la cabecera **`Accept`** la lista de contenidos que acepta como respuesta a una petición
  - Los métodos que producen tipos más específicos (de entre todos los que encajen con alguno de los aceptados por el cliente) tienen preferencia
  - Si el servidor no encuentra ningún método capaz de devolver el resultado en alguno de esos formatos, devuelve un error 406 (Not Aceptable) sin cuerpo
  - El tipo se incluye en la cabecera **`Content-Type`** de la respuesta

Tipos de Contenido (y 2)

- Anotaciones **`@Produces`** y **`@Consumes`**
  - Si no se utilizan se asume **`*/*`** (cualquier formato)
  - Es posible que un método pueda producir o consumir varios formatos (e.g. XML, JSON)
  - También es posible tener distintos métodos para consumir/producir cada formato
    - JAX-RS usará las anotaciones para saber a cuál llamar en cada caso
  - JAX-RS maneja transparentemente todas las cabeceras HTTP involucradas

- En nuestro ejemplo, siempre trabajamos con XML
  - Todas las operaciones que reciben contenido están anotadas con **`@Consumes(MediaType.APPLICATION_XML)`**
  - Todas las operaciones que producen contenido están anotadas con **`@Produces(MediaType.APPLICATION_XML)`**
    - Se incluyen los métodos con tipo de retorno void, si pueden devolver excepciones

Asociación de Peticiones a Métodos

- Cuando el servidor recibe una petición, decide qué método de una clase recurso debe procesarla teniendo en cuenta (y en este orden) ...
  - El URI
  - El método HTTP
  - El tipo de contenido aceptado
  - El tipo de contenido solicitado
    - Se priorizan los que encajan mejor con el contenido aceptado y después los que encajan mejor con el contenido solicitado

Tipos Devueltos (1)

- Los métodos de una clase recurso pueden devolver
  - **`void`**
    - Se devuelve un código 204 sin contenido en el cuerpo
  - **`Response`**
    - Permite construir respuestas en las que se pueden configurar más elementos que el código y el cuerpo de la respuesta
    - Tiene métodos estáticos para crear un **`Response.ResponseBuilder`** que representa a los tipos de respuestas más comunes en HTTP
      - **`created`**. Le asigna el código 201 e incluye el parámetro recibido como valor de la cabecera **`Location`**
      - **`ok`**. Le asigna el código 200 e incluye como cuerpo de la respuesta la entidad que recibe como parámetro
      - Otros: **`notmodified`**, **`nocontent`**, etc.
    - También permite crear un **`Response.ResponseBuilder`** especificando el código de la respuesta deseado, a través del método **`status`**
    - Proporciona métodos adicionales para personalizar la respuesta

Tipos Devueltos (2)

- Los métodos de una clase recurso pueden devolver (cont.)
  - **`Response`** (cont.)
    - Si en el objeto **`Response`** devuelto se estableció una entidad (propiedad **`entity`**) entonces se envía una representación de esa entidad como cuerpo de la respuesta
    - Si en el objeto **`Response`** devuelto no se estableció un código de respuesta (propiedad **`status`**) entonces se devuelve
      - un código 200 si se estableció una entidad no nula
      - un código 204 si la entidad es nula
    - Si el valor (de tipo **`Response`**) devuelto es **`null`** entonces se envía un código 204 sin contenido en el cuerpo

Tipos Devueltos (y 3)

- Los métodos de una clase recurso pueden devolver (cont.)
  - **`GenericEntity`**
    - Como cuerpo de la respuesta se envía una representación de la entidad que se le haya establecido (propiedad **`entity`**)
    - Como código de respuesta se envía 200 si la entidad no es nula o 204 en caso de ser nula
  - Cualquier otro tipo Java
    - Como cuerpo de la respuesta se envía una representación del objeto devuelto
    - Como código de respuesta se envía 200 si se devuelve un valor no nulo o 204 en caso de devolver un valor nulo

- La conversión entre el tipo Java del objeto devuelto por un método, o el tipo Java del objeto establecido como entidad de un objeto **`Response`** o **`GenericEntity`**, y el formato concreto del cuerpo del mensaje de respuesta (e.g. XML) es responsabilidad de un **Entity provider**

