ReST

Representational State Transfer

¿Qué es REST?

REST es un estilo arquitectural para diseñar Web Services.

REST re-significó el protocolo HTTP.

Los lineamientos arquitecturales que sigue REST, son:

- uso de recursos direccionables
- uso de una interfaz uniforme y acotada
- es orientado a representación
- uso de un protocolo de comunicación sin estado

¿Qué es un Servicio Web?

Según el W3C:
"…un sistema de software diseñado para soportar interacción interoperable máquina a máquina sobre una red. Este tiene una interfaz descrita en un formato procesable por una máquina (específicamente WSDL). Otros sistemas interactúan con el servicios web en una manera prescrita por su descripción usando mensajes SOAP, típicamente enviados usando HTTP con una serialización XML en relación con otros estándares relacionados con la web"

La propuesta de RESTful redefine los "servicios web" para trabajar mejor en la WEB, promoviendo la escalabilidad, la performance y la modificabilidad.

Generalidades del estilo RESTful

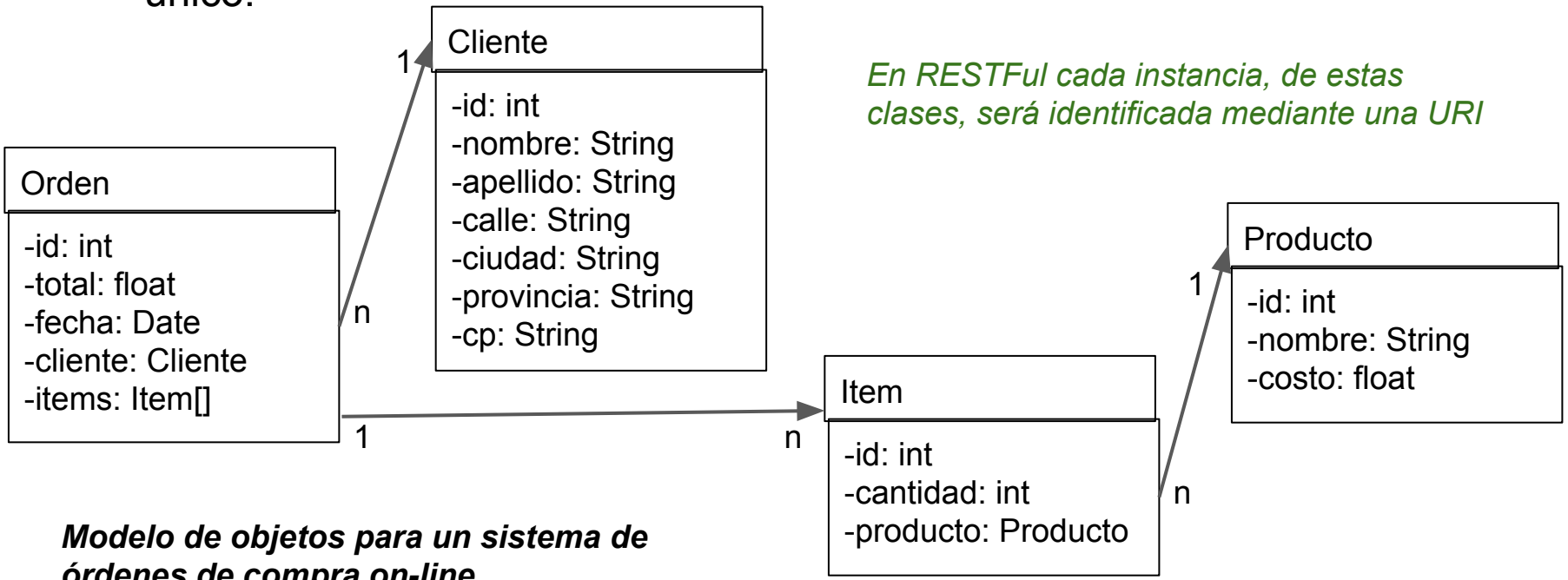
Las aplicaciones RESTFul son **cliente-servidor**.

