

4.5 APIs Java y Java EE para Servicios Web

4.5.1 JAXB (Java Architecture for XML Binding)

APIs "clásicas" para procesamiento de documentos XML

- SAX (*Simple API for XML*): procesamiento de XML basado en eventos (etq. de apertura, etq. de cierre, contenido)
- DOM (*Document Object Model*): representación del documento XML como un árbol + interfaz estándar para recorrer el árbol

JAXB ofrece un acceso a XML a más alto nivel que SAX y DOM

- API + conjunto de anotaciones para representar documentos XML
- Gestiona el mapeo entre documentos XML y objetos Java (análogo a JPA)
- marshaling : transformación de objetos Java en elementos de un doc. XML (~ serialización en XML)
- unmarshaling: transformación de un doc. XML a un árbol de objetos Java (~ deserialización en XML)
- Permite que las APIs de servicios web de Java no "trabajen" directamente con XML

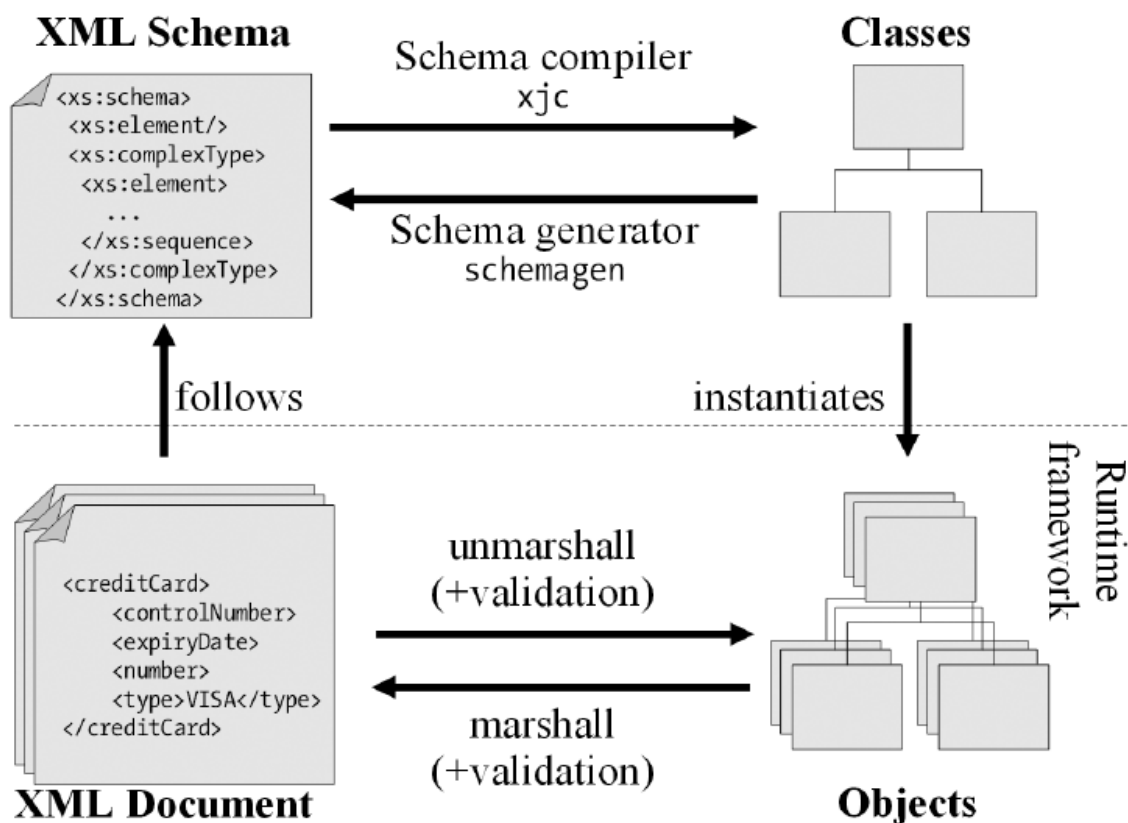


Figure 14-3. JAXB architecture

Herramientas de línea de comandos

- **xjc**: compilador de esquemas XML
 - Genera un conjunto de clases que representan los documentos XML cuya estructura está descrita en un fichero XSD (*XML Schema Definition*)
- **schemagen**: generador de esquemas XML
 - Genera un fichero XSD a partir de un paquete de clase Java que define la representación XML de las mismas.

