Desarrollo Basado en Componente y Servicios

Servicios Web de tipo REST

Parte I: Conceptos Básicos

Introducción

Como en el caso de los servicios web basados en SOAP, los conceptos teóricos han sido vistos en teoría. Aquí nos vamos a centrar en cuestiones prácticas.

El objetivo aquí es doble. Por un lado ayudar a la comprensión de cómo funcionan los servicios web de tipo REST de manera práctica, es decir, apoyar a lo visto en teoría. Por otro lado, construir y consumir servicios básicos usando tecnología JEE.

No hay una manera única de realizar la tarea indicada. Hay muchos frameworks que nos permitirían abordar la creación y consumo de servicios web RESTful de manera muy sencilla, pero que ocultarían detalles importantes de implementación. No los vamos a usar aquí y la razón es porque el objetivo principal NO es construir servicios web, si no aprender cómo funcionan de manera práctica.

Sí nos vamos a apoyar, obviamente, en las herramientas que proporciona Netbeans y los APIs de JEE (JAX-RS - JSR 311), para facilitar el trabajo, pero muchas cosas las haremos “a mano”, por la razón indicada. En el mismo sentido, también vamos a apoyarnos en el framework Jersey[1] (desarrollado por Oracle para facilitar el acceso a determinadas funcionalidades de JAX-RS) para simplificar la creación tanto del servicio como del cliente.

Aunque el contenido a tratar no es muy extenso, sólo abordamos los conceptos más importantes relacionados con servicios web RESTful, para una mayor claridad, se ha dividido en varias partes.

Objetivos de esta parte

- Mejorar la comprensión de los principios básicos de funcionamiento de los servicios RESTful
- Construir servicios RESTful simples
- Construir consumidores de servicios RESTful

Existe abundante documentación on-line acerca de cómo crear y usar servicios web de tipo REST con Netbeans (Ej. https://netbeans.org/kb/docs/websvc/rest.html), que se aconseja consultar para ampliar la pequeña introducción que aquí mostraremos

[1] https://jersey.java.net/

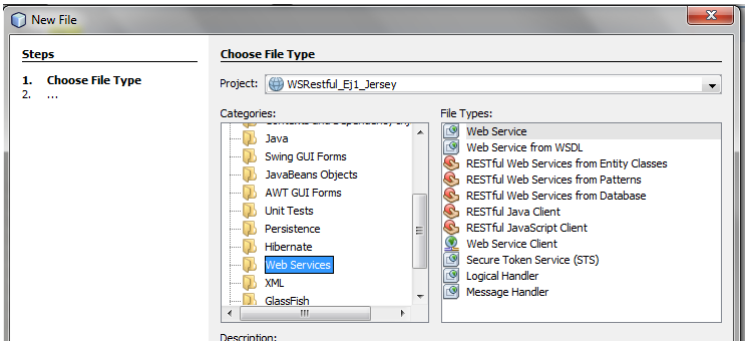
Creación de un servicio RESTful básico

1. Creando el contenedor

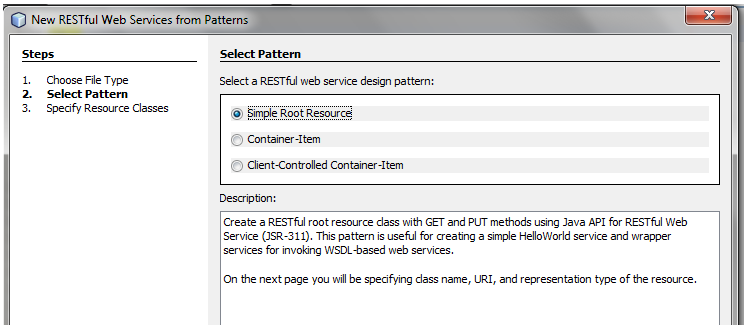
En este caso el servicio web debe estar contenido en una aplicación web. Crearla. Se aconseja crear uno o varios paquetes donde agrupar las clases que crearemos a continuación; no es aconsejable usar “default package”

