DSIC
DEPARTAMENT DE SISTEMES INFORMÀTICS Y COMPUTACIÓ

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Internet de las Cosas

Máster MUCNAP

Seminario 0
Diseño e Implementación de Servicios RESTful con RESTlet

Versión 1.0
Actualización: 16/11/2020

Profesor: Joan Fons i Cors

Control de versiones

Fecha	Autor	Descripción
16/11/2020	Joan Fons	Versión inicial del documento

Departamento de Sistemas Informáticos y Computación
Camino de Vera, s/n 46022 Valencia
Joan Fons i Cors, jjfons@dsic.upv.es, PROS/VRAIN UPV

Departamento de Sistemas Informáticos y Computación
Camino de Vera, s/n 46022 Valencia
Joan Fons i Cors, jjfons@dsic.upv.es, PROS/VRAIN UPV

Desarrollo de Servicios RESTful

Objetivos

- Aprender a Diseñar e Implementar Servicios RESTful usando el framework RESTlet.
- Utilizar el IDE Eclipse para la implementación de los proyectos de Ejemplo.
- Utilizar RESTClient para consultar servicios REST a través del Navegador.

Tabla de Contenido

Departamento de Sistemas Informáticos y Computación
Camino de Vera, s/n 46022 Valencia
Joan Fons i Cors, jjfons@dsic.upv.es, PROS/VRAIN UPV

Introducción

Este documento presenta un pequeño tutorial o laboratorio práctico para aprender a diseñar e implementar una serie de servicios RESTful usando el framework RESTlet. Los servicios RESTful a diseñar servirán para acceder a servicios de gestión de listas. La implementación de los servicios hará uso de las clases implementadas en el proyecto `ToDoList`.

Se le proporcionará al alumno el siguiente código fuente Java:

- Proyecto Java que incluye las clases que permiten implementar la gestión de listas `ToDoList` (y su empaquetado, ToDoList.jar)
- Parte del código Java de la capa REST 'myRESTToDoList'
- Las librerías para desarrollar con RESTlet y JSON.

Departamento de Sistemas Informáticos y Computación
Camino de Vera, s/n 46022 Valencia
Joan Fons i Cors, jjfons@dsic.upv.es, PROS/VRAIN UPV

Implementación de la Gestión de Listas. ToDoList

En esta parte de la práctica vamos a implementar un proyecto Java que incluya las clases que permitan llevar a cabo la gestión de una lista de cosas o acciones a llevar a cabo (una `ToDoList`).

Empezaremos por la definición de la estructura básica: La **Lista**. Una lista no será más que una estructura de datos, con un identificador de lista y un nombre, que nos permitirá albergar contenidos en la lista (o entradas de lista). Una entrada de la lista a su vez, poseerá un identificador y un texto.

Así, por ejemplo, podríamos tener la siguiente lista:

- Lista: (id) **compra**, (nombre) **Cosas a comprar**
  - Entrada Lista: (id) leche, (texto) 1 caja de 6 bricks de leche desnatada
  - Entrada Lista: (id) **huevos**, (texto) **1 docena de huevos**
  - Entrada Lista: (id) **arroz**, (texto) **2Kgs arroz**

Una vez definida la Lista, una ToDoList podrá verse como un conteneder de varias listas (cada una con sus entradas de lista, como hemos visto antes):

- ToDoList
  - Lista: **compra**
  - Lista: **recordatorios**
  - Lista: **trabajos**

Conceptualmente, este podría ser el modelo o estructura de datos que representara la información sobre las listas y sus entradas:

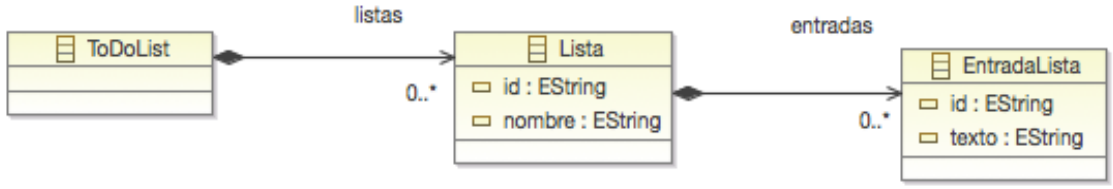

Figura 1. Modelo de Datos

Se adjunta como material complementario, la implementación en Java de las clases que implementan esta estructura de datos para facilitar la gestión de listas

(proyecto Java `ToDoList`). El proyecto se puede agregar al espacio de trabajo de Eclipse según podemos ver en la figura 2:

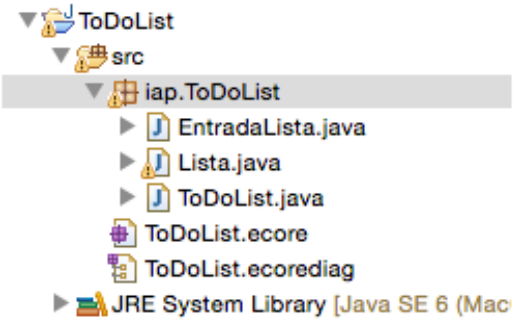

Figura 2. Proyecto ToDoList

El paquete *`iap.ToDoList`* ofrece la implementación de las 3 clases, ofreciendo la funcionalidad para crear una ToDoList (o borrarla), añadirle Listas (y borrarlas), y añadir/modificar/borrar entradas a una Lista. A continuación se presenta el código que implementa las tres clases.

ToDoList.java

```
package iap.ToDoList;

import java.util.Collection;

public class ToDoList {

    protected Map<String, Lista> listas = new Hashtable<String, Lista>();

    public Lista addLista(String nom) {
        Lista ll = new Lista(nom);
        this.listas.put(nom, ll);
        return ll;
    }

    public void removeLista(String nom) {
        Lista ll = this.getLista(nom);
        if (ll == null)
            return;
        ll.remove();
        this.listas.remove(nom);
    }

    public Collection<Lista> getListas() {
        return this.listas.values();
    }

    public Lista getLista(String nom) {
        return this.listas.get(nom);
    }

    public void remove() {
        this.listas.clear();
    }
}
```

Figura 3. Código Java de la clase ToDoList

Lista.java

```java
package iap.ToDoList;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Hashtable;
import java.util.List;
import java.util.Map;

public class Lista {

    protected String id = null;
    protected String nombre = null;
    protected Map<String, EntradaLista> elements = new Hashtable<String, EntradaLista>();

    public Lista(String nom) {
        this.setID(nom.trim());
        this.setNombre(nom);
    }

    public void setID(String id) {  this.id = id; }

    public String getID() { return this.id; }

    public String getNombre() { return this.nombre; }

    public void setNombre(String nombre) { this.nombre = nombre; }

    public Collection<EntradaLista> getEntrades(){ return this.elements.values(); }

    public EntradaLista getEntrada(String id) { return this.elements.get(id); }

    public EntradaLista addEntrada(String id, String text) {
        EntradaLista ell = new EntradaLista(this.getNombre(), id, text);
        this.elements.put(id, ell);
        return ell;
    }

    public void removeEntrada(String id) {
        this.elements.remove(id);
    }

    public void remove() {
        this.elements.clear();
    }

}
```

Figura 4. Código Java de la clase Lista

EntradaLista.java

```java
package iap.ToDoList;

public class EntradaLista {

    protected String llista = null;
    protected String text = null;
    protected String id;

    public EntradaLista(String lista, String id, String text) {
        this.llista = lista;
        this.id = id;
        this.text = text;
    }

    public String getLista() {
        return this.llista;
    }

    public String getID() {
        return this.id;
    }

    public String getText() {
        return this.text;
    }

    public void updateText(String text) {
        this.text = text;
    }


}
```

Figura 5. Código Java de la clase EntradaLista

Este proyecto se usará más adelante para implementar los servicios RESTful que permitirán ofrecer servicios para crear y mantener una ToDoList.

En la siguiente sección vamos a presentar cómo se llevará a cabo el diseño de los servicios RESTful siguiendo la filosofía y las prescripciones comentadas en clase.