La comunicación entre el **cliente** y el **servidor** es a través de un **protocolo sin estado**, típicamente HTTP.

El **cliente** y el **servidor** intercambian recursos mediante una interface estandarizada.

Principios de RESTful

1) **Identificación de recursos mediante URIs**: en REST cada objeto o recurso es alcanzable mediante el uso de una URI como identificador único.



En RESTFul cada instancia, de estas clases, será identificada mediante una URI

Modelo de objetos para un sistema de órdenes de compra on-line

Identificación de recursos mediante URIs

http://rest.com/ordenes/233

```
{
  "id": "233",
  "url": "http://rest.com/ordenes/233",
  "total": "$199.02",
  "fecha": "22/12/2015 06:56",
  "cliente": {
    "id": "117",
    "url": "http://rest.com/clientes/117",
    "nombre": "Diego",
    "apellido": "Capusoto",
    "calle": "Corrientes",
    "ciudad": "Buenos Aires",
    "provincia": "Buenos Aires",
    "cp": "1043"
  },
  "items": {
    "item": {
      "id": "144",
      "producto": {
        "id": "543",
        "url": "http://rest.com/productos/543",
        "nombre": "iPhone",
        "costo": "$199.99"
      }
    }
  ...
  }
}
```

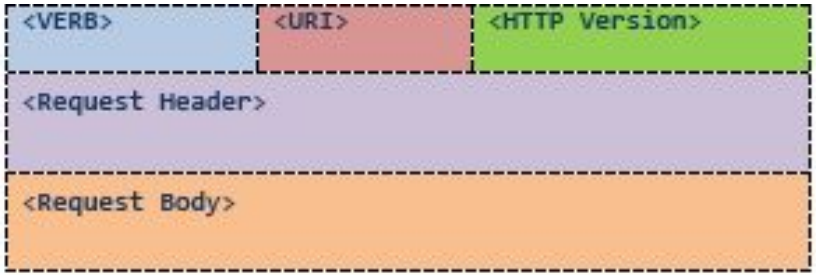
```
{
  "id": "117",
  "url": "http://rest.com/clientes/117",
  "nombre": "Diego",
  "apellido": "Capusotto",
  "calle": "Corrientes",
  "ciudad": "Buenos Aires",
  "provincia": "Buenos Aires",
  "cp": "1043"
}
```

```
{
 "id": "543",
 "url": "http://rest.com/productos/543",
 "nombre": "iPhone",
 "costo": "$199.99"
}
```

URIs RESTFul que identifican los objetos del modelo

Principios de RESTful

2) **Uso de una interfaz uniforme y acotada**: los recursos son manipulados usando un conjunto fijo de cuatro operaciones que permiten su creación, lectura, actualización y eliminación. RESTFul implementa esta funcionalidad mediante los métodos PUT, GET, POST y DELETE de HTTP.



<VERB>: GET, PUT, POST, DELETE, HEAD y OPTIONS
<Request Body>: se especifican los recursos a transferir
<URI> es la URI del recurso sobre el cual se realiza la operación

Principios de RESTful

GET

Se utiliza para consultar por información específica
Operación de sólo lectura
Es idempotente y segura

POST

Se utiliza para almacenar una nueva entidad de datos como un nuevo recurso que se identificara con una URI generada por el servidor
Es no-idempotente e inseguro

