



Servicios REST

Sesión 1:

Introducción a REST. Diseño y creación de servicios RESTful



Índice

- **Servicios web**
- Fundamentos de REST
- Diseño de servicios web RESTful
- Implementación de un primer servicio JAX-RS



¿Qué es un servicio web?

*“Sistema software diseñado para soportar de modo **interoperable** interacciones **máquina a máquina** a través de la red”*

W3C, Web Services Architecture

- Claves:
 - **Interoperabilidad:** un servicio puede ser llamado por cualquier aplicación, usando cualquier lenguaje de programación
 - **Máquina a máquina:** podemos crear una aplicación usando servicios como módulos
 - **Web:** las llamadas se hacen a través de HTTP
- Servicio web \neq Aplicación web
 - Servicio web = interacción **máquina - máquina**
 - Aplicación web = interacción **humano - máquina**



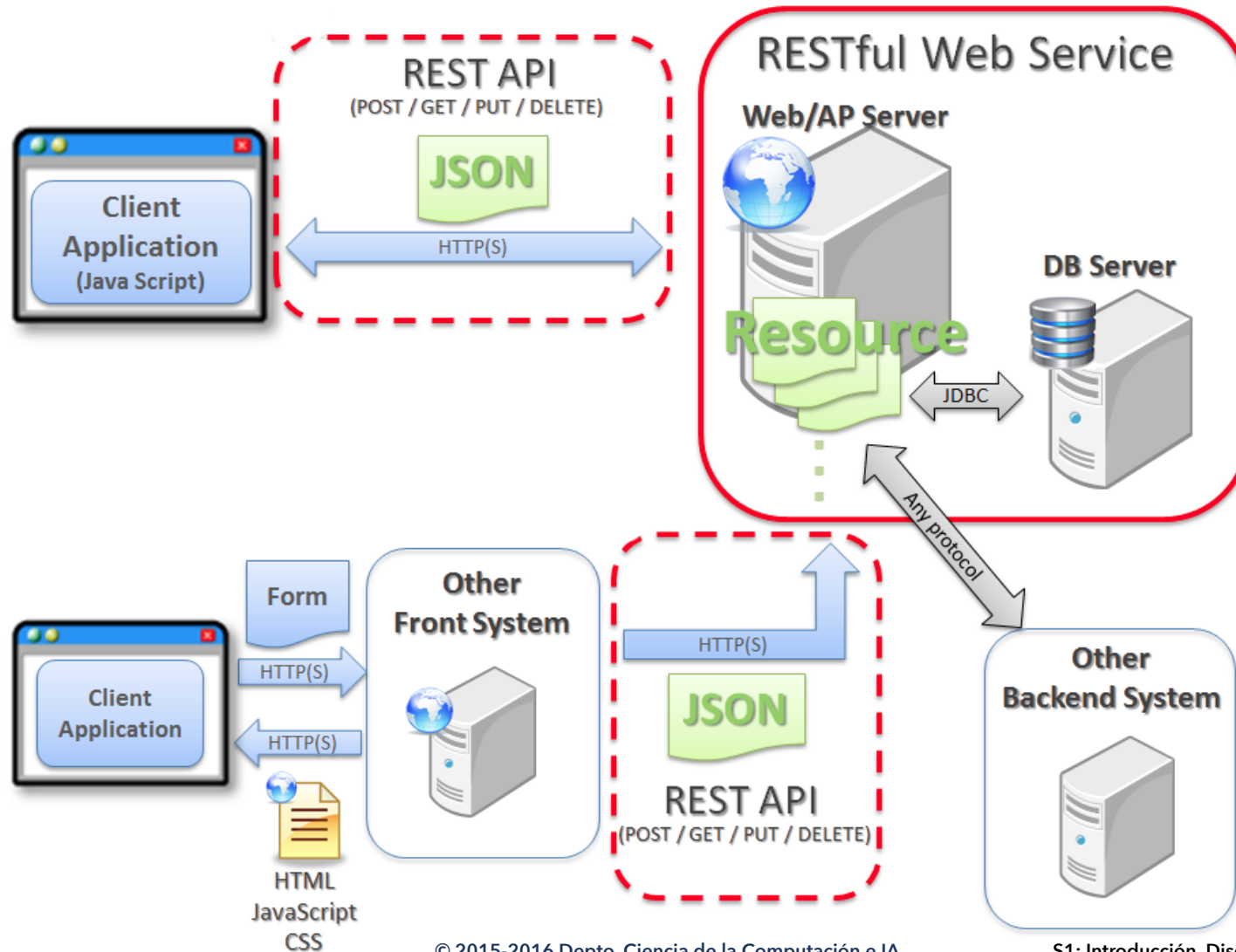
Servicios web "Big" vs. "Light"