Diseño de los Servicios RESTful

Para diseñar una capa REST (siguiendo las recomendaciones para construir soluciones 'RESTful compliant') hay que definir los siguientes conceptos:

- **Recursos e Identificadores**: identificar los recursos que van a poder ser manipulados a través de los servicios RESTful. A cada recurso deberá asignársele un identificador único y universal para poder referenciarlos adecuadamente.
- **Representaciones de los recursos**: información (datos) que se mostrará para cada recurso. Según el patrón RESTful se puede ofrecer más de una representación por cada recurso.
- **Interfaz Uniforme**: para cada recurso, se ofrecerá operaciones para consultar información del recursos y modificarlo a través de operaciones de la interfaz uniforme de HTTP (típicamente, GET, POST, PUT y DELETE, pero también HEAD u OPTIONS). Las implementación de cada operación deberá ser acorde a la naturaleza de la operación HTTP.

RECURSOS E IDENTIFICADORES

Existen dos recursos principales y fáciles de obtener a partir del proyecto ToDoList presentado: las Listas, y las Entradas de Lista. De ellos nos interesará ofrecer la posibilidad de consultar su información, y solicitar operaciones de actualización.

Además, nos hará falta un tercer recurso, la ToDoList, para poder lanzar consultas y actualizaciones sobre las listas de cosas a llevar a cabo disponibles.

Así pues, tendremos tres recursos relevantes para nuestro sistema:
- ToDoList
- Lista
- EntradaLista

El siguiente paso será decidir la manera en que nos referiremos a ellos (mediante sus identificadores de recurso). El patrón RESTful reutiliza el concepto de URL para identificar recursos, para soportar esa universalidad y unicidad.

La sintaxis típica de una URL es la siguiente:

```
http://servidor:puerto/ruta-recurso
```

Dado que el servidor y el puerto están por definir (en función de cómo se configure la solución), definiremos las URLs de momento usando este servidor y puerto como 'parámetros de configuración del programa' (los sabremos cuando ejecutemos el programa).

Así pues, los identificadores para los diferentes recursos podrían ser:

- ToDoList: `http://servidor:puerto/listas`
- Lista: `http://servidor:puerto/lista/{LISTA}`
- EntradaLista: `http://servidor:puerto/lista/{LISTA}/entrada/{ID}`

donde `{LISTA}` e `{ID}` representan los identificadores de la lista y de la entrada de lista respectivamente. Como se puede observar, tanto la Lista como EntradaLista están parametrizadas para poder acceder a una lista o entrada de lista en concreto. En la ToDoList, no hay parámetro (habrá una única ToDoList).

Así, por ejemplo, en el ejemplo anterior, podríamos tener los siguientes identificadores de recursos:

`http://servidor:puerto/listas` : acceso a las listas disponibles
`http://servidor:puerto/lista/compra` : acceso a la lista de la compra
`http://servidor:puerto/lista/compra/entrada/huevos`: acceso a la entrada huevos de la lista de la compra

REPRESENTACIONES

Las representaciones describen qué datos y qué formato representarán al recurso. Recordad que el patrón RESTful recomienda que desarrollemos aplicaciones con recursos enlazados. Esto significa, que accediendo a un recurso, éste nos dará el acceso a otro relacionado (como si de un enlace HTML se tratara). Dado que los identificadores de los recursos REST son las URLs, en la práctica, REST recomienda asociar a cada representación de recurso la URL de dicho recurso.

En cuanto a la información a recuperar de cada recurso, según lo comentado antes:

- **ToDoList**: no tiene atributos, pero contiene un listado de listas, por lo que se asociará a la representación de este recurso obtener esta lista de listas, indicando para cada lista su id, y su identificador REST (URL).
- **Lista**: mostrará su identificador, su nombre y su URL, y de la misma manera que antes, un listado de las entradas (su id) y su URL.
- **EntradaLista**: mostrará su identificador, texto y URL. Además, se le asociará la URL a la lista en la que se encuentra definido.

Llegados a este punto, hay que decidir en qué formatos de datos se almacenan y ofrecen las representaciones. Típicamente se utiliza JSON (listado de parejas clave-valor), pero también podría ser XML, HTML, texto plano, PDF, imágenes, etc.

En el ejemplo que mostraremos, hemos elegido JSON, muy utilizado en soluciones REST. Así, por ejemplo,

a) una representación en JSON de la ToDoLlist, podría ser:

```
{
	"link" : "/listas",
	"listas" : [
		{
			"id" : "compra",
			"link" : "/lista/compra"
		},
		{
			"id" : "recordatorios",
			"link" : "/lista/recordatorios"
		},
		{
			"id" : "trabajos",
			"link" : "/lista/trabajos"
		}
	]
}
```

b) una representación en JSON de la lista 'compra' de antes, podría ser:

```
{
	"id" : "compra",
	"nombre" : "Cosas a comprar",
	"link" : "/lista/compra",
	"entradas" : [
		{
			"id" : "leche",
			"link" : "/lista/compra/entrada/leche"
		},
		{
			"id" : "huevos",
			"link" : "/lista/compra/entrada/huevos"
		},
		{
			"id" : "arroz",
			"link" : "/lista/compra/entrada/arroz"
		}
	]
}
```

De manera similar, tendríamos las representaciones para una entrada de lista, o la ToDoList.

INTERFAZ UNIFORME

El último paso para diseñar los servicios REST es mapear las operaciones de la interfaz uniforme de http con acciones a realizar para cada recurso.

Normalmente esta asignación se realiza mediante una tabla, en la que se indica qué operaciones HTTP están permitidas (indicando su significado sobre la aplicación), y cuáles no. La tabla 1 muestra la asignación para la aplicación de la ToDoList:

	ToDoList	Lista	EntradaLista
GET	✓ Info Listas	✓ Info Lista	✓ Info Ent. Lista
POST	✓ Crear Lista	✓ Crear Ent. Lista	✗
PUT	✗	✓ Modif. Lista	✓ Modif. Ent. Lista
DELETE	✓ Borrar ToDoList	✓ Borrar Lista	✓ Borrar Ent. Lista
HEAD	✓	✓	✓
OPTIONS	✓	✓	✓

Tabla 1. Mapeo de Operaciones

Si nos fijamos en esta asignación, podemos observar lo siguiente:

- Las operaciones GET están asociadas a obtener información de cada recurso.
- Las operaciones POST están asociadas a constructores (en la ToDoList se pueden crear Listas, y en las Listas se crean Entradas de Lista). Para EntradaLista no se ofrece esta operación.
- Las operaciones PUT están asociadas a modificar los datos de los recursos. No se permite modificar una ToDoList (no tiene mucho sentido, ya que no tiene datos).
- Las operaciones DELETE están asociadas a borrar recursos.
- Las operaciones HEAD nos permitirán obtener información resumida sobre los recursos.
- Las operaciones OPTIONS nos mostrarán, asociado a cada recurso, el conjunto de operaciones de la interfaz uniforme disponible (tabla 1).

Implementación con RESTlet

RESTlet es un framework para desarrollar servidores y clientes REST. La versión más reciente de RETSlet es la 2.X (actualmente 2.3.1), que simplifica significativamente el desarrollo frente a su versión anterior. Este framework es considerado uno de los más fieles al patrón RESTful, frente a otros frameworks como Jersey, JAX-RS o Django.