Principios de RESTful

PUT

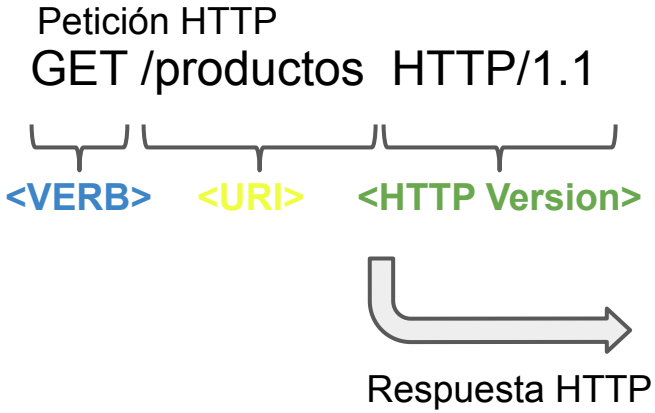
Se utiliza para almacenar el cuerpo del mensaje bajo la ubicación provista en el mensaje HTTP.
Se utiliza para *inserts* o *updates*
Es idempotente, porque ejecutarlo más de una vez no afecta al servicio

DELETE

Se utiliza para eliminar recursos bajo la <URI>
Es idempotente

Uso de una interfaz uniforme y acotada

¿Cómo consultar todos los productos disponibles en el sistema?

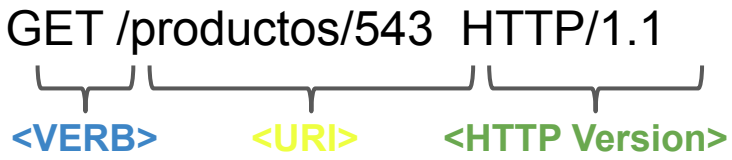



```
HTTP/1.1 200 OK
Content-Type: application/json

[
   {
      "id": "543",
      "url": "http://rest.com/productos/543",
      "nombre": "iPhone",
      "costo": "$199.99"
   },
   {
      "id": "544",
      "url": "http://rest.com/productos/544",
      "nombre": "Samsung",
      "costo": "$159.99"
   }
   ...
]
```

Uso de una interfaz uniforme y acotada

¿Cómo consultar un producto particular?

Petición HTTP



Respuesta HTTP

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

{
 "id": "543",
 "url": "http://rest.com/productos/543",
 "nombre": "iPhone",
 "costo": "$199.99"
}
```

Uso de una interfaz uniforme y acotada

¿Cómo crear un producto como un nuevo recurso?

POST /productos/ HTTP/1.1

<VERB> <URI> <HTTP Version>

```
POST /productos/ HTTP/1.1
Content-Type: application/json

{
 "nombre": "Alcatel",
 "costo": "$100.99"
}
```

Petición HTTP

La respuesta HTTP incluye la nueva URI del recurso creado, para que el cliente pueda referenciarlo

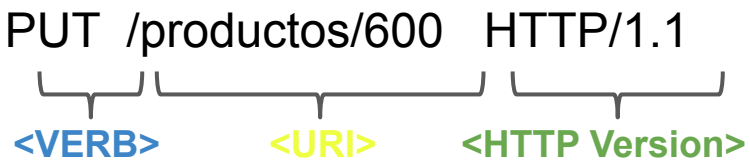
```
HTTP/1.1   201  Created
Content-Type: application/json
Location: http://rest.com/productos/600

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$100.99"
}
```

Obligatorio

Respuesta HTTP

Uso de una interfaz uniforme y acotada

¿Cómo actualizar los datos de un producto existente?




```
PUT /productos/600 HTTP/1.1
Content-Type: application/json

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$123.99"
}
```

Petición HTTP

```
HTTP/1.1   204  No Content
```

ó

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

{
 "id": "600",
 "url": "http://rest.com/productos/600",
 "nombre": "Alcatel",
 "costo": "$123.99"
}
```

Dos posibles respuestas HTTP

Uso de una interfaz uniforme y acotada

¿Cómo eliminar un producto existente?