2. Creando el servicio web.

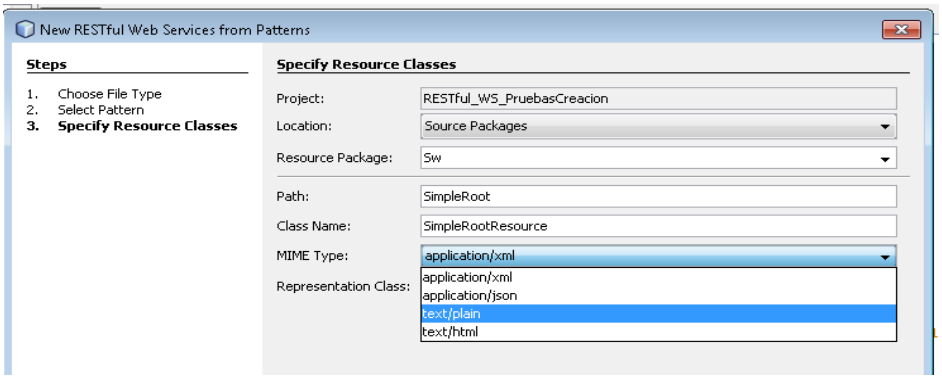
Al seleccionar en “insertar nuevo”, categoría “Web Services”, vemos que hay varias opciones de servicios web tipo REST que Netbeans genera automáticamente. Como vamos a crear un caso simple, seleccionar “from Patterns”.



En el siguiente formulario vamos a seleccionar la opción “Simple Root Resource”; ver la breve descripción de lo que vamos a crear.



El siguiente formulario es más complejo. Podemos escoger lo siguiente:



- El paquete donde crear el servicio web.

- **Path**: La ventaja de los servicios web de tipo REST, y que los hace más ligeros, es que la comunicación cliente-servidor está basada directamente en el protocolo HTTP. Admiten, por lo tanto, operaciones, por ejemplo, de tipo GET, POST, PUT o DELETE. De la misma forma, la

identificación de recursos se realiza mediante la URI correspondiente (no mediante métodos como en los SOAP), y es aquí donde interviene esta parte del formulario: el valor de PATH será la identificación en la URI del servicio web creado. La URI de un servicio web creado y desplegado en un servidor glassfish local será:

localhost:8080/*context-root*/*webresources*/*Path*

Donde *context-root* es la raíz de la URI definida en el proyecto web (esto es configurable) y *webresource* es la parte de la ruta definida en el argumento a la anotación *@javax.ws.rs.ApplicationPath("webrseurce")* (veremos esto un poco más adelante, de momento dejarlo así). Como servicio basado en HTTP si se pone esa URI en un navegador, el servicio web generará una respuesta, que será el valor retornado por el método que implemente la operación GET (lo veremos más adelante).

En definitiva, dar a ese campo el valor que queramos que aparezca en la URI asociado a nuestro servicio web.

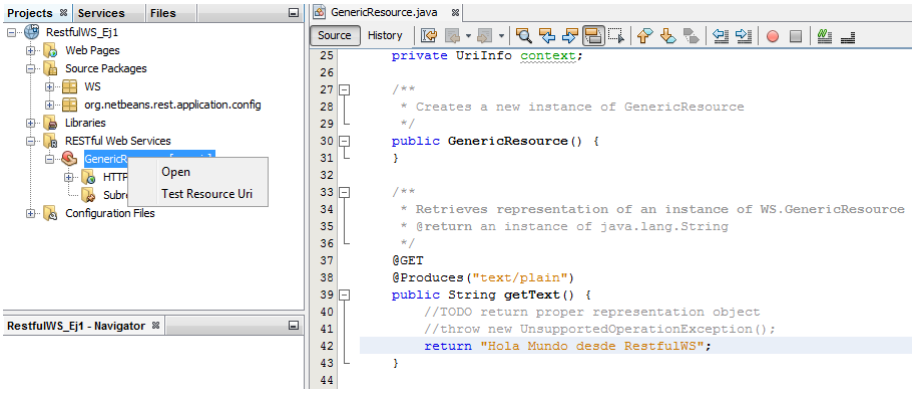
- **Class Name**: nombre de la clase que implementa el servicio web.

- **MIME type:** formato MIME de la información intercambiada entre cliente y servidor en las peticiones y respuestas del servicio web. Se le pasará como argumento a las anotaciones @Produces y @Consumes. Se puede cambiar luego en el código. Formatos MIME más importantes: test/html, test/plain, application/xml (nos va a permitir intercambiar objetos, por ejemplo Entities, convertidos a XML), application/json y application/octet-stream (para cualquier tipo de información en binario). Hay muchos más. Se deja al alumnos la ampliación de esta parte

- **Representation Class**: tipo de clase (objeto) que será pasado como argumento a las operaciones PUT y POST (operaciones que permiten enviar datos de cliente a servidor. Se aconseja dejar el de por defecto (String), pero se puede cambiar. Se puede cambiar luego en el código, también.