La jerarquía de clases que ofrece RESTlet es relativamente sencilla (ver figura 6):

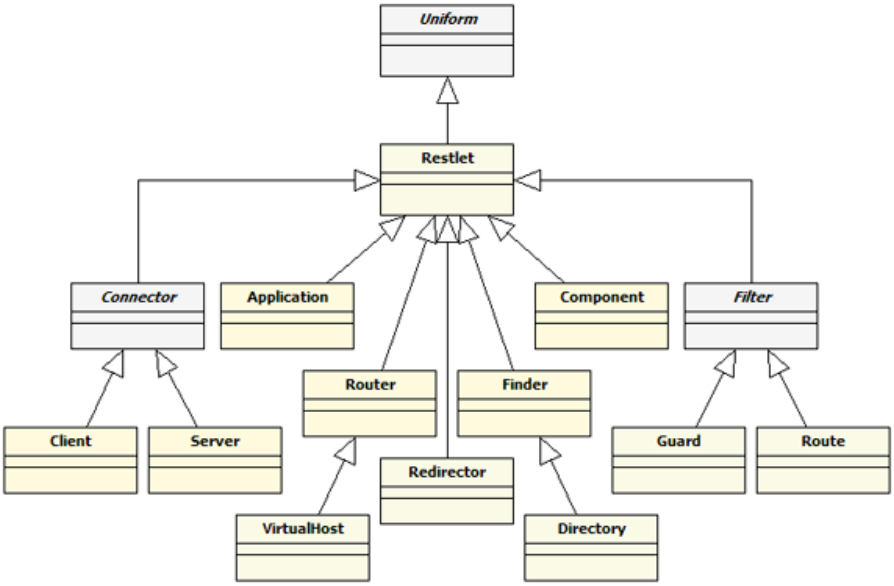

Figura 6. Jerarquía de clases del framework RESTlet

De estas clases (figura 6), las que usaremos para implementar la solución REST son:

- **Component**: alberga un conjunto de aplicaciones REST, y define (entre otros) los mecanismos de acceso a la funcionalidad REST (VirtualHost).
- **Applicaton**: define una aplicación REST.
- **Router**: nos permite configurar la URL de acceso a cada recurso que la aplicación ofrece
- **ServerResource**: representan los recursos. Cada recurso extenderá esta clase

A continuación vamos a construir el proyecto `myRESTToDoList` que ofrecerá los servicios REST mediante esta implementación de RESTlet 2.0.

CREACIÓN Y CONFIGURACIÓN DEL PROYECTO

El primer paso a llevar a cabo será crear el proyecto. Tenemos diferentes opciones, pero podemos usar un proyecto Java básico.

Ejercicio 1: Crear un proyecto Java llamado 'myRESTToDoList'.