Petición HTTP

DELETE /productos/545 HTTP/1.1

<VERB> <URI> <HTTP Version>

Dos posibles respuestas HTTP

```
HTTP/1.1   204  No Content
```

ó

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

{
 "id": "545",
 "url": "http://rest.com/productos/545",
 "nombre": "Nokia",
 "costo": "$137.99"
}
```

Interfaz uniforme y acotada

Las **operaciones** de nuestro *sistema de órdenes de compra on-line* se ajustan correctamente a los métodos del protocolo HTTP: se utiliza **GET** para leer, **PUT** para actualizar, **POST** para crear y **DELETE** para remover.

Si bien es posible utilizar parámetros en las peticiones HTTP, RESTFul propone no cambiar la semántica de los métodos HTTP. Por ejemplo agregar un parámetro de nombre “action” al GET indicando la acción de actualizar un recurso, no cumple los principios de RESTFul. En este caso se debería usar el método PUT.

Un buen uso de parámetros en los métodos HTTP, por ejemplo es recuperar todos los productos cuyo nombre comienza con A, B, C, o D

GET /productos?start=A&end=D HTTP/1.1

JAX-RS

JAX-RS es un *framework* Java que se enfoca en el uso de anotaciones para la implementación de servicios RESTFul.

Las **anotaciones** son las que vinculan las **URIs** y las **operaciones HTTP** con los **métodos** de clases Java que **implementan la atención** de los servicios RESTFul.

JAX-RS - Características

- Provee anotaciones que vinculan URIs y métodos HTTP con métodos Java

- Provee inyección de parámetros en las anotaciones, de manera de usarlos fácilmente en los métodos

- Provee lectores y escritores para facilitar la conversión de xml a objetos

Implementaciones JAX-RS

Entre las implementaciones de JAX-RS se incluyen:

- **Apache CXF**, un framework de servicios web de código abierto.
- **Jersey**, la implementación de referencia de Sun (ahora Oracle).
- **RESTeasy**, implementación de JBoss.
- **Restlet**, creado por Jerome Louvel, un pionero en frameworks de REST.
- **Apache Wink**, proyecto de Apache Software Foundation Incubator, el módulo del servidor implementa JAX-RS.
- **JBoss**, la librería org.jboss.resteasy da soporte de JAX-RS sobre la plataforma JBoss.

Anotaciones JAX-RS

JAX-RS proporciona algunas anotaciones para ayudar a mapear una clase POJO como un recurso web. Entre estas anotaciones se incluyen:

- **@Path** especifica la ruta de acceso relativa para una clase recurso o método.
- **@GET**, **@PUT**, **@POST**, **@DELETE** y **@HEAD** especifican el tipo de petición HTTP de un recurso.
- **@Produces** especifica los tipos de medios MIME de respuesta.
- **@Consumes** especifica los tipos de medios de petición aceptados.

JAXB - JAX-RS - Jackson

- **JAXB** es un framework Java que permite mapear y transformar ***clases Java a XML y viceversa***. **JAXB** está basado en anotaciones y es independiente de JAX-RS
- **JAX-RS** tiene incorporado el soporte de JAXB, de esta manera la transformación entre XML y objetos Java es automática facilitando la tarea de programación.
- Nosotros utilizaremos la librería Jackson, que provee conversión automática de JSON a clases Java y viceversa de manera directa. https://github.com/FasterXML/jackson

Tener en cuenta las anotaciones para evitar bucles:

@JsonIgnoreProperties anotación a nivel de clase, para indicar una lista de atributos que no deben serializarse

@JsonIgnore anotación a nivel atributo, para indicar que el atributo no debe serializarse

```
@JsonIgnoreProperties({ "id" })
public class BeanWithIgnore {
    public int id;
    public String name;
}
```

```
public class BeanWithIgnore {
    @JsonIgnore
    public int id;

    public String name;
}
```

Anotaciones JAX-RS

Además proporciona anotaciones adicionales, en general para facilitar la extracción de información de parámetros e información de la solicitud.

@PathParam
@QueryParam
@MatrixParam
@HeaderParam
@CookieParam
@FormParam
@DefaultValue
@Context

```
@Path(/employees/"{firstname}.{lastname}@{domain}.com")
public class EmpResource {