@XmlRootElement

```
public class CreditCard {  
  
    private String number;  
    private String expiryDate;  
    private Integer controlNumber;  
    private String type;  
  
    // Constructors, getters, setters  
}  
  
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<creditCard>  
    <controlNumber>6398</controlNumber>  
    <expiryDate>12/09</expiryDate>  
    <number>1234</number>  
    <type>Visa</type>  
</creditCard>  
  
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>  
<xs:schema version="1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema">  
<xs:element name="creditCard" type="creditCard"/>  
<xs:complexType name="creditCard">  
    <xs:sequence>  
        <xs:element name="controlNumber" type="xs:int" minOccurs="0"/>  
        <xs:element name="expiryDate" type="xs:string" minOccurs="0"/>  
        <xs:element name="number" type="xs:string" minOccurs="0"/>  
        <xs:element name="type" type="xs:string" minOccurs="0"/>  
    </xs:sequence>  
</xs:complexType>  
</xs:schema>
```

Clase ***javax.xml.bind.JAXBContext*** gestiona mapeo entre docs. XML y objetos Java.

- Responsable de proporcionar los objetos *Marshaller* y *Unmarshaller*
- Las clases *Unmarshaller* transforman un doc. XML en un grafo de objetos Java (opcionalmente valida el doc. contra su esquema XSD)

El origen de datos puede ser un *InputStream*, un *String*, un *Node* de DOM, ...

- Las clases *Marshaller* transforman un grafo de objetos Java en un doc. XML

```
public class Main {  
  
    public static void main(String[] args) {  
  
        CreditCard creditCard = new CreditCard("1234", "12/09", 6398, "Visa");  
        StringWriter writer = new StringWriter();  
  
        JAXBContext context = JAXBContext.newInstance(CreditCard.class);  
        Marshaller m = context.createMarshaller();  
        m.marshal(creditCard, writer);  
  
        System.out.println(writer.toString());  
    }  
}
```

Anotaciones JAXB

Controlan como se realiza el mapeo entre los elementos del doc. XML y los objetos Java

- Paquete *javax.xml.bind.annotation*
- Como mínimo la clase que actúe como raíz del documento XML deberá anotarse con *@XmlRootElement*
 - Debe ser un *JavaBean* [constructor vacío + acceso a atributos con *get()* y *set()*]
 - Por defecto todos y cada uno de sus atributos públicos que no sean *static* o *transient* y todos y cada uno de sus métodos *getXXX()* públicos se transformarán en un elemento XML cuya etiqueta coincide con el nombre del atributo.
 - Con *@XmlAccessorType* se puede configurar qué tipo de elementos se serializarán en XML por defecto:
 - *XmlAccessType.FIELD* (sólo atributos públicos)
 - *XmlAccessType.NONE* (sólo elementos marcados con *@XmlElement* o *@XmlAttribute*)
 - *XmlAccessType.PROPERTY* (sólo atributos con un *getXXX()* público)
 - *XmlAccessType.PUBLIC_MEMBER* (comportamiento por defecto)
 - Los tipos básicos de Java se convierten en elementos XML simples
 - Los objetos se convierten en elementos XML compuestos
 - Los arrays, listas y tablas hash se convierten conjuntos de elementos XML
- La conversión por defecto puede configurarse mediante anotaciones JAXB (útiles para trabajar con WSDL o XSD definidos previamente)
 - Se anotan los atributos o los métodos *get()*.
 - *@XmlAttribute*: indica que un atributo del objeto Java se mapeará como un atributo XML
 - *@XmlElement*: indica que un atributo del objeto Java se mapeará como un elemento XML (se usa para especificar un nombre de elemento XML diferente del asignado por defecto)
 - *@XmlElements*: actúa como un contenedor de múltiples anotaciones *@XmlElement*
 - *@XmlList*: especifica cómo se mapea un atributo de tipo *List* o *Collection*
 - *@XmlEnum*: especifica el mapeo de un *Enum* de Java
 - *@XmlTransient*: informa a JAXB de un atributo que no debe ser serializado a XML
 - *@XmlID*: identifica un atributo que se podrá usar para referenciar el elemento XML desde otros elementos anotados con *@XmlIDREF*
 - *@XmlIDREF*: mapea un atributo del objeto como referencia a un ID de XML