GenericEntity

- Cuando se construye un objeto **Response** estableciéndole como entidad un objeto de un tipo parametrizado (e.g. **List<Product>**) es necesario utilizar la clase **GenericEntity** para que en tiempo de ejecución no se pierda el tipo parametrizado de la entidad y sea posible seleccionar el Entity provider adecuado para serializarla
  - Es necesario crear una subclase de **GenericEntity** parametrizándola con el tipo adecuado, y utilizar un objeto de ese tipo
  - A continuación se muestra un ejemplo

```
@GET
@Produces(MediaType.APPLICATION_XML)
public Response findProductsByName(
      @DefaultValue("") @QueryParam("keyword") String keyword) {

   List<Product> list =
      MockProductService.findProducstByName(keyword);

   GenericEntity<List<Product>> entity =
      new GenericEntity<List<Product>>(list) {
   };
   Response response = Response.ok(entity).build();
   return response;
}
```

Entity Providers (1)

- Sirven para convertir entre un tipo Java y un formato de representación concreto
- Deben estar anotados con **`@Provider`**
- Se utilizan
  - Para mapear el cuerpo de las peticiones a los parámetros de entrada de los métodos de las clases recurso
    - Es necesario implementar la interfaz **`MessageBodyReader<T>`**
    - Puede utilizarse la anotación **`@Consumes`** para restringir los tipos de contenido para los que es apropiado
  - Para mapear los valores devueltos por los métodos de las clases recurso al cuerpo de las respuestas
    - Es necesario implementar la interfaz **`MessageBodyWriter<T>`**
    - Puede utilizarse la anotación **`@Produces`** para restringir los tipos de contenido para los que es apropiado

ProductReader.java

```
@Consumes(MediaType.APPLICATION_XML)
@Provider
public class ProductReader implements MessageBodyReader<Product> {

   @Override
   public boolean isReadable(Class<?> type, Type genericType,
         Annotation[] antns, MediaType mt) {
      return type == Product.class;
   }

   @Override
   public Product readFrom(Class<Product> type, Type genericType,
         Annotation[] antns, MediaType mt,
         MultivaluedMap<String, String> mm, InputStream in)
         throws IOException, WebApplicationException {
      return XmlConversor.toProduct(in);
   }

}
```

ProductWriter.java

```
@Produces(MediaType.APPLICATION_XML)
@Provider
public class ProductWriter implements MessageBodyWriter<Product> {
   @Override
   public boolean isWriteable(Class<?> type, Type genericType,
         Annotation[] antns, MediaType mt) {
      return type == Product.class;
   }

   @Override
   public long getSize(Product t, Class<?> type, Type genericType,
        Annotation[] antns, MediaType mt) {
      return -1;
   }

   @Override
   public void writeTo(Product t, Class<?> type, Type type1,
         Annotation[] antns, MediaType mt,
         MultivaluedMap<String, Object> mm, OutputStream out)
         throws IOException, WebApplicationException {
      XmlConversor.toXML(t, out, true);
   }
}
```

ProductListWriter.java (1)

```
@Produces(MediaType.APPLICATION_XML)
@Provider
public class ProductListWriter implements  MessageBodyWriter<List<Product>> {

@Override
public boolean isWriteable(Class<?> type, Type genericType,
      Annotation[] antns, MediaType mt) {
   boolean isWritable;
   if (List.class.isAssignableFrom(type)
      && genericType instanceof ParameterizedType) {
      ParameterizedType parameterizedType = (ParameterizedType) genericType;
      Type[] actualTypeArgs = (parameterizedType.getActualTypeArguments());
      isWritable = (actualTypeArgs.length == 1 &&
                    actualTypeArgs[0].equals(Product.class));
   } else {
      isWritable = false;
   }
   return isWritable;
}
```

ProductListWriter.java (y 2)

```
    @Override
    public long getSize(List<Product> t, Class<?> type, Type genericType,
          Annotation[] antns, MediaType mt) {
       return -1;
    }

    @Override
    public void writeTo(List<Product> t, Class<?> type, Type genericType,
          Annotation[] antns, MediaType mt,
          MultivaluedMap<String, Object> mm, OutputStream out)
          throws IOException, WebApplicationException {
       XmlConversor.toXML(t, out, true);
    }

}
```

Entity Providers (2)

- **`MessageBodyReader`**
  - El método **`isReadable`** permite determinar a qué objetos puede aplicarse esta implementación
    - **`ProductReader`** puede aplicarse para construir parámetros de tipo **`Product`** partiendo del cuerpo de una petición en formato XML
      - Necesario en **`addProduct`** y **`updateProduct`**
  - El método **`readFrom`** debe encargarse de leer, del **`InputStream`** que recibe como parámetro, la representación textual del cuerpo de la petición y devolver un objeto del tipo adecuado
    - **`ProductReader`** utiliza la clase utilidad **`XmlConversor`** (internamente utiliza JDOM) para parsear XML y devolver un objeto de tipo **`Product`**