- **Big Web Services (SOAP):**
 - Cliente y servidor **intercambian mensajes en XML** siguiendo el formato SOAP (Simple Object Access Protocol)
 - Existe una **descripción formal del servicio** escrita en lenguaje WSDL (Web Services Description Language)
- **Light Web Services (RESTful)**
 - Cliente y servidor pueden intercambiar información en distintos formatos, no está especificado
 - No es necesario definir formalmente un protocolo para el servicio (se utiliza fundamentalmente HTTP)





¿Dónde encajan los servicios web en la aplicación?





Índice

- Servicios web
- **Fundamentos de REST**
- Diseño de servicios web RESTful
- Implementación de un primer servicio JAX-RS



¿Qué es REST?

*“Un Estilo de **arquitectura** software consistente en un conjunto de **restricciones** arquitectónicas que se aplican sobre ciertos elementos, dentro de un sistema distribuido hipermedia [...] . en la actualidad se usa en el sentido más amplio para describir cualquier interfaz web simple que utiliza **XML** y **HTTP**”*

Wikipedia

- **No** es una **tecnología concreta**
- **No** es un **protocolo**
- **No** es una **metodología de diseño**
- **No** es una religión (... pero casi)

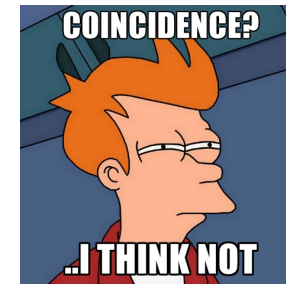
“El resultado de cumplir con dichas restricciones, posibilita que cualquier sistema hipermedia distribuido posea las siguientes propiedades no funcionales: rendimiento, escalabilidad, simplicidad, mantenibilidad, visibilidad, portabilidad y fiabilidad.

Fielding, Roy Thomas Dissertation (2000)



Los orígenes de REST

- Tesis doctoral de **Roy Fielding**, coautor de la especificación de HTTP
- **Formalmente**, un sistema es REST si sigue estos principios:
 - Es **cliente-servidor**
 - **No mantiene estado**
 - Soporta **caches**
 - **Interfaz uniforme**
 - Basado en recursos
 - Orientado a representaciones
 - Interfaz restringida a un pequeño conjunto de métodos bien definidos
 - Hipermedia como "máquina de estados" de la aplicación (HATEOAS)
 - Sistema por **capas** (soporte de *proxies*)





Recursos

- Un **recurso REST** es cualquier cosa que sea **direccionable** a través de la Web. Por direccionable nos referimos a recursos que puedan ser **accedidos y transferidos** entre clientes y servidores
- Ejemplos
 - Una noticia de un periódico
 - La temperatura de Alicante a las 4:00pm
 - Un valor de IVA almacenado en una base de datos
 - Una lista con el historial de las revisiones de código en un sistema CVS
 - Un estudiante en alguna aula de alguna universidad
 - El resultado de una búsqueda de un ítem particular en Google



Representación de los recursos

- Los **datos** que se intercambian entre los servidores y clientes.
- Clientes diferentes son capaces de consumir diferentes representaciones del mismo recurso. Un recurso puede tener **varias representaciones** (una imagen, un texto, un fichero XML, o un fichero JSON), pero tienen que estar disponibles **en la misma URI**.
 - Luego veremos cómo indicar qué representación queremos consultar
- REST es “agnóstico” en cuanto a **la representación, queda a elección del desarrollador**.