Cada anotación cuenta con sus propios atributos para configurar detalles del mapeo (orden en que serializar los descendientes, etc)

- Anotaciones sobre clases y paquetes
 - *@XmlRootElement*: anota la clase que operará como raíz del doc. XML
 - *@XmlAccessorType*: especifica la serialización por defecto de la clase.
 - *@XmlType*: anota una clase como un tipo compuesta en XSD
 - *@XmlSchema*: vincula un paquete Java (package) a un namespace XML

4.5.2 JAX-WS (Java API for XML-Based Web Services)

API de Java EE para la publicación y acceso a servicios web basados en WSDL y SOAP

- La implementación por defecto (proyecto Metro) se incluye también en Java SE 6 (jdk 1.6)

La gestión de los documentos XML incluidos en las peticiones y respuestas SOAP se delega en JAXB

JAX-WS define su propio conjunto de anotaciones

- para definir las clases y métodos que actúan como puntos finales de los mensajes que conforman las invocaciones SOAP
- para especificar la definición del fichero WSDL y del binding SOAP

La implementación del servicio web puede desplegarse empleando un Servlet (servlet endpoint) o un EJB (EJB endpoint)

- En contenedores que soporten Java EE 6 el despliegue es automático en ambos casos [al iniciar la aplicación el contenedor inspecciona las clases en busca de las notaciones `@WebService` y establece los mapeos de URL pertinentes]
- En contenedores no Java EE 6, debe configurarse en el descriptor de despliegue de la aplicación (*WEB-INF/web.xml*) el servlet de "escucha" del API JAX-WS

Herramientas de línea de comandos incluidas

wsimport: genera, a partir de un doc. WSDL, los "artefactos" Java necesarios para invocar un servicio web

- Clases Java con anotaciones JAXB que mapean los datos XML intercambiados en los mensajes SOAP
- Clases e interfaces Java que implementan los *proxy* (*stub*) de los elementos *service* y *portType* definidos en WSDL
 - Son los objetos Java que reciben las llamadas de los clientes y gestionan los mensajes SOAP que implementan el diálogo con el servidor
 - Denominados *Service Endpoint Interface* (SEI) [representación Java (interfaz) del servicio web]

wsgen: genera el doc. WSDL (y opcionalmente el esquema XSD) a partir de la clase de implementación (*endpoint*) del servicio web (en función de las anotaciones JAX-WS incluidas)

Definición de servicios web con JAX-WS

- El único requisito es contar con un interfaz y/o una clase de implementación anotado con `@WebService`
 - En el caso de EJB *endpoints*, además deben de estar anotados como `@Stateless` (los servicios web son sin estado)
- La clase de implementación debe ser pública y no puede ser *final* ni *abstract*
- La clase de implementación debe contar con un constructor vacío
- La clase de implementación no puede definir un método *finalize()*
- Debe garantizarse una implementación sin estado
 - La clase de implementación no puede guardar info. de estado entre llamadas del cliente

- Por defecto, para la clase/interface de implementación:
 - se generará un elemento WSDL *service* con el mismo nombre de la clase y el sufijo *Service*
 - se generará un elemento WSDL *portType* con el nombre de la clase
- Para cada método público de la clase se generará:
 - un elemento WSDL *operation* con el mismo nombre del método
 - dos elementos WSDL *message*: uno para la petición (con el nombre del método) y otro para la respuesta (añadiendo al nombre del método el sufijo *response*)
 - los parámetros y valores de retorno deben de ser tipos básicos Java, clases anotadas con JAXB o *arrays*, *Map*, *List* o *Collection* de los anteriores