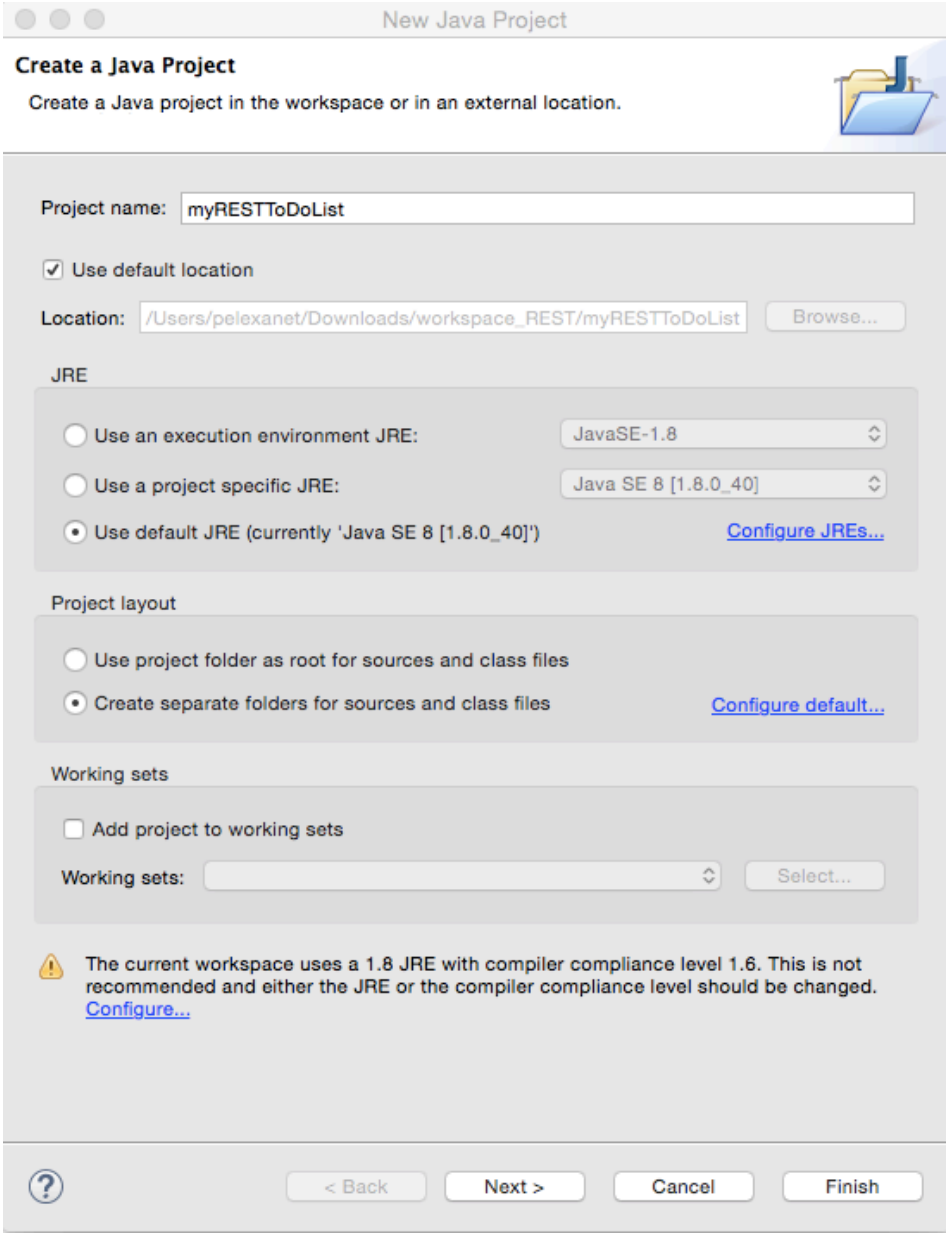

Figura 7. Creación del proyecto myToDoList

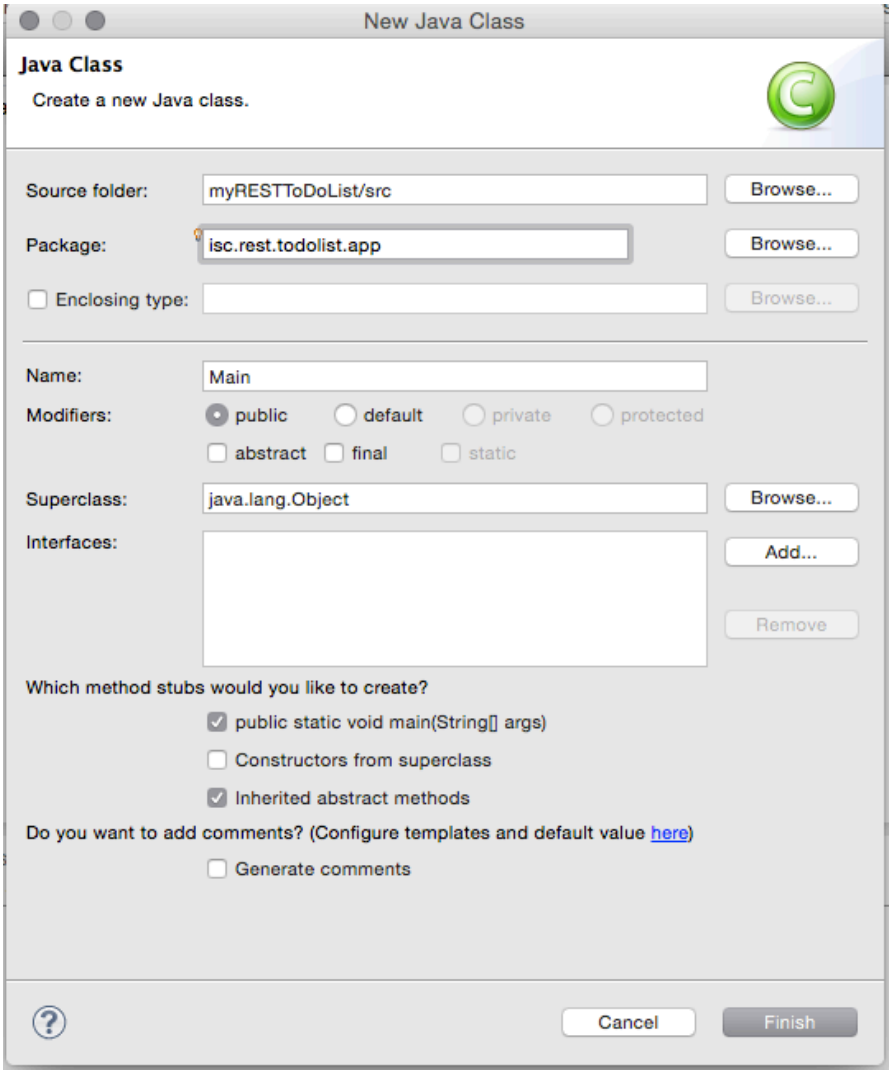

Figura 8. Creación de la clase Main

Ejercicio 2: Crear una clase Main (dentro del paquete isc.rest.todolist.app) y solicitar que nos genere el '`main()`'

Vamos a crear una clase que inicializará el proyecto creando un conjunto inicial de listas. Ver Figura 8.

Usaremos el método `main()` para inicializar un conjunto de listas. Para poder hacerlo, necesitaremos importar primero la librería ToDoList.jar, que contiene la implementación del proyecto ToDoList.

Ejercicio 3: Importar la librería ToDoList.jar.

Para hacerlo, primero deberemos crear una carpeta 'lib' en el proyecto y copiar la librería en esta carpeta.

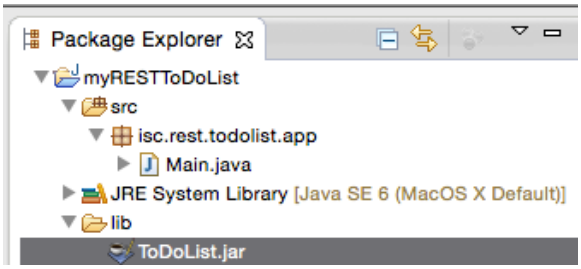

Figura 9. Importación de ToDoList.jar

Ahora debemos añadirla al **Build Path** (haciendo click secundario sobre ella)

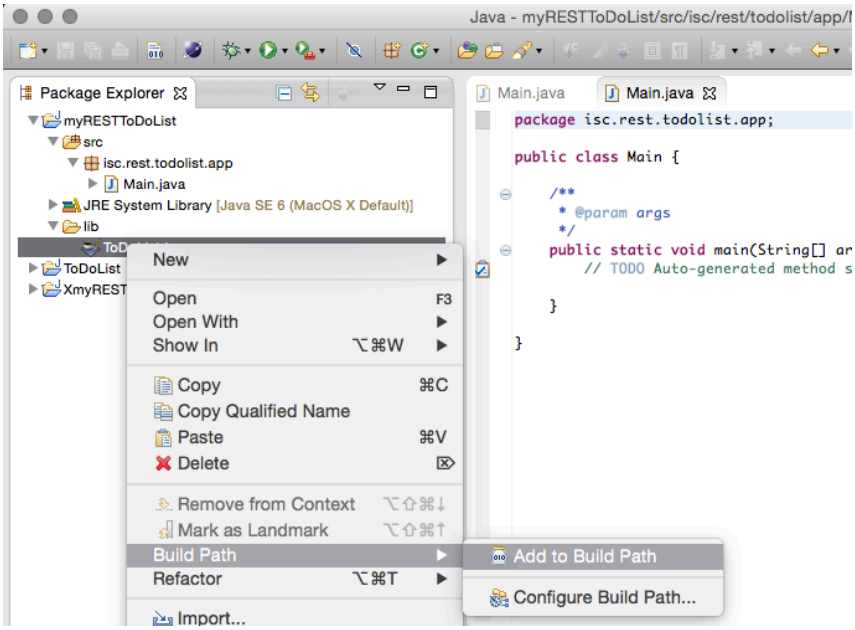

Figura 10. Añadimos el archivo .jar al Build Path

Una vez hecho este paso, ya podemos crear una **ToDoList**, **Listas** y **Entradas de Lista**.

Ejercicio 4: Inicializar un conjunto de listas (dentro del main).

```
ToDoList tdl = new ToDoList();

Lista compra = tdl.addLista("compra", "Lista de la compra");
compra.addEntrada("leche", "1 caja de 6 bricks de leche");
compra.addEntrada("huevos", "1 docena de huevos");
compra.addEntrada("arroz", "2 Kgs de arroz");

Lista recordatorios = tdl.addLista("recordatorios", "Recordatorios
pendientes");
recordatorios.addEntrada("paquete", "Recojer paquete de correos");
recordatorios.addEntrada("reunion", "Reunion trimestral del colegio");

Lista tp = tdl.addLista("trabajos", "Trabajos Pendientes");
tp.addEntrada("ninot", "Hacer manualidad 'ninot de falla'");
tp.addEntrada("rest", "Desarrollar API REST para asignatura ISC");

muestraToDoList(tdl);
```

Se requerirá importar los siguientes paquetes:

```
import iap.ToDoList.EntradaLista;
import iap.ToDoList.Lista;
import iap.ToDoList.ToDoList;
```

Para visualizar el contenido de la lista por consola, vamos a crear un método en la clase Main que nos imprima la lista en la consola. Como no está todavía creado nos generará un error.

Ejercicio 5: Crear el método muestraToDoList() e invocarlo desde el main con la ToDoList.

```
public static void muestraToDoList(ToDoList tdl) {

 for (Lista l : tdl.getListas()) {
      System.out.println(l.getID() + ": " + l.getNombre());
      for (EntradaLista el : l.getEntradas())
            System.out.println("\t" + el.getID() + ": "+ el.getText());
 }

}
```

Ahora vamos a ejecutar el programa y comprobar que la consola nos muestra adecuadamente las listas creadas.