Direccionabilidad de los recursos

- Una **URI (Uniform Resource Identifier)** identifica a un recurso
- En servicios Web las URIs son **hiperenlaces**. El formato de una URI es el siguiente:
 - `scheme://host:port/path?queryString#fragment`
- La URI de un recurso **no debe cambiar a lo largo del tiempo** aunque cambie la implementación subyacente

URI

```
http://expertojava.ua.es/recursos/clientes
```

Representación

```
<?xml version="1.0"?>
<clientes>
  <cliente>http://expertojava.ua.es/recursos/cliente/1"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/2"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/4"<cliente/>
  <cliente>http://expertojava.ua.es/recursos/cliente/6"<cliente/>
</clientes>
```



Interfaz uniforme

- Solo se permite un **conjunto reducido de operaciones** sobre los recursos
 - Ejemplo: no implementaríamos una operación `listarUsuarios` sino que tomaríamos una operación genérica `listar` y la aplicaríamos al recurso `Usuario`
- En **servicios web** las **operaciones** permitidas se corresponden con los **métodos HTTP**: GET/POST/PUT/DELETE
- Podemos considerar estos métodos como **operaciones CRUD** sobre los recursos

Acción sobre los datos	Protocolo HTTP equivalente
CREATE	POST
RETRIEVE	GET
UPDATE	PUT
DELETE	DELETE



Aplicaciones web tradicionales vs. REST

- En las aplicaciones **tradicionales** las operaciones se representan con **verbos** en las URIs. Al método HTTP no se le da “excesiva importancia” (solo se suele usar **GET/POST**, el segundo asociado habitualmente a un formulario con muchos datos)
 - Petición **GET** a `http://miapp.com/listarUsuarios?orden=asc`
 - Petición **GET** a `http://miapp.com/modificarUsuario?localidad=Alicante`
 - Petición **POST** (con los datos del nuevo usuario) a `http://miapp.com/crearUsuario`
- En **REST** la **URI representa el recurso (nombre, no verbo)** y el **método HTTP la operación**
 - Petición **GET** a `http://miapp.com/usuarios?orden=asc`
 - Petición **PUT** (con los nuevos datos para el usuario) a `http://miapp.com/usuario/1`
 - Petición **POST** (con los datos del nuevo usuario) a `http://miapp.com/usuarios`



GET

- Operación **solo de lectura**. Para recuperar información
- Se utiliza la cabecera **Accept** para indicar el tipo de representación solicitado
- Típicamente el **cuerpo** de la petición estará **vacío**
- **Idempotente**: no importa si se ejecuta 1 o 1000 veces, el resultado es el mismo
- **Segura**: no tiene efectos laterales

- Ejemplos:
 - Petición **GET** a `http://miapp.com/usuarios?orden=asc` (*Devuelve un conjunto de usuarios*)
 - Petición **GET** a `http://miapp.com/usuario/1` (*Devuelve un usuario sabiendo su id*)



PUT

- Representa la **actualización** de un recurso ya existente con los datos contenidos en el cuerpo de la petición
- Se utiliza la cabecera **Content-Type** para indicar el tipo de representación del recurso del cuerpo de la petición
- Cuando se usa PUT, **el cliente conoce el id del recurso** sobre el que está actuando
- **Idempotente**

- Ejemplo:
 - Petición **PUT** a `http://miapp.com/usuario/1` (*modifica un usuario con los datos contenidos en el cuerpo de la petición*)



DELETE

- Representa el **borrado** de un recurso existente
- Cuando se usa DELETE, **el cliente conoce el id del recurso** sobre el que está actuando
- Típicamente el **cuerpo** de la petición está **vacío**
- **Idempotente**

- Ejemplo:
 - Petición **DELETE** a `http://miapp.com/usuario/1` (*elimina el usuario con los datos contenidos en la URI de la petición*)



POST

- Representa la **inserción** de un nuevo recurso con los datos contenidos en el cuerpo de la petición
- Se utiliza la cabecera **Content-Type** para indicar el tipo de representación del recurso del cuerpo de la petición
- Se utiliza la cabecera **Accept** para indicar el tipo de representación solicitado en el cuerpo de la respuesta
- Cuando se usa POST, **el cliente no conoce el id del recurso** ya que este no se ha creado todavía
- **No Idempotente ni segura**
- Ejemplo:
 - Petición **POST** a `http://miapp.com/usuarios/` (*crea un nuevo usuario con los datos contenidos en el cuerpo de la petición*)