Ya hemos acabado. Analizar el código creado. Se han creado las funcionalidades básicas GET y PUT para desde un cliente obtener datos del servicio web (GET) y enviar datos (PUT).

IMPORTANTE: ANTES DE COMPILAR Y DESPLEGAR CAMBIAR EL CONTENIDO DEL MÉTODO getText(). ELIMINAR Y COMENTAR EL ENVÍO DE LA EXCEPCIÓN (SI NO VA A DAR, OBVIAMENTE, UN ERROR) POR UN RETURN DE UNA CADENA DE TEXTO (en la figura se tiene un ejemplo).



Ya se puede ya compilar y desplegar.

AVISO IMPORTANTE: Para asociar la clase creada a un servicio web, debe estar implementada la clase javax.ws.rs.core.Application y retornar una instancia a cada una de las clases que contienen un servicio web. Esto se crea automáticamente al seguir los pasos indicados (comprobarlo) en una clase llamada "ApplicationConfig.java". Según vayamos añadiendo servicios, esta clase es actualizada automáticamente por Netbeans. En la siguiente figura se puede ver un ejemplo del contenido de esta clase, para los ejemplos que se van a mostrar en los apuntes sobre servicios web RESTful

```
@javax.ws.rs.ApplicationPath("webresources")
public class ApplicationConfig extends Application {

    @Override
    public Set<Class<?>> getClasses() {
        Set<Class<?>> resources = new java.util.HashSet<>();
        addRestResourceClasses(resources);
        return resources;
    }

    /**
     * Do not modify addRestResourceClasses() method.
     * It is automatically populated with
     * all resources defined in the project.
     * If required, comment out calling this method in getClasses().
     */
    private void addRestResourceClasses(Set<Class<?>> resources) {
        resources.add(WSs.Prueba1.class);
        resources.add(WsEjbBasado.EjbAsRestWs.class);
        resources.add(WsEjbBasado.service.BodegaFacadeREST.class);
        resources.add(WsEjbBasado.service.CategoriaFacadeREST.class);
        resources.add(WsEjbBasado.service.DenominacionOrigenFacadeREST.class);
        resources.add(WsEjbBasado.service.VinoFacadeREST.class);
    }

}
```

Se puede ver la anotación *@javax.ws.rs.ApplicationPath()* cuyo argumento indica la parte de la URL que sigue al "context root" y precede a lo indicado en la anotación *@Path* de la clase que implementa el servicio web, es decir, de alguna manera, la ruta en el servidor donde se despliega nuestro servicio web.

Una alternativa a esa clase sería añadir al POJO que implemente cada uno de los servicios web la anotación @javax.ws.rs.ApplicationPath() y hacer que extiendan la clase javax.ws.rs.core.Application. Por ejemplo:

```
@javax.ws.rs.ApplicationPath("recursos")
@Path("/miServicioWeb")
public class servicioWeb extends Application {

    @GET
    @Produces("text/plain")
    public String prueba() {
        return "Hola desde servicio web 2";
    }

}
```

En el caso de la figura anterior la URL del servicio será:

localhost:8080/*context-root*/recursos/miServicioWeb

Para que el servicio web implementado sea accesible se debe realizar lo indicado. Si al crear un servicio web da problemas de que no se encuentra, comprobar esto. Si se crea una nueva aplicación y se copia y pega la clase "ApplicationConfig.java" con refactorización, Netbeans las actualiza automáticamente a las clases de ese nuevo proyecto.

3. Probando el servicio web

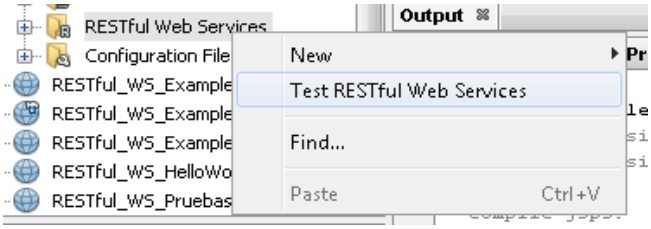
Vamos a empezar probándole sin usar un cliente. El objetivo es reforzar la idea y comprender mejor como los servicios web RESTful usan el protocolo HTTP sin artificios para la especificación del servicio. Ver como este servicio va en la operación HTTP y la URI. Por eso vamos a probar el servicio usando las URIs directamente desde navegador y ver cómo responde el servidor.