Ejercicio 6: Ejecutar el Main y observar las listas en la consola de ejecución

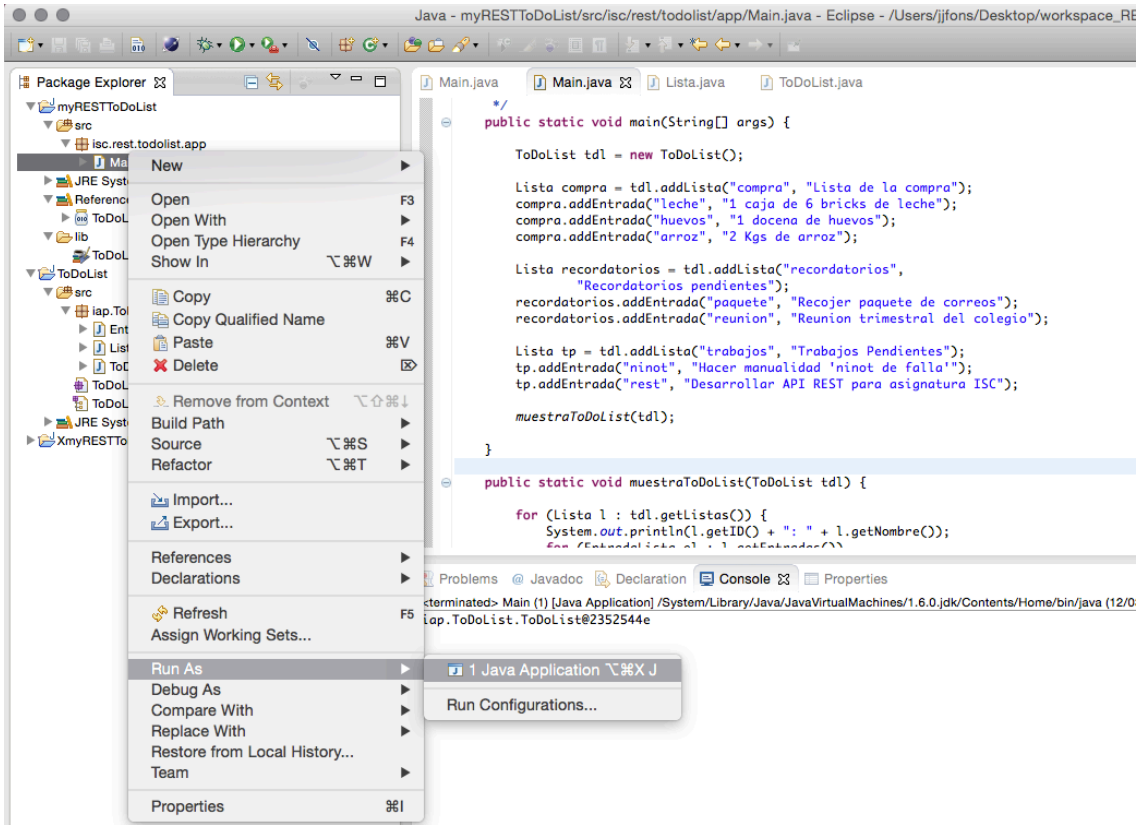

Figura 11. Ejecución de la Aplicación

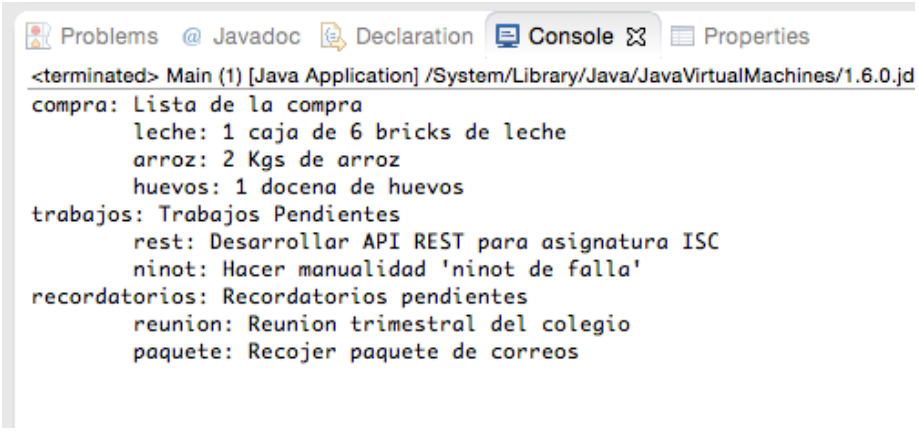

Figura 12. Visualización en Consola de la Lista

CONFIGURACIÓN INICIAL DE LA CAPA REST con RESTLET 2.0

El primer paso será configurar un **Component** que será el contenedor donde albergar el servicio REST, para ello primero necesitaremos importar el paquete RESTlet.

Ejercicio 7: Añadir la librería org.reslet.jar al directorio lib e importarlo en el Build Path.

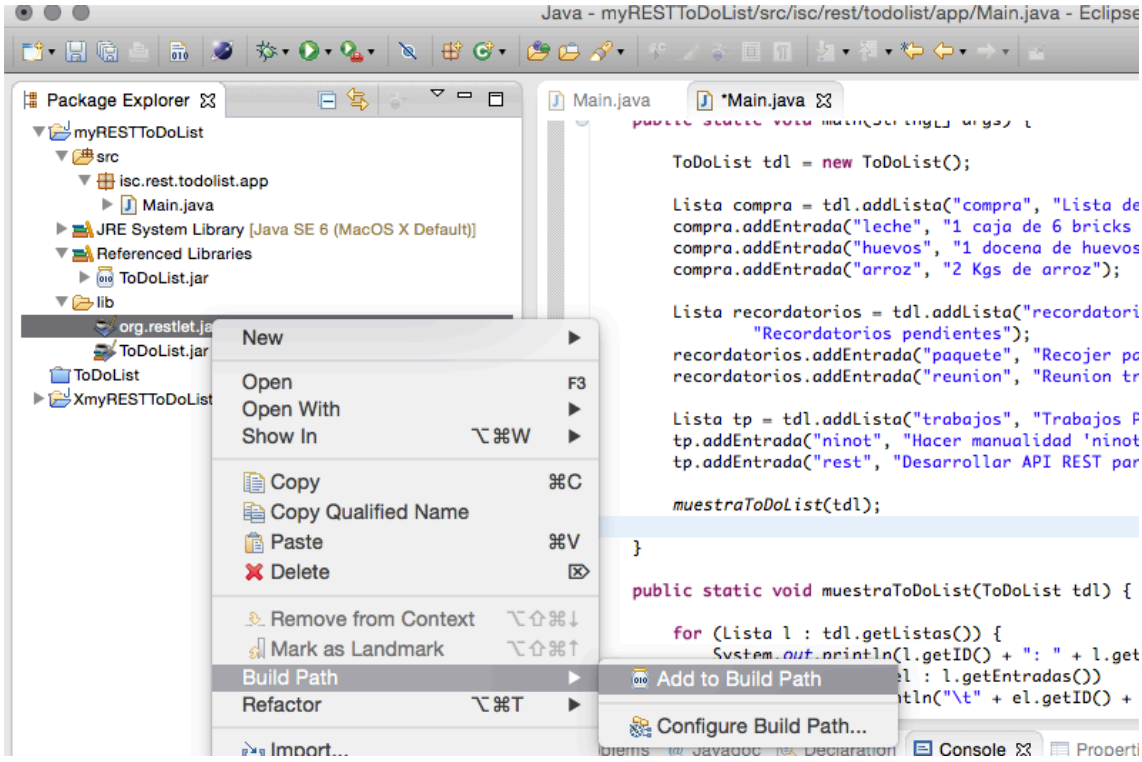

Figura 13. Añadir e importar el Build Path

Ejercicio 8: Extender el main() para configurar el contenedor del servicio REST para que responda a peticiones HTTP en el puerto 8182.

```
// Create a new Component.
Component component = new Component();

// Add a new HTTP server listening on port 8182.
component.getServers().add(Protocol.HTTP, 8182);
```

Deberemos importar los paquetes:

```
import org.restlet.Component;
import org.restlet.data.Protocol;
```

CREACIÓN DE LA 'APPLICATION' REST

Ahora debemos construir lo que es en sí la aplicación REST que atenderá las peticiones HTTP en el puerto 8182 (como hemos configurado en el paso anterior).

Para ello, deberemos implementar una clase que extienda la clase `Application`. Se nos exigirá implementar el método `createInboundRoot`, donde se registrarán los enrutadores (`Routers`) a los recursos que gestionarán las peticiones.

Ejercicio 9: Crear una nueva clase RESTToDoListApplication (en el paquete isc.rest.todolist.app).

```
import iap.ToDoList.ToDoList;

import org.restlet.Application;
import org.restlet.Restlet;
import org.restlet.routing.Router;

public class RESTToDoListApplication extends Application {

	protected ToDoList list = null;

	public RESTToDoListApplication(ToDoList list) {
		this.list = list;
	}
```

```
	public ToDoList getToDoList() {
		return this.list;
	}

	@Override
	public Restlet createInboundRoot() {

		Router router = new Router(getContext());

		// Define routers for users
		router.attach("/listas", ToDoListResource.class);

		return router;
	}

}
```

Inicialmente nos dará un error en la línea donde definimos el **router** (de momento lo ignoramos), ya que todavía no existe la clase que gestionará el recurso asociado (ToDoListResource) a la URL. Este **router** se ha configurado sobre la URL `"/listas"` que es la URL que va a identificar los recursos ToDoList (tal y como habíamos definido en la sección RECURSOS E IDENTIFICADORES).