Entity Providers (3)

- **`MessageBodyWriter`**
  - El método **`isWriteable`** permite determinar a qué objetos puede aplicarse esta implementación
    - **`ProductWriter`** se utiliza para construir el cuerpo de una respuesta en formato XML partiendo de un objeto de tipo **`Product`**
      - Necesario en **`addProduct`** y **`findProductById`**
    - **`ProductListWriter`** se utiliza para construir el cuerpo de una respuesta en formato XML partiendo de un objeto de tipo **`List<Product>`**
      - En este caso hay que comprobar que el tipo es un subtipo de **`List`** y que es un tipo parametrizado con un único parámetro de tipo **`Product`**
      - Necesario en **`findProductsByName`**
  - El método **`getSize`** se usa para fijar la cabecera **`Content-Length`** (-1 cuando no se conoce por anticipado)
  - El método **`writeTo`** debe encargarse de transformar el objeto recibido a texto, en el formato de representación adecuado, y escribirlo en el **`OutputStream`** que recibe como parámetro
    - **`ProductWriter`** / **`ProductListWriter`** utilizan la clase utilidad **`XmlConversor`** (internamente utiliza JDOM) para generar XML a partir de un objeto de tipo **`Product / List<Product>`**

Entity Providers (4)

- Cuando un método de una clase recurso recibe / devuelve un objeto, JAX-RS busca un **`Reader`** / **`Writer`** apropiado en función del tipo Java del objeto y el tipo de contenido especificado en la petición / esperado en la respuesta
  - Si no se encuentra un **`MessageBodyReader`** apropiado, se genera una respuesta de error con código 415 (Unsupported Media Type) y cuerpo vacío
  - Si no se encuentra un **`MessageBodyWriter`** apropiado, se genera una respuesta de error con código 500 (Internal Server Error) y cuerpo vacío

Entity Providers (y 5)

- La especificación de JAX-RS establece que cualquier implementación debe proporcionar implementaciones de **`MessageBodyReader`** y **`MessageBodyWriter`** para una serie de tipos Java y formatos específicos
- Algunos de ellos son
  - **`byte[]`**, **`String`**, **`InputString`**, **`Reader`** y **`File`** para todos los tipos de contenido (**`*/*`**)
  - **`MultivaluedMap<String,String>`** para el contenido **`application/x-www-form-urlencoded`**
    - Para parámetros enviados utilizando **`POST`**
  - **`StreamingOutput`** para todos los tipos (**`*/*`**)
    - Solamente **`MessageBodyWriter`**
    - Permite hacer raw streaming de los cuerpos de las respuestas
  - ...
- Algunos pueden depender el entorno. En el ámbito de esta asignatura:
  - Clases anotadas con Jakarta XML Binding (JAXB) para los tipos xml (**`text/-xml`**, **`application/xml`** y **`application/*+xml`**)
    - Lo veremos en los Temas 6 y 7

Tratamiento de Errores (1)

- Es posible enviar una respuesta de error a un cliente de dos maneras
  - Construyendo y devolviendo el objeto **`Response`** adecuado (con el código de error y cuerpo del mensaje pertinentes)
    - Solamente válido para métodos que devuelvan un objeto de tipo **`Response`**
    - Es necesario capturar explícitamente todas las excepciones que puedan lanzarse y en cada caso construir el objeto **`Response`** adecuado
  - Lanzando una excepción
- Excepciones
  - Los métodos de las clases que conforman una aplicación JAX-RS pueden lanzar excepciones de tipo checked (hijas de **`Exception`**) o de tipo unchecked (hijas de **`RuntimeException`**)
  - El entorno de ejecución de JAX-RS trata todos los tipos de excepciones para los cuales la aplicación proporcione un Exception Mapping provider
    - Un Exception Mapping provider se encarga de convertir un tipo de excepción a un objeto **`Response`**, a partir del cual se genera la respuesta HTTP

Tratamiento de Errores (2)