Otros métodos HTTP

- **HEAD**

- Idéntico a **GET**, pero devuelve una **respuesta con cuerpo vacío**, solo devuelve código de estado y cabeceras HTTP. Útil por ejemplo si queremos comprobar simplemente si un recurso existe, pero no nos interesa su contenido

- **OPTIONS**

- Solicitar información al servidor sobre los **métodos HTTP disponibles para un recurso** en el que estamos interesados. Por ejemplo el servidor nos podría indicar que el recurso no se puede borrar

```
200 OK
Allow: HEAD,GET,PUT,OPTIONS
```

Posible respuesta de un servidor a una petición OPTIONS



Índice

- Servicios web
- Fundamentos de REST
- **Diseño de servicios web RESTful**
- Implementación de un primer servicio JAX-RS



Diseño de servicios web RESTful

1. **Elicitación de requerimientos y creación del modelo de objetos:** similar al diseño orientado a objetos. El resultado puede ser un modelo de clases UML. Obtenemos los “recursos”
2. **Definición de las URLs asociadas a los recursos**
3. **Definición de la representación de los recursos:** formato de los datos que utilizaremos para intercambiar información entre nuestros servicios y clientes
4. **Definición de los métodos de acceso a los recursos:** qué métodos HTTP nos permitirán acceder a las URLs que queremos exponer, así como qué hará cada método.