Ejercicio 10: Crear la clase ToDoListResource (crear un paquete nuevo, isc.rest.todolist.resources) que extienda la clase ServerResource de RESTlet.

```
import org.restlet.resource.ServerResource;
import iap.ToDoList.ToDoList;
import isc.rest.todolist.app.RESTToDoListApplication;

public class ToDoListResource extends ServerResource {

 protected ToDoList getToDoList() {
   return
((RESTToDoListApplication)this.getApplication()).getToDoList();
 }

}
```

El método `getToDoList()` nos servirá más adelante para obtener la ToDoList (almacenada en la Aplicación) sobre la que trabajaremos.

La clase `ToDoListResource` deberá gestionar las operaciones de la interfaz uniforme definidas en la tabla de operaciones vista en la sección INTERFAZ UNIFORME. Es decir, deberemos soportar los métodos de la ToDoList: GET, POST, DELETE, HEAD[1] y OPTIONS.

RESTlet nos ofrece anotaciones para cada operación (`@Get`, `@Post`, `@Put`, `@Delete`, `@Options`) que asociaremos a los métodos que implementen estas operaciones.

Estas operaciones devuelven una `Representation` (que nosotros mapearemos a las representaciones que hemos definido en REPRESENTACIONES), que representan los datos que devuelve la operación REST definida (en nuestra implementación, usaremos como formato, JSON).

Las operaciones que reciban un `Payload` (datos de entrada) tendrán un parámetro en el método de tipo Representation con los datos que se nos envíe (en nuestra implementación usaremos como formato JSON).

Una vez creado este paquete con la clase `ToDoListResource` recordad que debemos importar este paquete en la implementación de la clase `RESTToDoListApplication`.

```
import isc.rest.todolist.resources.ToDoListResource;
```

Las siguientes secciones nos mostrarán cómo implementar cada operación de la ToDoList.

Pero antes de seguir, nos queda un último paso: inicializar nuestra aplicación REST en el main().

[1] Para no complicar el desarrollo con acceso a los metadatos de la cabecera que requiere la operación HEAD, esta operación no se implementará en este seminario.

Ejercicio 11: Acabar de configurar el arranque de la aplicación REST en main(), instanciando una aplicación ToDoListRESTApplication.

```
public static void main(String[] args) {

	...

	// Create a new Component.
	Component component = new Component();

	// Add a new HTTP server listening on port 8182.
	component.getServers().add(Protocol.HTTP, 8182);

	// Attach the ToDoList application.
	Application app = new RESTToDoListApplication(tdl);

	component.getDefaultHost().attach("", app);

	// Start the component.
	try {
		component.start();
	} catch (Exception e) {
		e.printStackTrace();
	}
}
```

Incluir el siguiente **import** para que no tengamos problemas con `Application`.

```
import org.restlet.Application;
```

IMPLEMENTACIÓN DE UN @GET

Este método nos permite obtener información de un recurso. Su sintaxis típica en RESTlet 2.0 es la siguiente:

```
@Get
public Representation getResourceInfo() throws JSONException
```

Ejercicio 12: Implementar la operación @Get para la ToDoList en la clase ToDoListResource.

```
@Get
public Representation getListasJSON() throws JSONException {

 return new StringRepresentation(
  ToDoListResourceSerializer.getAsJSON(this.getToDoList()).toString(),
  MediaType.APPLICATION_JSON);
}
```

Analicemos el código. Este método devuelve una `StringRepresentation`. Este es el tipo de `Representation` más habitual, ya que la mayoría del contenido web es textual. Si nos fijamos, en el segundo parámetro estamos indicando que el MediaType (formato de datos) es de tipo JSON (`APPLICATION_JSON`). Existen otros MediaType como `APPLICATION_XML`, `TEXT_PLAIN`, `IMAGE_PNG`, `VIDEO_MP4`, por poner algunos ejemplos. Esto le indicará al navegador del cliente cómo debe gestionar/visualizar el contenido que se le envía.

Para tratar adecuadamente con las representaciones necesitamos importar los siguientes paquetes:

```
import org.restlet.representation.EmptyRepresentation;
import org.restlet.representation.Representation;
import org.restlet.representation.StringRepresentation;
```

Para tratar adecuadamente con los distintos formatos de datos necesitamos importar los siguientes paquetes:

```
import org.restlet.data.MediaType;
import org.restlet.data.Status;
```

Para tratar adecuadamente con el formato JSON necesitamos importar los siguientes paquetes:

```
import org.json.JSONException;
import org.json.JSONObject;
```

Por otro lado, en el primer parámetro lo que se está haciendo es 'serializar' una ToDoList para obtener sus datos en formato JSON, y pasarlo a texto (que es lo que requiere la `StringRepresentation`). Este serializador (desarrollado por nosotros) no es necesario, y RESTlet no obliga a su implementación, pero es una solución bastante útil para separar responsabilidades y simplificar la implementación del método. Este serializador (asociado a un recurso), podría implementar diferentes representaciones (getAsXML, getAsPlainText, getAsImage,...), sin obligar a que el recurso sepa el formato de salida.

Para poder usar adecuadamente el serializador que hemos comentado necesitamos importar el siguiente paquete:

```
import isc.rest.todolist.serializers.ToDoListResourceSerializer;
```

Por último para poder implementar adecuadamente las operaciones GET y POST deberíamos importar los siguientes paquetes:

```
import org.restlet.resource.Get;
import org.restlet.resource.Post;
```

Ejercicio 13: Implementar el serializador ToDoListResourceSerializer

Se creará un paquete nuevo, isc.rest.todolist.serializers donde se construirá la representación que se le ha asignado a una ToDoList. Requerirá importar y añadir al Build Path la librería JSON (java-json.jar), siguiendo el mismo procedimiento llevado a cabo en los ejercicios 3 y 7.

```
import iap.ToDoList.Lista;
import iap.ToDoList.ToDoList;

import org.json.JSONArray;
import org.json.JSONException;
import org.json.JSONObject;

public class ToDoListResourceSerializer {

	public static JSONObject getAsJSON(ToDoList tdl) {

		JSONObject tdl_json = new JSONObject();

		try {

			tdl_json.put("link", "/listas");

			JSONArray listas_json = new JSONArray();

			JSONObject lista_json = null;
			for (Lista l : tdl.getListas()) {
				lista_json = new JSONObject();
				lista_json.put("id", l.getID());
				lista_json.put("link", "/lista/" + l.getID());
				listas_json.put(lista_json);
			}
			tdl_json.put("listas", listas_json);

		} catch (JSONException e) {
			e.printStackTrace();
		}

		return tdl_json;
	}
}
```

Llegados a este punto, podemos hacer la primera prueba y consumir este servicio desde un navegador. Más adelante se muestra un ejemplo en la sección Ejemplos de peticiones sobre el API REST de la ToDoList.

IMPLEMENTACIÓN DE UN @POST

El método @Post se utiliza para crear nuevos datos sobre un recurso. Este método tiene una sintaxis diferente a la del @Get, ya que este método recibe un parámetro Representation en el que podremos obtener el Payload que nos envíen.

Para el caso de la ToDoList, y asociado al recurso ToDoListResource, con este método se podrán añadir nuevas listas a la ToDoList. Como resultado, nos mostrará la ToDoList con la nueva lista introducida.

Una característica que diferencia a un POST de un GET es justamente el Payload, que se utiliza para enviar datos. En el caso de la aplicación de la ToDoList se ha decidido que el formato de estos datos sea JSON. Por tanto, para crear una nueva lista, será necesario indicar el id y el nombre de la lista, codificado en JSON como:

```
{ "id" : "identificador-lista",   "nombre" : "nombre-lista"      }
```

El método POST deberá primero recorrer estos datos, procesarlos, crear la lista, y finalmente devolver la representación (en JSON también) de la ToDoList.