- Excepciones (cont.)
  - Si la aplicación JAX-RS no proporciona un Exception Mapping provider para un cierto tipo de excepción, entonces se propaga hacia el contenedor de Servlets, dentro del cual se está ejecutando la aplicación, y es tratada por este
  - JAX-RS proporciona una excepción, llamada **`WebApplicationException`**, para la cual no es necesario escribir un Exception Mapping provider
    - Es capturada y procesada automáticamente por el entorno de ejecución de JAX-RS
    - Si la excepción se ha creado especificando un código de respuesta o un objeto **`Response`** entonces serán utilizados para crear la respuesta a enviar al cliente
      - Si se ha especificado una entidad en el objeto **`Response`**, debe proporcionarse un **`MessageBodyWriter`** adecuado para ese tipo de entidad (igual que si se tratase de un objeto devuelto por un método de una clase recurso)
    - En otro caso se envía un código de error 500, “Internal Server Error” al cliente

Tratamiento de Errores (y 3)

```
@DELETE
@Produces(MediaType.APPLICATION_XML)
@Path("/{id : \\d+}")
public void deleteProduct(@PathParam("id") long id) {

   try {
      MockProductService.removeProduct(id);
   } catch (InstanceNotFoundException e) {
      throw new WebApplicationException(Response
            .status(Response.Status.NOT_FOUND)
            .entity(e)
            .build());
   }
}
```

- Cuando el código que es invocado desde la capa de servicios puede lanzar la misma excepción desde varios puntos resultaría muy tedioso capturarla y relanzarla como una **`WebApplicationException`** en todos esos puntos
  - Si siempre que salta un cierto tipo de Excepción se quiere enviar el mismo tipo de respuesta entonces es más cómodo proporcionar un **Exception Mapping provider**

Exception Mapping Providers (1)

- Deben estar anotados con **`@Provider`**
- Deben implementar la interfaz **`ExceptionMapper<T>`**
- El método **`toResponse`**
  - Recibe una excepción de un determinado tipo
  - Devuelve el objeto **`Response`** a partir del cual se creará la respuesta HTTP a enviar al cliente, cuando algún método de la aplicación lance ese tipo de excepción
- No es necesario capturar la excepción explícitamente en ningún punto de nuestro código
  - Cuando un método de la aplicación JAX-RS lanza una excepción, el entorno de ejecución comprueba si existe un Exception Mapper para tratar ese tipo de excepción

InstanceNotFoundExceptionMapper.java

```
@Provider
public class InstanceNotFoundExceptionMapper implements
      ExceptionMapper<InstanceNotFoundException> {

   @Override
   public Response toResponse(InstanceNotFoundException ex) {
      return Response.status(Response.Status.NOT_FOUND)
               .entity(ex).build();
   }

}
```

InputValidationExceptionMapper.java

```
@Provider
public class InputValidationExceptionMapper implements
      ExceptionMapper<InputValidationException> {

   @Override
   public Response toResponse(InputValidationException ex) {
      return Response.status(Response.Status.BAD_REQUEST)
               .entity(ex).build();
      }

}
```

InstanceNotFoundExceptionWriter.java (1)

```
@Produces(MediaType.APPLICATION_XML)
@Provider
public class InstanceNotFoundExceptionWriter implements
      MessageBodyWriter<InstanceNotFoundException> {

   @Override
   public boolean isWriteable(Class<?> type, Type genericType,
         Annotation[] antns, MediaType mt) {
      return type == InstanceNotFoundException.class;
   }

   @Override
   public long getSize(InstanceNotFoundException t, Class<?> type,
         Type genericType, Annotation[] antns, MediaType mt) {
      return -1;
   }
```

InstanceNotFoundExceptionWriter.java (y 2)

```
    @Override
    public void writeTo(InstanceNotFoundException t, Class<?> type,
          Type genericType, Annotation[] antns, MediaType mt,
          MultivaluedMap<String, Object> mm, OutputStream out)
          throws IOException, WebApplicationException {
       XmlConversor.toXML(t, out, true);
    }

}
```

InputValidationExceptionWriter.java (1)

```
@Produces(MediaType.APPLICATION_XML)
@Provider
public class InputValidationExceptionWriter implements
      MessageBodyWriter<InputValidationException> {

   @Override
   public boolean isWriteable(Class<?> type, Type genericType,
         Annotation[] antns, MediaType mt) {
      return type == InputValidationException.class;
   }

   @Override
   public long getSize(InputValidationException t, Class<?> type,
         Type genericType, Annotation[] antns, MediaType mt) {
      return -1;
   }
```