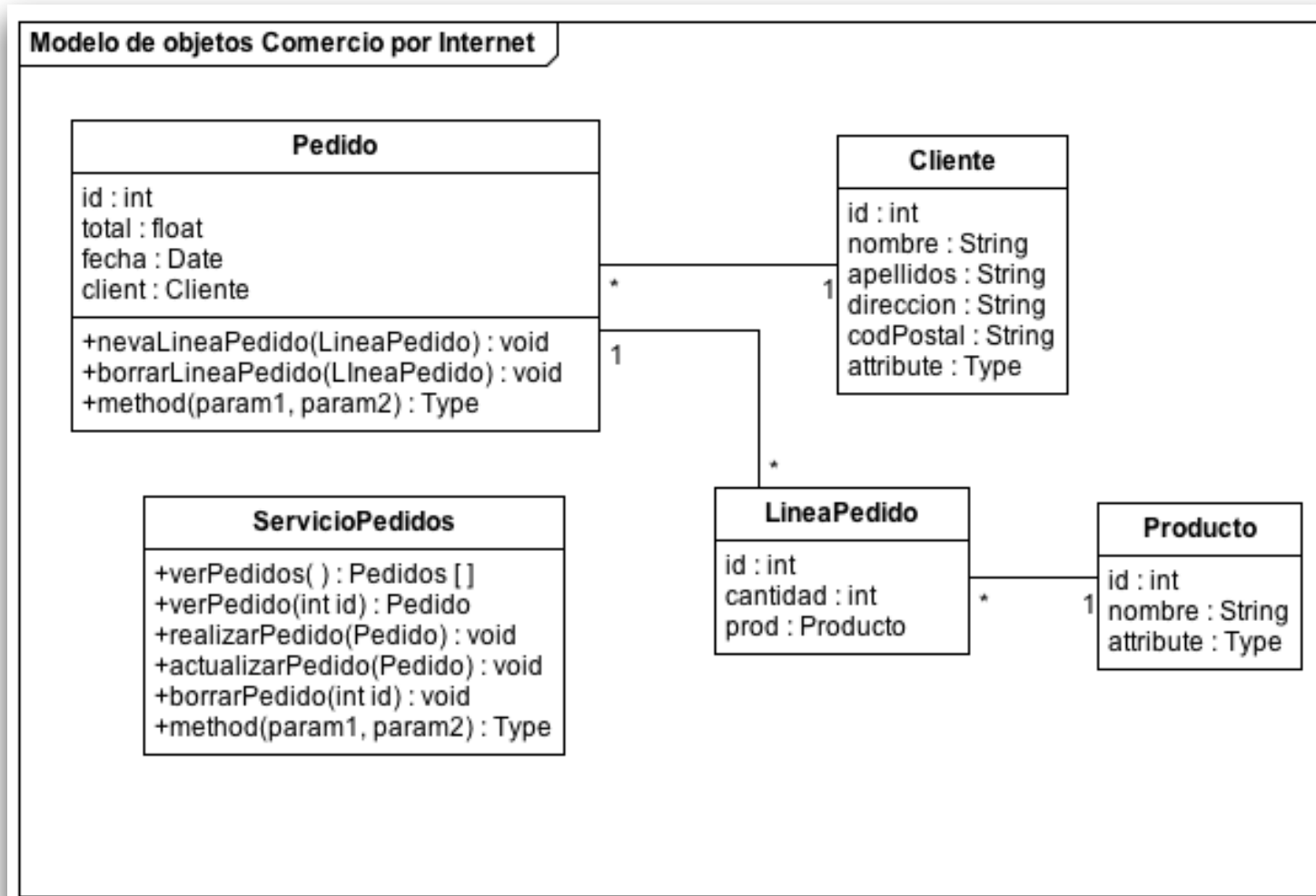


Ejemplo: un primer servicio con JAX-RS

Queremos definir una interfaz RESTful para un sistema sencillo de gestión de pedidos de un hipotético comercio por internet. Los potenciales **clientes** de nuestro sistema, podrán *realizar compras*, *modificar pedidos* existentes en nuestro sistema, así como *visualizar* sus **datos personales** o la información sobre los **productos** que son ofertados por el comercio



Paso 1: Modelo de objetos





Paso 2: Modelado de URIs

- **Recursos:** pedidos, clientes, productos
- **Subrecursos:** lineaPedido (*por el momento vamos a ignorar los subrecursos hasta la siguiente sesión*)

- **Posibles URIs:**
 - /pedidos
 - /pedidos/{id}
 - /productos
 - /productos/{id}
 - /clientes
 - /clientes/{id}



Paso 3: Definición del formato de datos

- Como ya hemos dicho en REST en principio el formato es libre
- Formatos típicos son XML y JSON
- Ejemplo de posibles representaciones para un cliente en formatos XML y JSON

```
<cliente id="8">
  <link rel="self"
        href="http://org.expertojava/clientes/8"/>
  <nombre>Pedro</nombre>
  <apellidos>Garcia Perez</apellidos>
  <direccion>Calle del Pino, 5</direccion>
  <codPostal>08888</codPostal>
  <ciudad>Madrid</ciudad>
</cliente>
```

XML

```
{ "cliente": {
  "id": "8",
  "link": {
    "rel": "self",
    "href": "http://org.expertojava/clientes/8"
  },
  "nombre": "Pedro",
  "apellidos": "Garcia Perez",
  "direccion": "Calle del Pino, 5",
  "codPostal": "08888",
  "ciudad": "Madrid"
}
```

JSON



Paso 4: Asignación de métodos HTTP

- **Listar recursos** de un tipo (por ejemplo productos) `GET /productos`
- Listar **solo algunos recursos** de un tipo (para no tener que obtenerlos todos, si hay demasiados) `GET /productos?startIndex=0&size=5`
- **Obtener un único recurso** de un tipo (conociendo su id) `GET /pedidos/233`
- **Crear un recurso** (el recurso se envía en el cuerpo de la petición) `POST /pedidos`
- **Actualizar un recurso** (enviamos el recurso en el cuerpo de la petición) `PUT /pedidos/233`
- **Eliminar un recurso** `DELETE /pedidos/233`
- **Otras operaciones que no encajan tan claramente en un CRUD**
 - Supongamos que podemos **cancelar un pedido** pero esta operación no lo borra permanentemente, solo lo cancela "temporalmente" y podríamos volver a activarlo
 - Si consideramos que el estar cancelado es parte del estado del pedido, podemos modelar la operación como un PUT que fije el estado a "cancelado"



Paso 4: Asignación de métodos HTTP (ejemplo GET)

- **Obtener un único recurso** de un tipo (conociendo su id)
 - Si no existiera un recurso con dicho id, deberíamos devolver un código de estado 404

```
GET /pedidos/233 HTTP/1.1
```

Petición

```
HTTP/1.1 200 OK  
Content-Type: application/xml  
  
<pedido id="233">...</pedido>
```