Ejercicio 14: Implementar la operacion @Post del recurso ToDoListResource.

```
@Post
public Representation postToDoListJSON(Representation entity) {

	// Procesamos el Payload de entrada
	JSONObject json = null;
	String id = null;
	String nombre = null;
	try {
		json = new JSONObject(entity.getText());
		id = json.getString("id");
		nombre = json.getString("nombre");
		if (id == null || id.equalsIgnoreCase("") ||
		    nombre == null || nombre.equalsIgnoreCase("")) {
		      setStatus(Status.CLIENT_ERROR_BAD_REQUEST);
		      return new EmptyRepresentation();
		}
	} catch (JSONException e) {
		setStatus(Status.CLIENT_ERROR_UNPROCESSABLE_ENTITY);
		return new StringRepresentation("JSON format error!",
		                        MediaType.TEXT_PLAIN);
	} catch (IOException e) {
		setStatus(Status.CLIENT_ERROR_UNPROCESSABLE_ENTITY);
		return new StringRepresentation("JSON format error!",
		                        MediaType.TEXT_PLAIN);
	}

	// Si todo ha ido bien, podemos crear la lista
	this.getToDoList().addLista(id, nombre);

	return new StringRepresentation(
	                ToDoListResourceSerializer.
	                getAsJSON(this.getToDoList()).toString(),
	            MediaType.APPLICATION_JSON);

}
```

Probar a crear una nueva lista 'invitados' (Invitados a la fiesta), tal y como se explica en la sección Petición POST: ejecutar una operación sobre un recurso.

EJERCICIOS PARA COMPLETAR EL PROYECTO

Para terminar por completo la práctica, sería necesario desarrollar las siguientes partes:

- Implementar las operaciones @Delete y @OPTIONS (de esta opción proporcionamos el código fuente Java en PoliformaT) de la ToDoList.
- Crear los recursos ListaResource y EntradaListaResource, y sus respectivos ListaResourceSerializer y EntradaListaResourceSerializer. Facilitaremos el código fuente Java de los serializadores en PoliformaT
- Implementar las operaciones asociadas a cada recurso (Get/Post/Put/Delete)
- Extender la RESTToDoListApplication para introducir nuevos routers a los recursos ListaResource y EntradaListaResource.

Un último apunte: las URLs que identifican tanto a una Lista como a una EntradaLista están parametrizadas, según lo visto en la sección RECURSOS E IDENTIFICADORES. Así, por ejemplo, el router para una Lista en la RESTToDoListApplication, sería:

```
@Override
public Restlet createInboundRoot() {

      Router router = new Router(getContext());

      // Define routers for users
      router.attach("/listas", ToDoListResource.class);
      router.attach("/lista/{LISTA}", ListaResource.class);
      return router;
}
```

Esto indica que "LISTA" es un parámetro que se recibirá en las llamadas. Este parámetro se usará en el recurso ListaResource, y se puede obtener con el siguiente código:

```
(String) getRequest().getAttributes().get("LISTA");
```

Ejemplos de peticiones sobre el API REST de la ToDoList

A continuación se muestran algunos ejemplos de invocaciones de servicios sobre el API REST de la ToDoList implementada.

Petición GET: obtener información de la ToDoList

La imagen de la Fig. 14 muestra las cabeceras (todo ha ido OK, Status Code 200) de la petición GET realizada sobre la URL

`http://localhost:8182/listas`

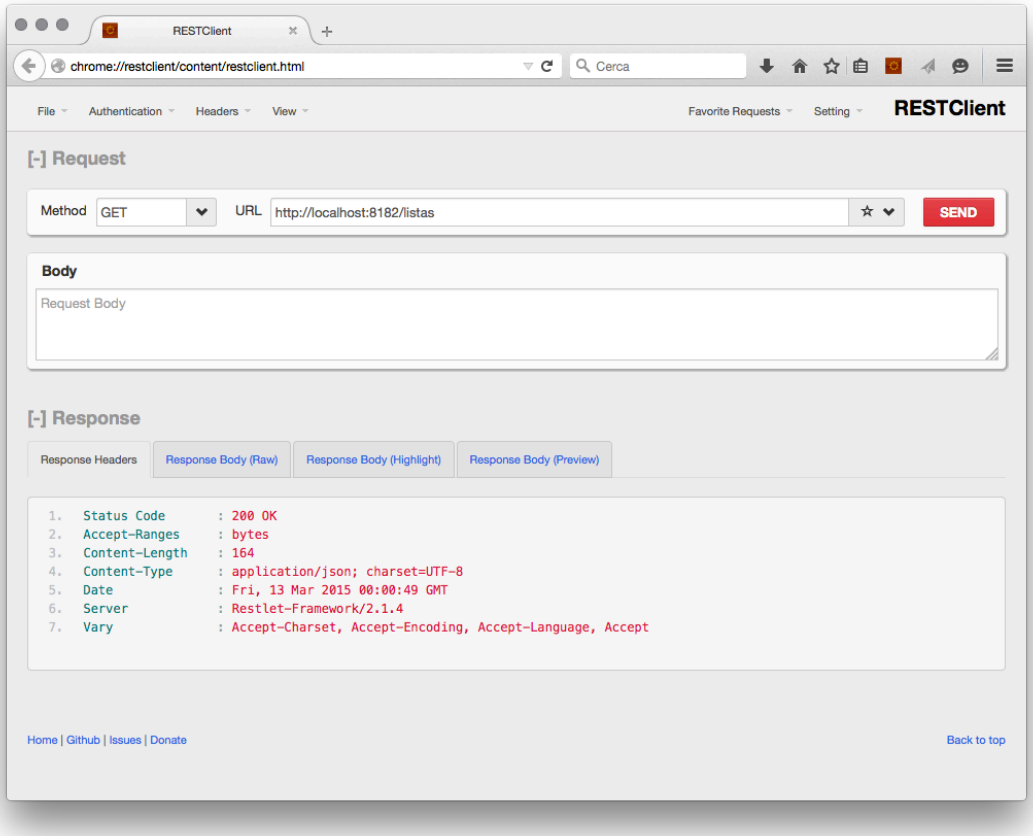

Figura 14. Llamada a GET funcionando correctamente

La Fig. 15 muestra el cuerpo de la petición anterior, donde se puede ver el resultado en JSON (tal y como se ha indicado en las Headers), obteniendo como resultado información sobre el recurso solicitado. En este caso, el recurso es una ToDoList, de la que vemos su link, e información resumida sobre las diferentes listas que la componen y su link.