InputValidationExceptionWriter.java (y 2)

```
    @Override
    public void writeTo(InputValidationException t, Class<?> type,
          Type genericType, Annotation[] antns, MediaType mt,
          MultivaluedMap<String, Object> mm, OutputStream out)
          throws IOException, WebApplicationException {
       XmlConversor.toXML(t, out, true);
    }

}
```

Exception Mapping Providers (y 2)

- En nuestro ejemplo se propocionan Exception Mapping providers para los dos tipos de excepciones que pueden saltar dentro de la implementación de los métodos de la clase recurso **`ProductResource`**
  - **`InstanceNotFoundException`**
    - Se devuelve un código de error 404 (Not Found)
    - Como entidad del objeto **`Response`** se establece la propia excepción
      - Se proporciona un **`MessageBodyWriter`** que delega en la clase utilidad **`XmlConversor`** para obtener una representación en XML de los objetos de tipo **`InstanceNotFoundException`**
  - **`InputValidationException`**
    - Se devuelve un código de error 400 (Bad Request)
    - Como entidad del objeto **`Response`** se establece la propia excepción
      - Se proporciona un **`MessageBodyWriter`** que delega en la clase utilidad **`XmlConversor`** para obtener una representación en XML de los objetos de tipo **`InstanceNotFoundException`**

Providers

- Ciclo de vida
  - Cada aplicación crea una única instancia de cada clase provider
  - Es posible que haya varios Threads (correspondientes a diferentes peticiones) ejecutando simultáneamente el mismo método
- Constructores
  - Deben tener un constructor público que el entorno de ejecución de JAX-RS sea capaz de utilizar para instanciar la clase (e.g. constructor sin argumentos)

Configuración de una Aplicación JAX-RS (1)

- Los recursos y providers que forman una aplicación JAX-RS, pueden especificarse a través de una subclase de la clase **`Application`**
  - Debe proporcionarla la aplicación
  - Han de sobreescribirse los métodos que especifican el conjunto de clases que conforman la aplicación
- La especificación de JAX-RS deja abierta la posibilidad de que las implementaciones proporcionen sus propios mecanismos para localizar las clases que representan recursos y providers
  - E.g. hacer scanning de clases en tiempo de ejecución

web.xml

```
...
<servlet>
  <servlet-name>RESTFulServlet</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>es.udc.rs.jaxrstutorial.service</param-value>
  </init-param>
</servlet>

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

Configuración de una Aplicación JAX-RS (y 2)

- En Jersey debe declararse un servlet de tipo **`ServletContainer`** y asociarle los URLs que debe procesar la aplicación JAX-RS (en nuestro ejemplo todas)
  - A través del parámetro de inicialización del servlet **`jersey.config.server.provider.packages`** indicamos los paquetes en los que se localizarán clases recurso y providers
    - **`es.udc.rs.jaxrstutorial.service`**

API Cliente de JAX-RS (a partir de v2.0)

- Orientada a recursos
  - Cada recurso está identificado por un URI
  - El cliente interactúa con los recursos a través de peticiones HTTP utilizando un conjunto fijo de métodos HTTP
  - Es posible obtener una o más representaciones, identificadas por el tipo de contenido (media type)
- Reutiliza partes de la API estándar de JAX-RS ya vistas para la parte servidor
  - Interfaces **`MessageBodyReader`** y **`MessageBodyWriter`**

ClientProductService.java (1)

```
public class ClientProductService {

   private static Client client = null;

   private final static String ENDPOINT_ADDRESS_PARAMETER =
         "ClientProductService.endpointAddress";
   private WebTarget endPointWebTarget = null;

   private static Client getClient() {
     if (client == null) {
       client = ClientBuilder.newClient();
       client.register(ProductReader.class);
       client.register(ProductWriter.class);
       client.register(ProductListReader.class);
       client.register(InstanceNotFoundExceptionReader.class);
       client.register(InputValidationExceptionReader.class);
     }
     return client;
   }
```

ClientProductService.java (2)

```
private WebTarget getEndpointWebTarget() {

  if (endPointWebTarget == null) {
    endPointWebTarget = getClient()
      .target(ConfigurationParametersManager
                .getParameter(ENDPOINT_ADDRESS_PARAMETER));
  }
  return endPointWebTarget;

}
```

ClientProductService.java (3)

```
public Long addProduct(ClientProduct product)
  throws InputValidationException {