Respuesta



Paso 4: Asignación de métodos HTTP (ejemplo POST)

- **Crear un recurso**
 - Nótese la **cabecera Content-Type**, que ahora va en la petición
 - El servidor debe devolver un **código 201** en caso de creación OK
 - Por convenio, se usa la cabecera **Location** en la respuesta para indicar la URI del nuevo recurso

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

```
<pedido>
  <total>199.02</total>
  <fecha>December 22, 2008 06:56</fecha>
  ...
</pedido>
```

Petición

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

```
<pedido id="233">
  <link rel="self" href="http://org.expertojava/pedidos/233"/>
  <total>199.02</total>
  <fecha>December 22, 2008 06:56</fecha>
  ...
</pedido>
```

Respuesta



Paso 4: Asignación de métodos HTTP (ejemplo PUT)

- Actualizar un recurso
 - Nótese que conocemos el id del recurso
 - El servidor debería devolver un **código 200 o 204** en caso de actualización OK

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

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

Petición

```
HTTP/1.1 200 OK
```

Respuesta



Paso 4: Asignación de métodos HTTP (ejemplo DELETE)

- **Eliminar un recurso**

- Nótese que conocemos el id del recurso
- El servidor debería devolver un **código 200 o 204** en caso de actualización OK

```
DELETE /pedidos/233 HTTP/1.1
```

Petición

```
HTTP/1.1 200 OK
```

Respuesta



Índice

- Servicios web
- Fundamentos de REST
- Diseño de servicios web RESTful
- **Implementación de un primer servicio JAX-RS**
 - Clases del dominio
 - Implementación del/los recursos
 - Configuración del servlet JAX-RS
 - Despliegue de la aplicación
 - Utilizamos un cliente para probar los servicios



JAX-RS

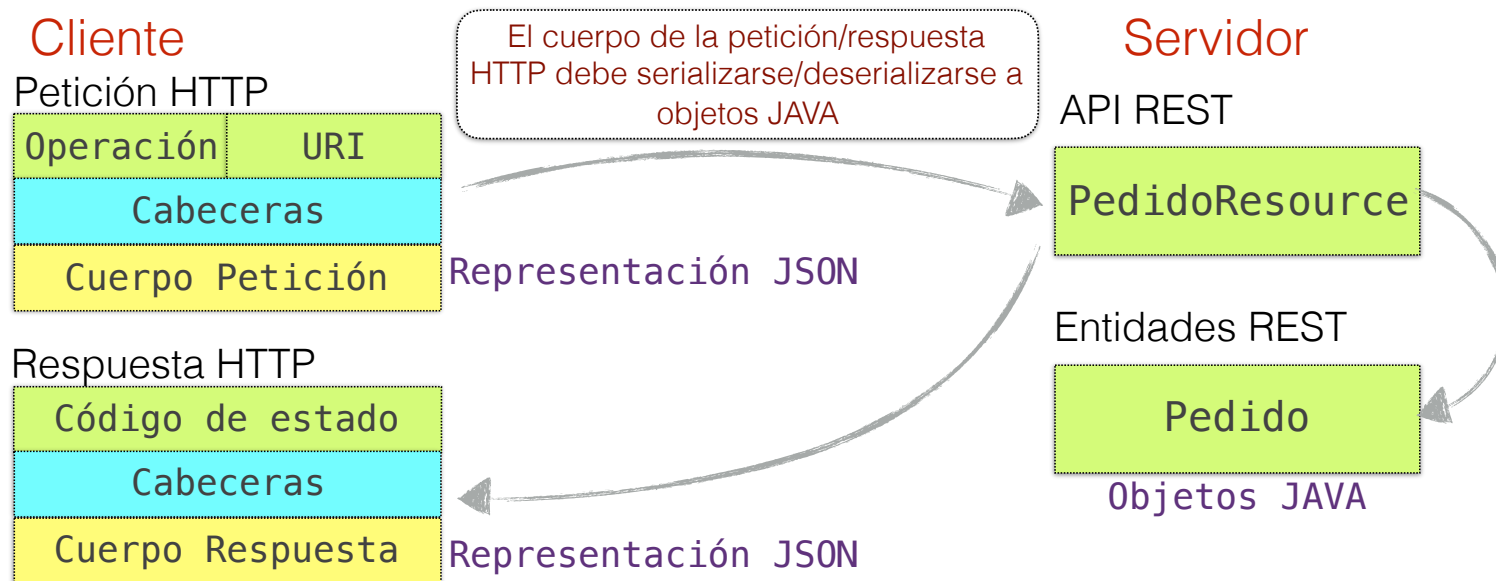
- API de Java para servicios REST (JSR 339)
- **Otros APIs útiles: JAXB** (Java API for XML Binding, JSR 31) para anotar clases Java y serializar/deserializar automáticamente a XML o JSON, **JSON processing** (JSR 353)
- **Implementaciones**
 - **Jersey**, implementación de referencia
 - **RestEASY**, incluida en WildFly