  @GET
  @Produces("text/xml")
  public String getEmployeelastname(@PathParam("lastname") String lastName) {
   ...
  }
}
```

```
GET /employees/john.doe@example.com
```

```
@Path(/employees/")
@GET
public Response getEmployees(
      @DefaultValue("2002") @QueryParam("minyear") int minyear,
      @DefaultValue("2010") @QueryParam("maxyear") int maxyear)
  {...}
```

```
GET /employees?maxyear=2009&minyear=1999
```

Ejemplo JAX-RS utilizando Jersey

Jersey provee un servlet dispacher para los requerimientos REST

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" id="WebApp_ID" version="3.0">
<display-name>RESTful Services CRUD</display-name>
<servlet>
    <servlet-name>RESTfulJersey</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>rest.recursos</param-value>
    </init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>RESTfulJersey</servlet-name>
<url-pattern>/rest/*</url-pattern>
</servlet-mapping>
</web-app>
```

Ejemplo JAX-RS

```
package rest.recursos;

import javax.persistence.RollbackException;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import daos.*;
import modelo.Usuario;

@Path("/usuarios")
public class UsuariosResource {

	@Context
	UriInfo uriInfo;

	@Context
	Request request;

	private UsuarioDAO udao = FactoryDAO.getUsuarioDAO();
	private String mensaje;

	@GET
	@Produces(MediaType.APPLICATION_JSON)
	public List<Usuario> getUsuarios(){
		return udao.list();
	}
	// x defecto devuelve 200 OK
```

```
// URI: /rest/usuarios/23

@GET
@Path("{id}")
@Produces(MediaType.APPLICATION_JSON)
public Response encontrar(@PathParam("id") Integer id){

	Usuario usu = udao.read(id);
	if (user != null){
		return Response
				.ok()
				.entity(usu)
				.build();
	} else {
		mensaje="No se encontró el usuario");
		return Response
				.status(Response.Status.NOT_FOUND)
				.entity(mensaje)
				.build();
	}

}
```

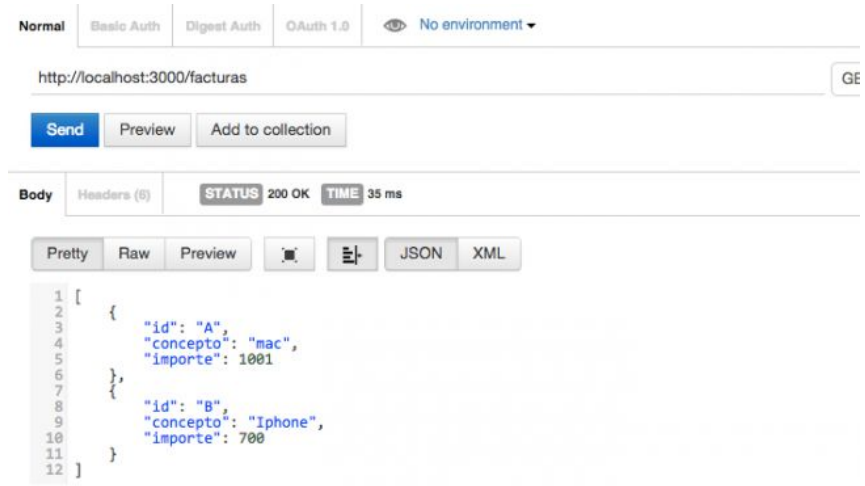
Ejemplo JAX-RS

```java
@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response crear(Usuario usuario){

  if(validar(usuario)){ //podría validar si ya existe el usuario
    udao.create(usuario);
    return Response.status(Response.Status.CREATED).build();
  } else {
    return Response.status(Response.Status.CONFLICT).build();
  }
}


@PUT
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response editar(Usuario usuario){
  Usuario aux = udao.read(usuario.getId());
  if (aux != null){
       udao.update(usuario);
       return Response.ok().entity(usuario).build();
  } else {
       return Response.status(Response.Status.NOT_FOUND)
                                        .entity("[]").build();
  }
}
```