  WebTarget wt = getEndpointWebTarget().path("products");
  Response response = wt.request()
    .accept(MediaType.APPLICATION_XML)
    .post(Entity.entity(product, MediaType.APPLICATION_XML));
  try {
    validateResponse(Response.Status.CREATED.getStatusCode(), response);
    ClientProduct resultProduct = response.readEntity(ClientProduct.class);
    return resultProduct.getId();
  } catch (InputValidationException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }

}
```

ClientProductService.java (4)

```
public void updateProduct(ClientProduct product)
  throws InstanceNotFoundException, InputValidationException {

  WebTarget wt = getEndpointWebTarget().path("products/{id}")
    .resolveTemplate("id", product.getId());
  Response response = wt.request()
    .accept(MediaType.APPLICATION_XML)
    .put(Entity.entity(product, MediaType.APPLICATION_XML));
  try {
    validateResponse(Response.Status.NO_CONTENT.getStatusCode(), response);
  } catch (InstanceNotFoundException | InputValidationException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }

}
```

ClientProductService.java (5)

```
public void deleteProduct(Long id) throws InstanceNotFoundException {

  WebTarget wt = getEndpointWebTarget().path("products/{id}")
    .resolveTemplate("id", id);
  Response response = wt.request()
    .accept(MediaType.APPLICATION_XML)
    .delete();
  try {
    validateResponse(Response.Status.NO_CONTENT.getStatusCode(), response);
  } catch (InstanceNotFoundException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }
}
```

ClientProductService.java (6)

```
public ClientProduct findProduct(Long id)
  throws InstanceNotFoundException {

  WebTarget wt = getEndpointWebTarget().path("products/{id}")
    .resolveTemplate("id", id);
  Response response = wt.request()
    .accept(MediaType.APPLICATION_XML)
    .get();
  try {
    validateResponse(Response.Status.OK.getStatusCode(), response);
    ClientProduct product = response.readEntity(ClientProduct.class);
    return product;
  } catch (InstanceNotFoundException ex) {
      throw ex;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }
}
```

ClientProductService.java (7)

```
public List<ClientProduct> findProducts(String keyword) {

  WebTarget wt = getEndpointWebTarget().path("products")
    .queryParam("keyword", keyword);
  Response response = wt.request()
    .accept(MediaType.APPLICATION_XML)
    .get();
  try {
    validateResponse(Response.Status.OK.getStatusCode(), response);
    List<ClientProduct> products =
      response.readEntity(new GenericType<List<ClientProduct>>() {   });
    return products;
  } catch (Exception ex) {
      throw new RuntimeException(ex);
  } finally {
      if (response != null) {
        response.close();
      }
  }

}
```

ClientProductService.java (y 8)

```
private void validateResponse(int expectedStatusCode, Response response)
  throws InstanceNotFoundException, InputValidationException {

  Response.Status statusCode = Response.Status.fromStatusCode(response.getStatus());

  if (statusCode.getStatusCode() == expectedStatusCode) { return; }

  String contentType =
    response.getMediaType() != null ? response.getMediaType().toString() : null;
  boolean expectedContentType = MediaType.APPLICATION_XML
    .equalsIgnoreCase(contentType);

  if (!expectedContentType && statusCode != Response.Status.NO_CONTENT)) {
    throw new RuntimeException("HTTP error; status code = " + statusCode);
  }

  switch (statusCode) {
    case NOT_FOUND: { throw response.readEntity(InstanceNotFoundException.class); }
    case BAD_REQUEST: { throw response.readEntity(InputValidationException.class); }
    default:
        throw new RuntimeException("HTTP error; status code = " + statusCode);
}