Para probar la operación GET, en principio, es tan fácil como poner en el navegador la URI asociada (recordemos: localhost:8080/*context-root*/webresources/*Path*) y directamente en el navegador tenemos que ver la respuesta en el formato y con el contenido (lo que pongamos en return) definido en el servicio creado. Esto se puede hacer desde Netbeans, haciendo clic con el botón derecho en el servicio web creado (aparece automáticamente en el proyecto al ser compilado) y seleccionando la opción "TestResource Uri".

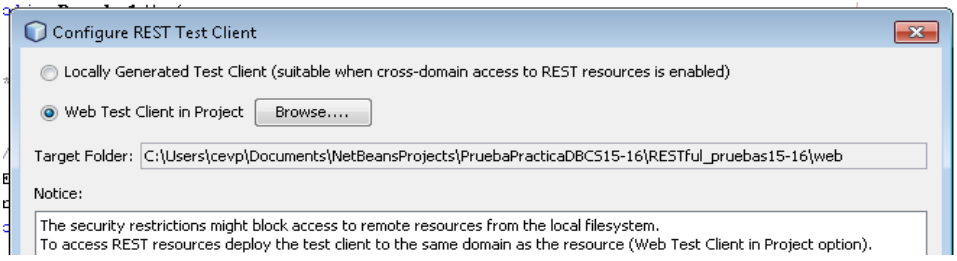
Sin embargo, no es posible probar así las operaciones PUT y POST, ya que hay que añadir el contenido del cuerpo a la petición. Para ello vamos a usar herramientas que nos permiten crear la petición del servicio (REQUEST) y ver la respuesta del servidor (RESPONSE). Cuando usemos un cliente, estos detalles se ocultan.

Hay muchas alternativas. Aquí se proponen las siguientes:

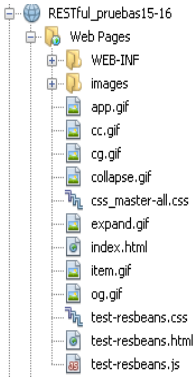
- Usar las herramientas que proporciona Netbeans. Mediante el menú contextual asociado al servicio web podemos, como en el caso de SOAP, probarlo en todas sus operaciones.



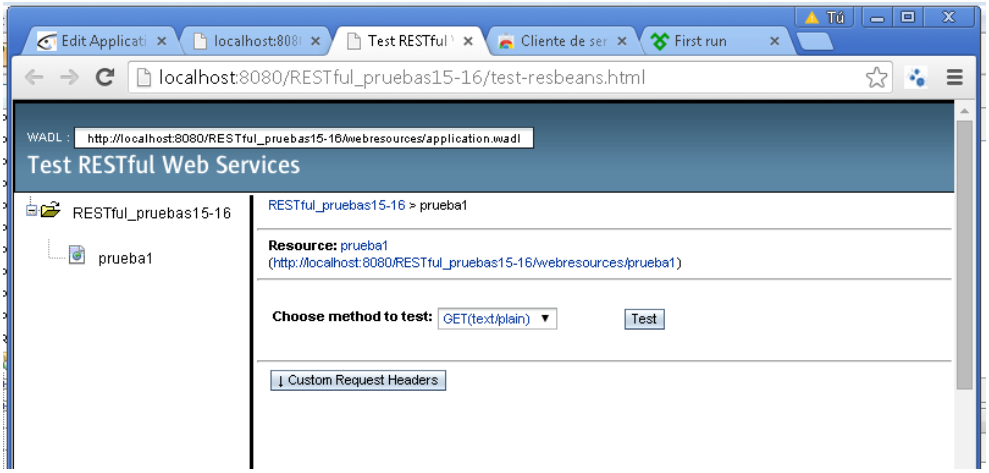
Para probarlos desde un navegador, escoger la opción "Web Test Client in Project"



Esto creará todos los ficheros de la aplicación web que nos permitirá probar el servicio creado