Anotaciones JAX-WS

Anotaciones que definen el mapeo WSDL (modifican el comportamiento por defecto):

- **@WebService**: señala una clase o interfaz como *endpoint* de un servicio web
 - incluye atributos para modificar el nombre del elemento service, portType, el name space, etc (*name*, *targetNamespace*, *serviceName*, *portName*, *wsdlLocation*, *endpointInterface*)
- **@WebMethod**: permite modificar la definición de las operaciones WSDL (atributo *operationName*) o excluir métodos de la clase que no se desena exponer como operaciones del web service (con el atributo *exclude=true*)
- **@WebResult**: permite controlar el nombre del elemento message de WSDL que contendrá el valor de retorno (atributo *name*)
- **@WebParam**: permite configurar los elementos parameter de WSDL vinculados a los parámetros de una operación (atributos: *name*, *mode* [*IN*, *OU*, *INOUT*], *targetNamespace*, *header*, *partName*)
- **@OneWay**: permite indicar que un método no tendrá valor de retorno

Anotaciones que definen el binding SOAP de las operaciones/métodos

- **@SOAPBinding**: para un método de la clase endpoint especifica el estilo de codificación de los mensajes (RPC vs. document) y el tipo de codificación de los parámetros a usar (encoded vs.literal).
Atributos: *style*, *use*, *parameterStyle*.
- **@SOAPMessageHandler**: especifica detalles de la gestión de los mensajes (petición y respuesta)
Atributos: *name*, *className*, *initParams*, *roles*, *headers*

```

@WebService(name = "CreditCardValidator", portName = "ValidatorPort")
public class CardValidator {

    @WebMethod(operationName = "ValidateCreditCard")
    @WebResult(name = "IsValid")
    public boolean validate(
        @WebParam(name = "CreditCard") CreditCard creditCard) {

        String lastDigit = creditCard.getNumber().substring(
            creditCard.getNumber().length() - 1,
            creditCard.getNumber().length());
        if (Integer.parseInt(lastDigit) % 2 != 0) {
            return true;
        } else {
            return false;
        }
    }

    @WebMethod(exclude = true)
    public void validate(String ccNumber) {
        // business logic
    }
}

```

Invocación de servicios web con JAX-WS

En entornos de "objetos gestionados" (contenedor de Servlets, contenedor de EJBs, contenedor de clientes JEE) se puede utilizar la anotación `@WebServiceRef` para que el contenedor inyecte una referencia al *Service Endpoint Interface* (SEI) que representa al *endpoint* del servicio web en el cliente.

- Ese SEI se corresponde con un proxy (stub), implementa el mismo interfaz que la clase de implementación y es quien recibe la invocación del cliente y envía/recibe los mensajes SOAP.

```

@WebServiceRef
private CardValidatorService cardValidatorService;
// ...
CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
cardValidator.validate(creditCard);

```

- La anotación `@WebServiceRef` se puede parametrizar con atributos: *name*, *type*, *mappedName*, *value*, *wsdlLocation*.

En el caso de clientes no ejecutados en un contenedor de objetos, las clases generadas por **wsimport** se pueden utilizar directamente.

```

CardValidatorService cardValidatorService = new CardValidatorService();
CardValidator cardValidator = cardValidatorService.getCardValidatorPort();
cardValidator.validate(creditCard);

```

Una vez obtenido el SEI (stub del servicio web) se puede obtener un stub de cada uno de sus *portType* e invocar las operaciones disponibles en cada uno de ellos.

4.6 REST y JAX-RS

4.6.1 REST

REST (*REpresentational State Transfer*): Se define como un "estilo arquitectónico" para el desarrollo de aplicaciones distribuidas