}
```

`Client` y `WebTarget` (1)

- La interfaz **`Client`** es el punto de entrada a la API cliente de JAX-RS
- Las instancias se crean utilizando la clase **`ClientBuilder`**
  - Utilizaremos el método estático **`newClient`** para crear una instancia utilizando la implementación por defecto
- **`Client`**
  - Implementa a su vez la interfaz **`Configurable`**
  - Es necesario especificarle el conjunto de clases provider que serán utilizadas por la aplicación cliente
    - Invocando el método **`register(Class),`** especificando las clases provider una a una (no es posible especificar un conjunto de paquetes como en el lado servidor)
    - Misma función que en el servidor (Entity y Context providers)
    - Se crea una única instancia de cada clase provider, que será compartida por toda la aplicación cliente

`Client` y `WebTarget` (2)

- A partir de la instancia de **`Client`** se pueden obtener “referencias” a los recursos a través de su URI (instancias de la interfaz **`WebTarget`**)
  - Se utiliza el método **`target`** pasándole el URI del recurso
- Es costoso crear instancias de la clase **`Client`**, por lo que es recomendable tener una única instancia a partir de la cual obtener las instancias de recursos **`WebTarget`**
  - La creación de objetos de tipo **`WebTarget`**, y la construcción de peticiones y recepción de respuestas a través de ellos son “thread safe”
    - Las instancias de las clases **`Client`** y **`WebTarget`** pueden ser compartidas por varios Threads
  - NOTA
    - Es recomendable invocar el método **`close`** sobre una instancia de la clase **`Client`** cuando deje de ser usada para asegurar la rápida liberación de recursos (gestionan sockets)

`Client` y `WebTarget` (y 3)

- Una instancia de WebTarget incluye métodos adicionales para extender la URI al recurso
  - Es posible completar la ruta con el método **`path`**
  - Es posible indicar valores de parámetros si la ruta representa una plantilla, usando el método **`resolveTemplate`**
    - Dispone de varias firmas. En general, se indica el valor de parámetros de la plantilla
  - Es posible indicar parámetros adicionales en la query que representa la URI, con el método **`queryParam`**

Peticiones y Respuestas (1)

- Las peticiones a los recursos se construyen a partir de un **`WebTarget`** utilizando el patrón builder
  - En primer lugar se invoca el método **`request`** que devuelve una instancia de **`Invocation.Builder`**
  - Un **`Invocation.Builder`** incluye métodos para preparar una petición
    - Especificar los tipos de contenido aceptados por el cliente (método **`accept`**)
      - También es posible indicarlos como parámetro del método **`request`**, pero es más legible utilizar el método **`accept`**
    - Especificar cabeceras con el método **`header`**
    - Especificar cookies con el método **`cookie`**
    - Etc.
  - Una vez preparada la petición, un **`Invocation.Builder`** dispone de métodos para crear una invocación (interfaz **`Invocation`**)
    - **`get`**, **`post`**, **`put`**, **`delete`**, ...

Peticiones y Respuestas (2)

- Intefaz **`Invocation`** (cont.)
  - Métodos **`post`**, **`put`**
    - Permiten indicar el tipo mime de la entidad enviada como cuerpo y el tipo Java al que debe convertirse la respuesta
  - Métodos **`delete`**, **`get`**
    - Permiten indicar el tipo Java al que debe convertirse la respuesta
- Para especificar la entidad de una petición se utiliza la clase **`Entity`**
  - Dispone de varios métodos estáticos para crear una entidad a partir de un objeto y su tipo mime
    - **`Entity.entity`**
      - » **`E.g. Entity.entity(xx,MediaType.APPLICATION_XML)`**
  - También dispone de métodos específicos para convertir entidades a tipos mime concretos
    - **`Entity.xml(xx)`**
    - **`Entity.json(xx)`**

Peticiones y Respuestas (3)

- Existen dos posibilidades para indicar el tipo Java al que debe convertirse la respuesta
  - Si la respuesta devuelve una entidad en su cuerpo, entonces puede declararse el tipo Java al que debe convertirse
    - El cuerpo de la respuesta será deserializado a un objeto de ese tipo Java
    - Si se produce una excepción procesando la respuesta (**`ResponseProcessingException`**) o el servidor ha devuelto un código de error HTTP (**`WebApplicationException`**), es posible obtener la instancia de la respuesta completa utilizando su método **`getResponse`**
    - JAX-RS define una jerarquía de excepciones para WebApplicationException
      - » **`RedirectionException`** (código HTTP 3xx)
      - » **`ClientErrorException`** (código HTTP 4xx): **`BadRequestException`**, **`ForbiddenException`**, **`NotAcceptableException`**, **`NotAllowedException`**, **`NotAuthoriredException`**, **`NotFoundException`**, **`NotSupportedException`**
      - » **`ServerErrorException`** (código HTTP 5xx)

Peticiones y Respuestas (4)