Clases de nuestra aplicación

- Hay que diferenciar entre:
 - **Entidades** del dominio (Usuario, Pedido, Cliente, ...)
 - **Recursos**: clases Java con anotaciones de JAX-RS que representan los recursos REST (UsuarioResource, PedidoResource,...)
- Típicamente tenemos un recurso por cada entidad del dominio, aunque no necesariamente
 - En la siguiente sesión veremos el uso de subrecursos





Clases del dominio

- Podríamos tener también anotaciones JPA

```
package org.expertojava;

@XmlRootElement(name="cliente")
@XmlAccessorType(XmlAccessType.FIELD)
public class Cliente {
    private int id;
    private String nombre;
    private String apellidos;
    private String direccion;
    private String codPostal;
    private String ciudad;

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

    //resto de getters y setters
    ...
}
```

Estas anotaciones "**serializan**" la clase **Cliente** en formato **XML (o JSON)**. Hablaremos de ellas más adelante



Recursos JAX-RS

- No es necesario heredar de ninguna clase ni implementar ninguna interfaz
- La anotación `@Path` **asocia el recurso con una trayectoria** tomada desde la raíz del servicio (luego veremos cómo se establece esta raíz)
 - Si la raíz fuera `http://expertojava.io` el recurso estaría en `http://expertojava.io/clientes`

```
package org.expertojava;

import ...;

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

    ...

}
```

JAX-RS identifica esta clase como un recurso raíz



Consideración importante sobre los recursos REST

- Por defecto, los recursos REST NO guardan el estado (siguen un modelo de peticiones *per-request*, lo explicaremos en la siguiente sesión).
 - El *runtime* de JAX-RS genera una nueva instancia de nuestro recurso REST para CADA petición
 - Por lo tanto, NO debemos utilizar variables locales de instancia en nuestros recursos, y confiar en que la siguiente petición al recurso puede acceder al estado de la petición anterior. Nuestro recurso atenderá peticiones concurrentes, cada una en un *thread* diferente
 - Una forma de compartir el estado del recurso entre peticiones es utilizar una variable estática (se comparte por todas las instancias de la clase) y "protegerla" para accesos concurrentes

```
package org.expertojava;

import ...;

@Path("/clientes")
public class ClienteResource {
    private static Map<Integer, Cliente> clienteDB = new ConcurrentHashMap<Integer, Cliente>();
    private static AtomicInteger idContador = new AtomicInteger();...
    ...
}
```



Responder a las peticiones HTTP

- Anotamos los métodos de la clase recurso con @GET, @PUT, @POST, @DELETE,...
- La anotación @Produces genera una cabecera Content-type
- La anotación @PathParam "extrae" un parámetro de URL de la petición HTTP
 - Petición GET sobre <http://expertojava.io/clientes/2>

Método que atiende peticiones GET

```
@GET
@Path("/{id}")
@Produces("application/xml")
public Cliente recuperarCliente(@PathParam("id") int id) {
    //obtenemos los clientes
    final Cliente cli = clienteDB.getAll();
    if (cli == null) {
        throw new WebApplicationException(Response.Status.NOT_FOUND);
    }
    //devolvemos el resultado
    return cli;
}
```