- Forma de construir aplicaciones distribuidas
- Centrada en el concepto de **RECURSO** (~ objeto)
 - mecanismos RPC centrados en concepto de operación
- El estado de los recursos reside en el servidor
- Clientes acceden al estado de recurso mediante **REPRESENTACIONES** del mismo
 - transferidas desde el servidor o desde el cliente empleando el protocolo HTTP como mecanismo de transporte (podrían emplearse otros, REST no es específico de un protocolo)
 - cada aplicación puede utilizar diferentes formatos para la representación de los recursos
 - HTML para navegadores web
 - XML, JSON [*JavaScript Object Notation*] o YAML [*YAML Ain't Markup Language*] para aplicaciones
 - etc
 - con el parámetro *Content-Type* de las cabeceras HTTP se especifica el tipo MIME de los datos intercambiados (text/html, application/xml, application/json, text/yaml, ...)
 - cliente puede informar al servidor del tipo de representación que necesita (parámetro *Accept* en cabecera de las peticiones HTTP)
- Los recursos son identificados y están accesibles mediante uno o varios URI (Uniform Resource Identifier)
protocolo://host:puerto/path?queryString#fragmento

http://localhost:8080/gestionReservas-rest/hoteles/
http://localhost:8080/gestionReservas-rest/hoteles/207/
http://localhost:8080/gestionReservas-rest/hoteles/207/reservas/
http://localhost:8080/gestionReservas-rest/hoteles/207/habitaciones/
http://localhost:8080/gestionReservas-rest/hoteles/207/habitaciones/?fInicio=22/7/2010&fFin=28/7/2010
http://localhost:8080/gestionReservas-rest/hoteles/Santiago/
http://localhost:8080/gestionReservas-rest/reservas/30773/
http://localhost:8080/gestionReservas-rest/clientes/
http://localhost:8080/gestionReservas-rest/clientes/173/

- Se basa en el uso de un conjunto predefinido de operaciones sobre esos URI con una semántica predefinida
Operaciones HTTP

GET	- operación de lectura - pide al servidor una representación del recurso/s apuntado/s por el URI, que le será enviada dentro del cuerpo de la respuesta HTTP - idempotente y "segura" (no modifica el estado del recurso)
-----	---

PUT	- operación de actualización - pide al servidor que actualice el recurso apuntado por el URI empleando los datos del cuerpo de la petición HTTP - idempotente y "no segura"
DELETE	- operación de borrado - pide al servidor que elimine el recurso referenciado por el URI - idempotente y "no segura"
POST	- operación de creación de un recurso [en ocasiones también modificación] - pide al servidor que cree un nuevo recurso a partir de los datos enviados en el cuerpo de la petición HTTP ubicándolo "dentro" de la URI indicada en la petición - no idempotente y "no segura" <u>Convenciones en el uso de POST:</u> - la petición POST se realiza sobre la URI de nivel superior dentro de la cual se creará el nuevo recurso - una vez creado el recurso, el servidor envía un mensaje de respuesta con el código <u>201</u> y el URI del nuevo recurso creado (parámetro de cabecera <i>Location</i>)
Otros:	HEAD, OPTIONS, ...

- Son operaciones sin estado (cada mensaje contiene toda la información necesaria)
- Ejemplos:

Respuesta a GET sobre <http://localhost:8080/gestionReservas-rest/reservas/30773>

```
<reserva id="30773">
  <cliente>http://localhost:8080/gestionReservas-rest/clientes/1033</cliente>
  <tipoHabitacion> http://localhost:8080/gestionReservas-rest/hoteles/207/habitaciones/14</tipoHabitacion>
  <fechaEntrada>22/07/2010</fechaEntrada>
  <fechaSalida>28/07/2010</fechaSalida>
  <ocupantes>2</ocupantes>
  <nombreReserva>Juan Pérez</nombreReserva>
</reserva>
```

Respuesta a GET sobre <http://aplicacion.miempresa.net/clientes/32133/pedidos>

```
<lista-pedidos>
  <pedido id="1321" estado="entregado">
    <cliente>http://aplicacion.miempresa.net/clientes/32133</cliente>
    <fecha>12/3/2010</fecha>
    <lineas-pedido>
      <linea-pedido>
        <cantidad>5</cantidad>
        <producto>http://aplicacion.miempresa.net/productos/1341</producto>
      </linea-pedido>
      <linea-pedido>
        <cantidad>1</cantidad>
        <producto>http://aplicacion.miempresa.net/productos/203</producto>
      </linea-pedido>
      ...
    </lineas-pedido>
  </pedido>
  <pedido id="1451" estado="pendiente">
    <cliente>http://aplicacion.miempresa.net/clientes/32133</cliente>
    <fecha>17/7/2010</fecha>
    <lineas-pedido>
      <linea-pedido>
        <cantidad>1</cantidad>
        <producto>http://aplicacion.miempresa.net/productos/349</producto>
      </linea-pedido>
      ...
    </lineas-pedido>
  </pedido>
  ...
</lista-pedidos>
```