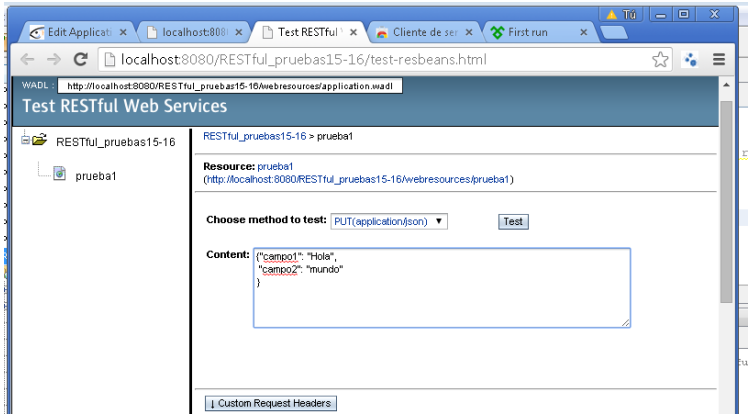


Se abrirá en el navegador la aplicación web que nos permitirá probar todas las opciones implementadas en nuestro servicio:



Ahí se pueden probar todas las operaciones, dar valor a las cabeceras de la petición HTTP (request), ver la respuesta y en las operaciones PUT y POST dar valor al cuerpo de la petición, cuerpo que será pasado como argumento al correspondiente método del servicio creado, por lo tanto tendrá que ser del tipo especificado en la notación @Consumes. Ej. :

```
 */
@PUT
//@Consumes("text/plain")
@Consumes("application/json")
public void putText(String content) {
    System.out.println("Argumento: " + content);
}
```

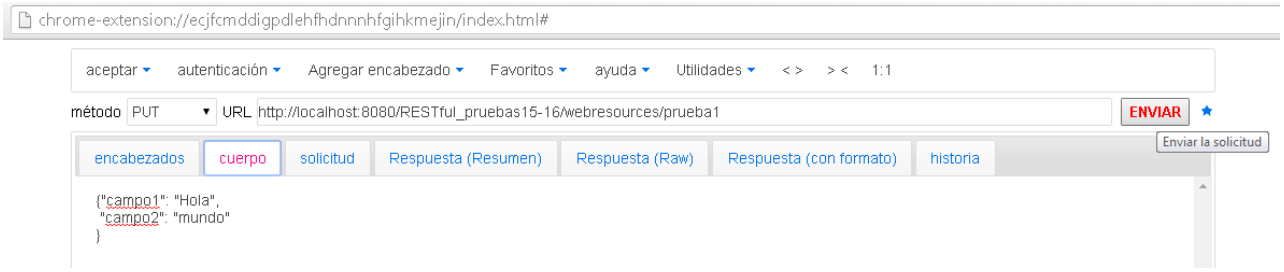


Tras hacer clic en "Test" en la consola de log del servidor obtenemos (recordar que estamos ejecutando la operación PUT, por lo tanto no hay respuesta del servidor al cliente, por eso usamos la consola del servidor para probar que la operación se ha realizado):

```
Información:   RESTful_pruebas15-16 was successfully deployed in 884 milliseconds.
Información:   Argumento: {"campo1": "Hola",
 "campo2": "mundo"
}
```

El problema de esta opción es el código que crea, innecesario para la aplicación final que consumirá el servicio mediante un cliente. Para no tener que crear este código se pueden utilizar las siguientes dos opciones, independientes de Netbeans.

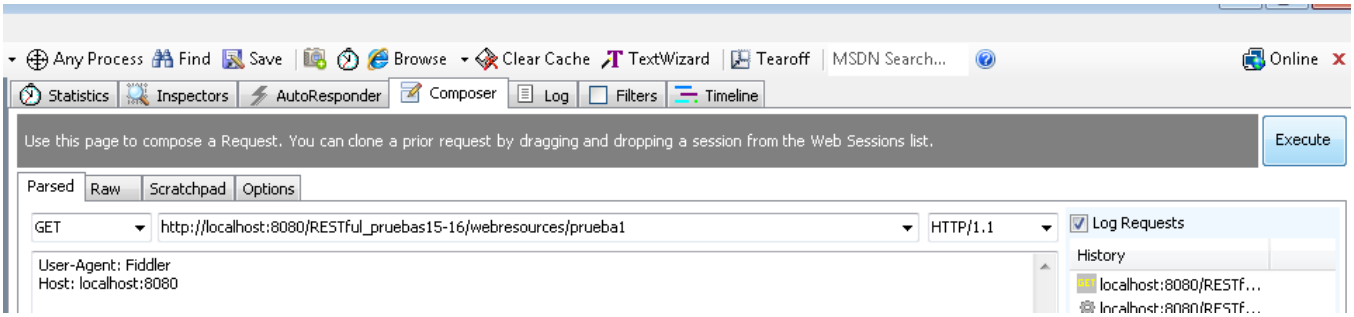
- Usar el complemento del navegador Chrome “Cliente de servicios Web RESTful” (https://chrome.google.com/webstore/detail/rest-web-service-client/ecjfcmddigpdlehfhdnnnhfgihkmejin). Para Firefox se tiene el complemento “RESTClient” (https://addons.mozilla.org/es/firefox/addon/restclient/). Aquí vamos a ver ejemplos del complemento Chrome (el de Firefox es similar).