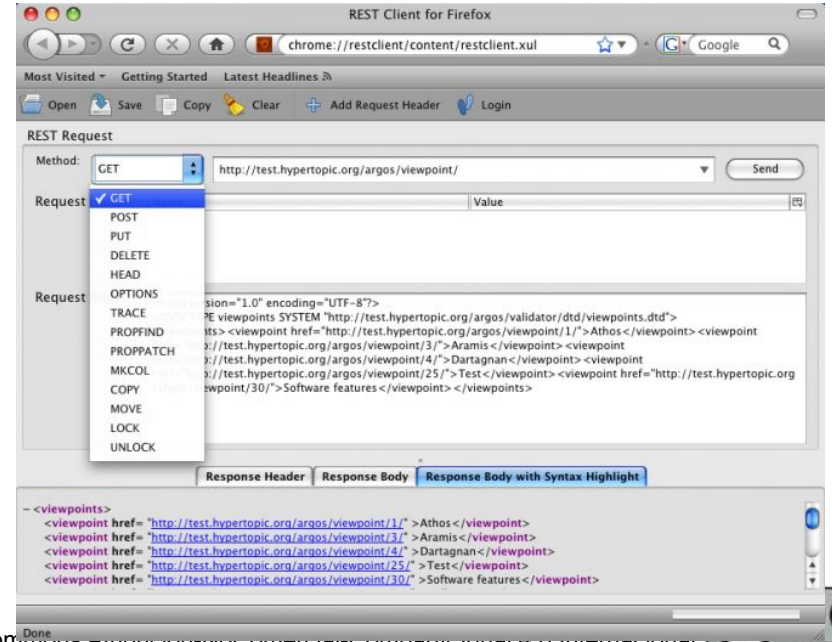
```java
@DELETE
@Path("{id}")
@Produces(MediaType.TEXT_PLAIN)
public Response borrar(@PathParam("id") Integer id){

  Usuario aux = udao.read(id);
  if (aux != null){
      udao.delete(aux);
      return Response.noContent().build();
  } else {
    mensaje = "No existe el usuario con ese id";
    return Response.status(Response.Status.NOT_FOUND)
                                    .entity(mensaje)
                                    .build();
  }
}
```

Cliente REST de prueba: POSTMAN

POSTMAN: Extensión de Chrome



POSTMAN: Extensión de Firefox



JAX-RS y Jersey - Configuraciones Avanzadas

JAX-RS usa la clase javax.ws.rs.core.Application como clase de configuración
Jersey usa org.glassfish.jersey.server.ResourceConfig que es una subclase de Application

Hay tres maneras de configurar Jersey (JAX-RS)

1) Usando solo web.xml (la presentada en el slide 22)
2) Usando web.xml y la clase Application/ResourceConfig
3) Usando solo una clase Application/ResourceConfig anotada con @ApplicationPath.

JAX-RS y Jersey - Configuraciones Avanzadas

Mediante el método 2)

web.xml

```
<servlet>
    <servlet-name>jersey-servlet</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
        <param-name>javax.ws.rs.Application</param-name>
        <param-value>com.example.JerseyApplication</param-value>
    </init-param>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</servlet>
```

JerseyApplication.java

```
package com.example;

public class JerseyApplication extends ResourceConfig {
    public JerseyApplication() {
        packages("com.abc.jersey.services");
    }
}
```

Mediante el método 3)

```
@ApplicationPath("services")
public class JerseyApplication extends ResourceConfig {
    public JerseyApplication() {
        packages("com.abc.jersey.services");
    }
}
```

En este caso el servlet-mapping es: */services/**
Equivalente al tag <url-pattern>/services/*</url-pattern>

La interface ServletContainerInitializer

Para que funcionen los métodos 2) y 3) se hace uso de una característica introducida en la especificación de Servlet 3.0.