- Existen dos posibilidades para indicar el tipo Java al que debe convertirse la respuesta (cont.)
  - Si es necesario acceder a más información de la respuesta además de la entidad devuelta en el cuerpo, entonces no se indica el tipo en la invocación y se obtiene una instancia de **Response**
    - Permite acceder al código de respuesta, cabeceras y entidad del cuerpo
    - Es el utilizado en el ejemplo, ya que para todas las peticiones se valida que el código y tipo de contenido de la respuesta son los esperados

Peticiones y Respuestas (5)

- **`Response` d**ispone del métodos para
  - Obtener el código HTTP de la respuesta (método **`getStatus`**)
  - Obtener el tipo mime de la respuesta (método **`getMediaType`**)
  - Obtener la entidad contenida en la respuesta como un objeto java del tipo especificado (**`readEntity`**)
    - Devuelve la excepción **`ProcessingException`** (unchecked) si no se puede convertir a ese tipo

- NOTA importante
  - Los objetos **`Response`** referencian a sockets abiertos
  - La especificación establece que se cierren de forma automática tras haber leído el contenido de la entidad de respuesta o cuando el garbagge collector lo reclame, pero por eficiencia se recomienda cerrarlos lo antes posible de forma explícita utilizando el método **`close`**

Peticiones y Respuestas (y 6)

- Para convertir entre los tipos Java de las entidades a enviar o recibir y su representación textual en el tipo de contenido adecuado, se utiliza el mismo mecanismo que en la parte servidora
  - Los que dice la especificación de JAX-RS
  - Para dar soporte a otros tipos es necesario proporcionar Entity providers
    - Implementar **`MessageBodyReader<T>`** para los objetos a recibir como cuerpo de una respuesta
    - Implementar **`MessageBodyWriter<T>`** para los objetos a enviar como cuerpo de una petición

`ClientProductService` (1)

- El método **`getClient`** permite obtener y cachear una instancia del cliente con los providers necesarios configurados
- El método **`getEndpointWebTarget`** crea el **`WebTarget`** base a partir del que se crearán los diferentes recursos a los que accede este cliente

`ClientProductService` (2)

- El método validate
  - Recibe
    - El código de respuesta esperado
    - El objeto **`Response`** con la respuesta recibida
  - Comprueba si el código de respuesta es el esperado y en ese caso finaliza
  - Comprueba que el tipo de contenido recibido es XML y si no lo es y se ha recibido un código de error se lanza una excepción de Runtime
  - Si se ha recibido un código de error 400 o 404 se deserializa el cuerpo de la respuesta a las excepciones **`InputValidationException`** o **`InstanceNotFoundException`** respectivamente
    - Sabemos que en estos casos el servidor ha incluido información sobre las excepciones en el cuerpo de la respuesta
  - En otro caso, se lanza una excepción de Runtime

`ClientProductService` (3)

- Tipos Java enviados como cuerpo de peticiones (requieren una implementación de **`MessageBodyWriter`**)
  - **`ClientProduct`** (**`addProduct`**, **`updateProduct`**)
    - **`ProductWriter`**
- Tipos Java recibidos como cuerpo de peticiones (requieren una implementación de **`MessageBodyReader`**)
  - **`ClientProduct`** (**`findProduct`**)
    - **`ProductReader`**
  - **`List<ClientProduct>`** (**`findProducts`**)
    - **`ProductListReader`**
  - **`InputValidationException`** (**`addProduct`**, **`updateProduct`**)
    - **`InputValidationExceptionReader`**
  - **`InstanceNotFoundException`** (**`updateProduct`**, **`deleteProduct`**, **`findProduct`**)
    - **`InstanceNotFoundExceptionReader`**

`ClientProductService` (4)

- Las clases **`Product, ProductReader`** y **`ProductWriter`** se utilizan tanto en el servicio como en el cliente
  - Se han replicado en ambos módulos en lugar de crear un módulo común en el que compartirlas porque

    1) no es lo habitual en este tipo de servicios reutilizar código del servicio en el cliente

    - El cliente no tiene por qué estar implementado en el mismo lenguaje de programación que el servicio

    2) en la práctica utilizaremos una tecnología que nos evitará en general tener que crear estos objetos
  - En el caso de **`Product`** se ha decidido utilizar como convención llamarla **`ClientProduct`**
    - Para **`ProductReader/ProductWriter`** podría haberse seguido la misma convención

`ClientProductService` (y 5)

- Cuando el cuerpo de una respuesta quiere convertirse a un tipo parametrizado es necesario utilizar la clase **`GenericType<T>`**
  - Es necesario crear una subclase de **`GenericType`** parametrizándola con el tipo adecuado, e indicar un objeto de ese tipo

- Es el caso de **`findProducts`**
  - No es posible especificar como tipo **`List<ClientProduct>.class`**
  - Se utiliza una instancia de un subclase anónima de **`GenericType`** parametrizada con el tipo **`List<ClientProduct>`**