4.6.2 JAX-RS

API de Java que define una infraestructura (clases e interfaces) para implementar una arquitectura REST (paquete `javax.ws.rs`)

- Incluye un conjunto de anotaciones para especificar el mapeo entre las URIs de los recursos y los métodos HTTP con los métodos Java de una clase de implementación (*endpoint*)
- Gestiona automáticamente las representaciones de los recursos intercambiados
 - Emplea JAXB para el tipo MIME `application/xml`
 - Emplea la librería *BadgerFish* para mapeo de XML a JSON en el caso del tipo MIME `application/json`
 - La generación y tratamiento de otros tipos de representaciones debe manejarse manualmente (imágenes, PDF, ...) implementando las interfaces `javax.ws.rs.ext.MessageBodyReader`, `javax.ws.rs.ext.MessageBodyWriter` en una clase anotada con `@jax.ws.rs.ext.Provider`.

Implementación de referencia Jersey: <http://jersey.java.net/>

ANOTACIONES JAX-RS

@Path

La clase de implementación de los recursos REST (*resource class*) debe de señalarse con una anotación `@Path`, especificando el path de alto nivel dentro del que se enmarcan las URIs gestionadas por la clase de implementación (*context root*)

Pueden anotarse clases Java "normales" o EJBs sin estado. En el caso de anotar clases "normales" se deberá especificar en el *web.xml* de la aplicación web el uso del Servlet de JAX-RS.

En los métodos de dicha clase se puede especificar el *path* "parcial" (dentro del path de la clase de implementación) al que se vinculan los métodos Java.

- puede asociarse un nombre a los distintos fragmentos que componen el path [irá señalado entre `{ . . }`] (es usado para recuperar "parámetros de path")
- pueden especificarse los fragmentos de path empleando expresiones regulares

Anotaciones de métodos HTTP

Especifican el método HTTP al que se vinculan los métodos Java anotados

- **@GET:** vincula al método anotado las peticiones HTTP GET dirigidas al path correspondiente a ese método
- **@PUT:** vincula al método anotado las peticiones HTTP PUT dirigidas al path correspondiente a ese método
- **@DELETE:** vincula al método anotado las peticiones HTTP DELETE dirigidas al path correspondiente a ese método
- **@POST:** vincula al método anotado las peticiones HTTP POST dirigidas al path correspondiente a ese método

Especificación del formato de las representaciones

Se puede especificar tanto a nivel de clase de implementación (*resource class*) como a nivel de método concreto los tipos de dato MIME que se envían en las peticiones y/o respuestas.

- **@Produces:** indica los tipos MIME que se pueden generar como respuesta a las peticiones HTTP (será el cliente quien especifique sus preferencia con el parámetro de cabecera *Accept*)
Determina como se serializarán en los mensajes de petición (PUT y POST fundamentalmente) los valores de los parámetros recibidos por métodos de implementación.
- **@Consumes:** indica los tipos MIME que pueden recibirse en las peticiones HTTP (en todas la peticiones HTTP el cliente debe especificar el parámetro de cabecera *Content-Type*).
Determinan como se serializarán en los mensajes de respuesta los valores de retorno de los métodos de implementación.