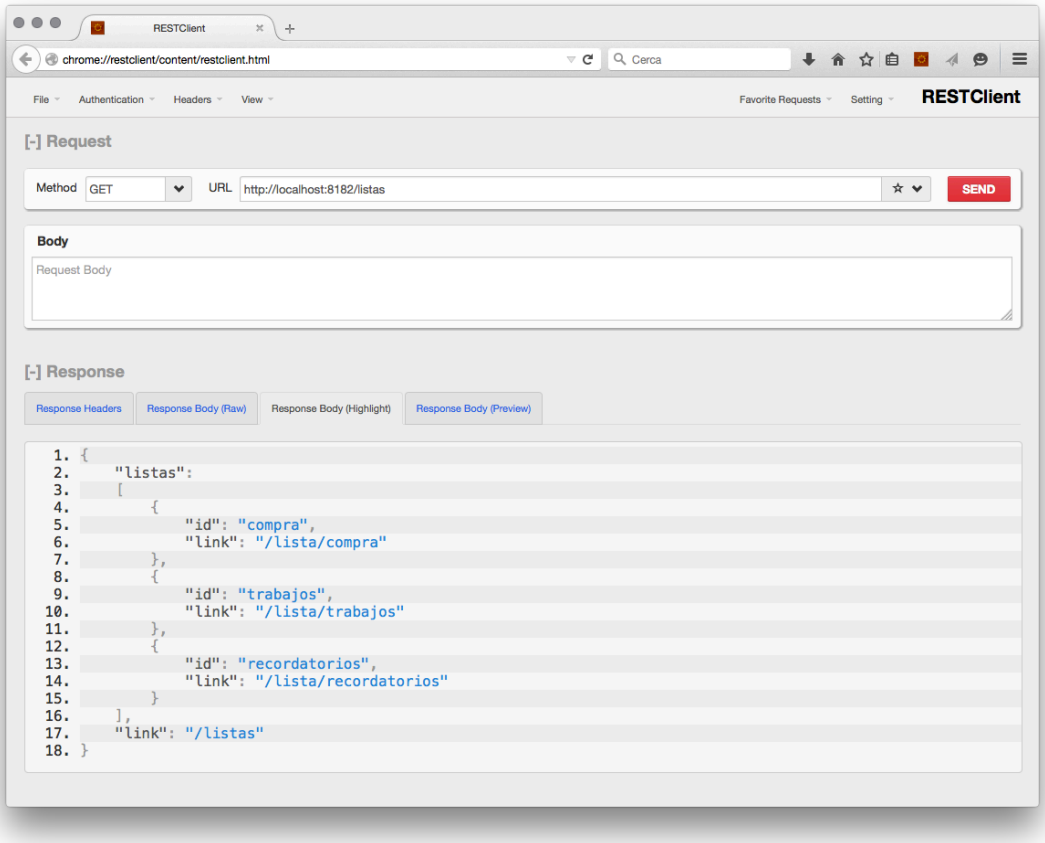

Figura 15. Resultado de la llamada en formato JSON

Petición POST: ejecutar una operación sobre un recurso

La figura 16 muestra las cabeceras (todo ha ido OK, Status Code 200) de la petición POST realizada sobre la URL

```
http://localhost:8182/listas
```

, con PAYLOAD

```
{"id": "invitados", "nombre":"Invitados a la fiesta"}
```

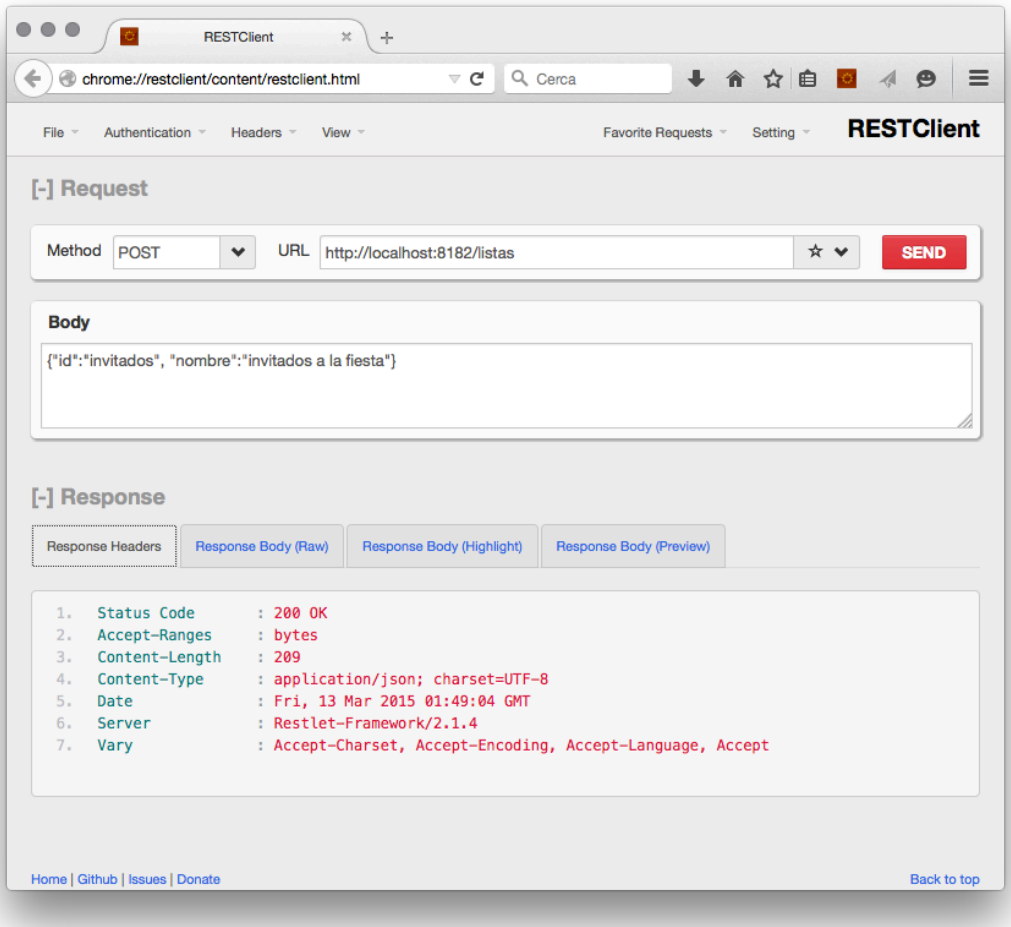

Figura 16. Llamada a POST

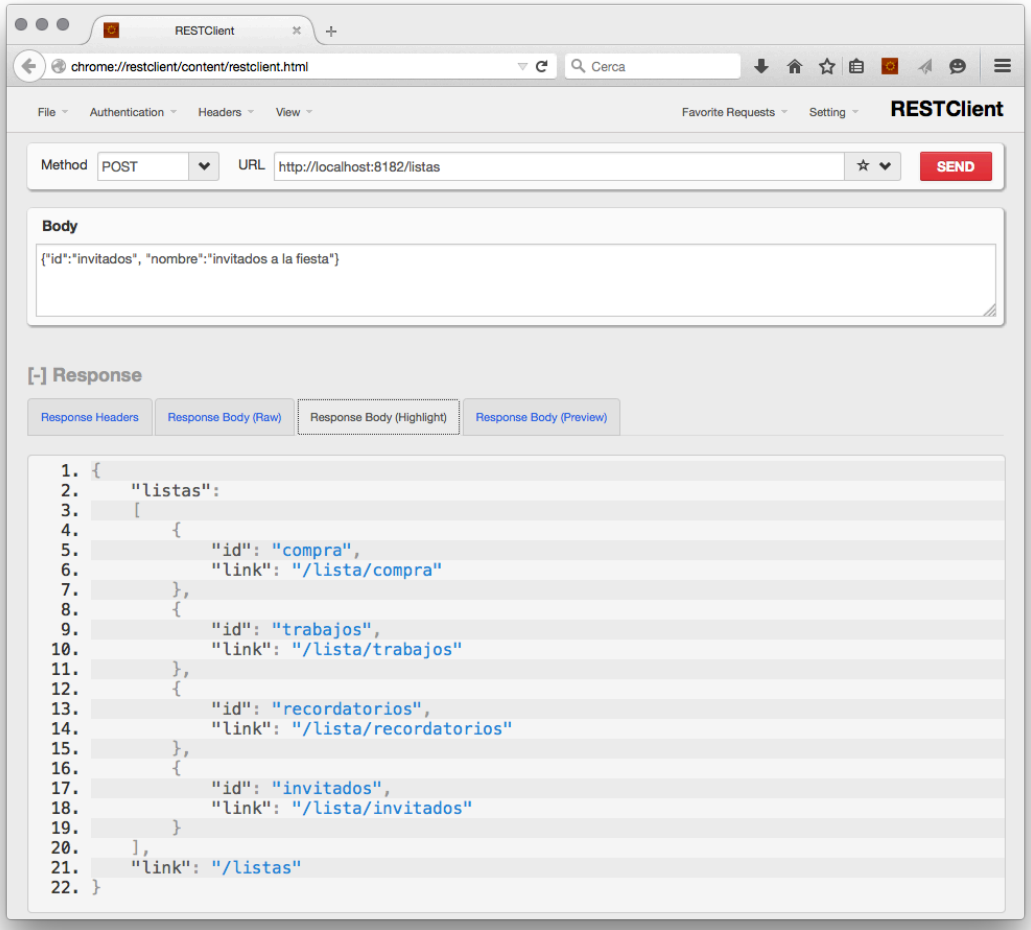

Figura 17. Resultado del POST