La especificación permite definir servlets, filtros y listeners de manera programática en vez de declarativa (web.xml o anotaciones).

Para llevarlo a cabo el estándar establece un archivo en una determinada ubicación:
META-INF/services/**javax.servlet.ServletContainerInitializer**

Dentro del archivo **javax.servlet.ServletContainerInitializer** se establece el nombre completo de la clase que implementa la interface **javax.servlet.ServletContainerInitializer**.

La interface **ServletContainerInitializer** tiene un método llamado ***onStartup*** que es invocado por el contenedor Java, dentro de este método es posible definir servlets, filtros y listeners.

La interface ServletContainerInitializer

```
public class MyServletContainerInitializer implements ServletContainerInitializer {

    private static final Logger LOGGER = LoggerFactory.getLogger(MyServletContainerInitializer.class);

    public MyServletContainerInitializer(){
        super();
    }

    @Override
    public void onStartup(final Set<Class<?>> handlerTypeSet, final ServletContext servletContext) throws ServletException {
        servletContext.addServlet(SERVLET_NAME, MyServlet.class);
        ...
        servletContext.addFilter(FILTER_NAME, MyFilter.class);
        ...
        servletContext.addListener(MyListener.class);
        ...
    }
}
```

Muchos frameworks (como Spring o Jersey) implementan **ServletContainerInitializer** para permitir a los programadores configurar sus aplicaciones usando clases en vez de archivos xml.

En el caso de Jersey la implementación de ServletContainerInitializer es la que busca la subclase de Application/ResourceConfig anotada con @ApplicationPath.

Integrando Jersey y HK2

Para integrar HK2 con Jersey se puede utilizar el método 2) y así registrar nuestro *binder*

AppConfig.java

```
package com.unlp.jyaa.txi.config;

import org.glassfish.jersey.server.ResourceConfig;

public class AppConfig extends ResourceConfig {

  public AppConfig() {


    register(new MyApplicationBinder());
    packages(true, "com.unlp.jyaa.txi.api");

  }
}
```

MyApplicationBinder.java

```
package com.unlp.jyaa.txi.config;

import org.glassfish.hk2.utilities.binding.AbstractBinder;
import org.glassfish.jersey.process.internal.RequestScoped;

public class MyApplicationBinder extends AbstractBinder {
 @Override
 protected void configure() {


   bind(ActividadServiceImpl.class).to(ActividadService.class)
                                   .in(RequestScoped.class);
   //ó bind(JustInTimeServiceResolver.class)
                       .to(JustInTimeInjectionResolver.class );
   ..


 }
}
```

Lineamientos para diseñar una API Rest

¿Cómo diseñamos una API Rest para nuestra aplicación?

Tenemos que pensar:

1) En las operaciones CRUD (Create, Read, Update, Delete) necesarias aplicar a cada entidad del modelo de objetos.
2) En las operaciones más complejas que involucren a más de una entidad.
3) Las URIs y métodos HTTP de cada operación con atención en la cabecera (content-type) y cuerpo de la petición (datos).
4) Las respuestas de cada método HTTP con atención en la cabecera y en el cuerpo de la respuesta (datos).

Luego se implementa….

Conclusiones

API Rest propone un **paradigma de desarrollo abierto** con una clara separación entre el cliente y el servidor.

El backend y el frontend pueden evolucionar/refactorizar de manera separada, siempre que se mantenga la interfaz de la API.

Es posible crear tantos clientes como se necesite con la misma API: una app web, una app Android, un app iOS, etc.

La filosofía de API Rest es independiente de la tecnología y de los lenguajes con la que se desarrollen los proyectos, sólo es necesario respetar "el contrato" de la API.

Mercado Libre, Twitter, Facebook, Despegar y otras empresas cuyos negocio se basa en tecnologías usan API Rest