Mapeo de parámetros de URIs y peticiones HTTP

Anotan los parámetros de las

- **@PathParam:** permite extraer valores de parámetros que forman parte de fragmentos del path de la URI
- **@QueryParam:** permite extraer valores de parámetros incluidos en los *queryString* de las URIs
- **@HeaderParam:** permite extraer valores enviados como parámetros en las cabeceras HTTP de petición
- **@FormParam:** permite extraer valores de los datos enviados en el cuerpo de peticiones POST como pares atributos-valor

Adicionalmente se pueden indicar los valores por defecto de estos parámetros con la anotación **@DefaultValue** y especificar cómo gestionar su codificación en los mensajes HTTP (anotación **@Encoded**)

@Path("ejemploRest")

@Stateless

public class EjemploRest {

 @Context private UriInfo context;

 @EJB HotelDAO hotelDAO;

 @EJB DisponibilidadService disponibilidadService;

 @EJB ReservaDAO reservaDAO;

 public EjemploRest() {

 }

@GET

@Produces({ "application/xml", "application/json" })

@Path("/hoteles")

 public List<Hotel> getHoteles() {

 return hotelDAO.buscarHoteles();

 }

@GET

@Produces({ "application/xml", "application/json" })

@Path("/hoteles/{localidad}")

 public List<Hotel> getHotelesPorLocalidad(

@PathParam("localidad") String localidad) {
 return hotelDAO.buscarHotelesPorLocalidad(localidad);

 }

@GET

@Produces({ "application/xml", "application/json" })

@Path("/hoteles/{id}")

 public Hotel getHotel(**@PathParam("id")** Long id) {

 return hotelDAO.buscarPorId(id);

 }

@PUT

@Consumes({ "application/xml", "application/json" })

@Path("/hoteles/{id}")

 public void actualizarHotel(**@PathParam("id")** Long id, Hotel hotel) {

 hotelDAO.actualizar(hotel);

 }

@DELETE

@Consumes({ "application/xml", "application/json" })

@Path("/hoteles/{id}")

 public void borrarHotel(**@PathParam("id")** Long id) {

 Hotel hotel = hotelDAO.buscarPorId(id);

 hotelDAO.borrar(hotel);

 }

```

@POST
@Consumes({"application/xml"}, {"application/json"})
@Produces({"application/xml"}, {"application/json"})
@Path("/hoteles")
public Response crearHotel(JAXBElement<Hotel> hotelJAXB) {
    Hotel hotel = hotelJAXB.getValue();
    hotel = hotelDAO.crear(hotel);
    URI hotelURI=context.getAbsolutePathBuilder().path(hotel.getId()).build();
    return Response.created(hotelURI).build();
}

@GET
@Produces({"application/xml"}, {"application/json"})
@Path("/hoteles/{id}/habitaciones")
public List<TipoHabitacion> getDisponibilidad(
    @PathParam("id") Long idHotel
    @QueryParam("fechaInicio") Date fechaInicio,
    @QueryParam("fechaFin") Date fechaFin) {
    return disponibilidadService.disponibilidadPorHotel(idHotel, fechaInicio, fechaFin);
}

@GET
@Produces({"application/xml"}, {"application/json"})
@Path("/hoteles/{id}/reservas")
public List<Reservas> getReservasPorHotel(@PathParam("id") Long idHotel) {
    return reservasDAO.buscarPorHotel(idHotel);
}

@GET
@Produces({"application/xml"}, {"application/json"})
@Path("/reservas/{id}")
public List<Reservas> getReserva(@PathParam("id") Long id) {
    return reservasDAO.buscarPorID(id);
}

@PUT
@Consumes({"application/xml"}, {"application/json"})
@Path("/reservas/{id}")
public void actualizarReserva(@PathParam("id") Long id, Reserva reserva) {
    reservaDAO.actualizar(reserva);
}

@DELETE
@Consumes({"application/xml"}, {"application/json"})
@Path("/reservas/{id}")
public void borrarReserva(@PathParam("id") Long id) {
    Reserva reserva = reservaDAO.buscarPorID(id);
    reservaDAO.borrar(reserva);
}

```

@POST

@Consumes({"application/xml", "application/json"})

@Produces({"application/xml", "application/json"})

@Path("/reservas")

```
public Response crearReserva(JAXBElement<Reserva> reservaJAXB) {  
    Reserva reserva = reservaJAXB.getValue();  
    reserva = reservaDAO.crear(reserva);  
    URI reservaURI=context.getAbsolutePathBuilder().path(reserva.getId()).build();  
    return Response.created(reservaURI).build();  
}  
  
}
```