Responder a una petición POST

- Debemos **deserializar el cuerpo de la petición** (ya veremos los detalles más adelante)
 - Con `@Consumes` indicamos el formato esperado. Si el cliente no nos indica con `Content-Type` que nos está enviando este formato se generará automáticamente un *status 415 Unsupported Media Type*
- Debemos **devolver la cabecera Location** con la URI del recurso recién creado

```
@POST
@Consumes("application/xml")
public Response crearCliente(Cliente cli) {
    //el parámetro cli se instancia con los datos del cliente del body del mensaje HTTP
    idContador++;
    cli.setId(idContador);
    clienteDB.put(cli.getId(), cliente);
    System.out.println("Cliente creado " + cli.getId());

    return Response.created(URI.create("/clientes/"
        + cli.getId())).build();
}
```

Solamente **UNO** de los parámetros del método Java puede representar el cuerpo del mensaje de la petición HTTP (**el que NO lleve anotaciones JAX-RS**)



“Activar” el servicio JAX-RS

- Una forma (veremos otras) es definir en el `web.xml` un *servlet* de la clase `javax.ws.rs.core.Application` asociado a la URL que queremos que sea la raíz del servicio
- Al arrancar, el *servlet* examina automáticamente todas las clases en busca de anotaciones JAX-RS

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



Despliegue de la aplicación (en WildFly)

- Incluiremos en el **pom.xml** un plugin para desplegar nuestra aplicación (de tipo **.war**) en Wildfly

```
<!-- Servidor de aplicaciones wildfly -->
<plugin>
  <groupId>org.wildfly.plugins</groupId>
  <artifactId>wildfly-maven-plugin</artifactId>
  <version>1.0.2.Final</version>
  <configuration>
    <hostname>localhost</hostname>
    <port>9990</port>
  </configuration>
</plugin>
```

La secuencia correcta de acciones (con Maven) para asegurarnos de que en el servidor tenemos los últimos cambios de nuestra aplicación es (suponiendo que el servidor Wildfly ESTÁ EN MARCHA) son:

- `mvn wildfly:undeploy`
- `mvn clean`
- `mvn package`
- `mvn wildfly:deploy`



Uso de clientes para probar el servicio

- Podemos probar nuestro servicio desde línea de comandos (con **Curl**), o desde **IntelliJ** (con la utilidad Tools->Test RESTful Web Service)
- O podemos utilizar **Postman** (una extensión de Chrome). Esta herramienta es más flexible que las anteriores

Search

Builder Runner Import

http://localhost:8080/ejemplo-rest/rest/clientes/1

GET `http://localhost:8080/ejemplo-rest/rest/clientes/1` Params Send

Authorization Headers (1) Body Pre-request script Tests

Accept application/xml

Header Value

Presets PETICIÓN

Body Cookies Headers (6) Tests Status 200 OK Time 39 ms RESPUESTA

Pretty Raw Preview XML

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <cliente>
3   <id>1</id>
4   <nombre>Pepe </nombre>
5   <apellidos>Ape1 Ape2 </apellidos>
6   <direccion>Calle numero 1 </direccion>
7   <codPostal>0001</codPostal>
8   <ciudad>Alicante</ciudad>
9 </cliente>
```

Cuerpo de la respuesta de la petición HTTP



Postman nos permite gestionar Colecciones

- También podemos crear "colecciones" (son carpetas que contienen un conjunto de peticiones de nuestro "historial")
- Podéis crearos una cuenta gratuita para almacenar y gestionar vuestras peticiones rest
- Las colecciones pueden descargarse en nuestro disco duro en formato json
- URLs de servicios REST que podéis probar:
 - <http://pokeapi.co>, <http://www.predic8.com/rest-demo.htm>

Search

History Collections

estadoResource
16 Oct at 2:52 pm • 5 requests

sesion1
7 Oct at 4:19 pm • 4 requests

Detalles de la colección

Edit
Add Folder
Share
Duplicate
Download
Delete

Details

Run

Podemos descargar desde aquí

sesion1
Modified Wednesday October 7, 2015 4:19:27pm

Peticiones para la aplicacion ejemplo-rest de la sesion 1

POST http://localhost:8080/ejemplo-rest/rest/productos

POST http://localhost:8080/ejemplo-rest/rest/productos

GET http://localhost:8080/ejemplo-rest/rest/productos

GET http://localhost:8080/ejemplo-rest/rest/productos...

También podemos descargar desde aquí



¿Alguna duda, pregunta...?