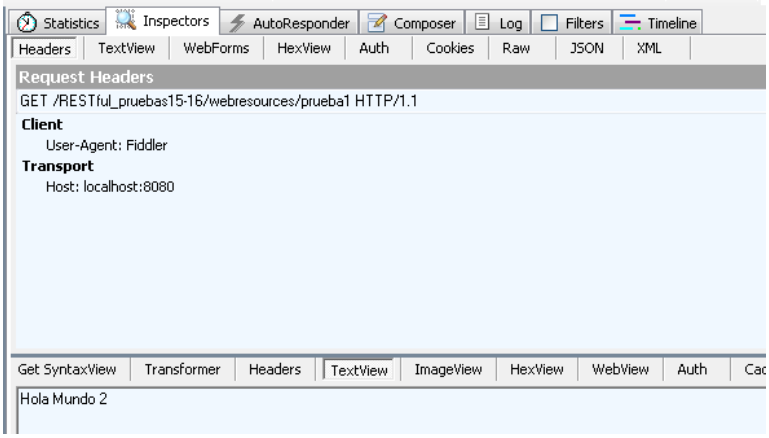
Permite como en el caso anterior, fijar las cabeceras y el cuerpo del mensaje, así como ver la respuesta del servidor, en caso de que la haya. El problema es que en la solicitud sólo se aceptan contenidos de tipo JSON.

- Usar la herramienta de escritorio gratuita “fiddler” (http://www.telerik.com/fiddler). Desarrollada por la compañía Telerik, se puede descargar y usar de manera gratuita.

Ejemplo de prueba de la operación GET



Visualización de la salida en la aplicación



La herramienta tiene una gran cantidad de funcionalidades relacionadas con la inspección del tráfico web. Por contra, se depende del desarrollo de una empresa.

Otras alternativas de tipo libre serían: NetTool (http://sourceforge.net/projects/nettool/), Wireshark (https://www.wireshark.org/), …

Ejercicio 1. Crear un servicio web como el mostrado en el ejemplo. Se puede usar la herramienta proporcionada por Netbeans o se puede crear la clase y añadir "a mano" las anotaciones. Hacer que las operaciones GET y PUT consuman y produzcan texto plano. Probar PUT mandando un mensaje al servidor mediante System.out.println, por ejemplo. Compilar y desplegar la aplicación. Probar las operaciones de cualquiera de las formas indicadas en el apartado 3 "Probando el servicio web".

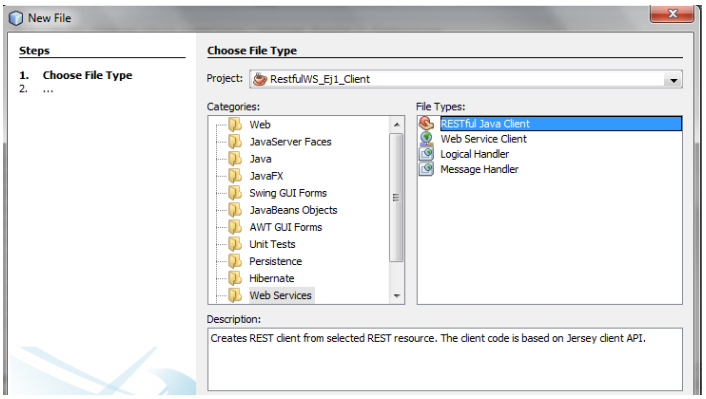
Ejercicio 2. Cambiar el tipo de información consumida por PUT a JSON, por ejemplo (se puede hacer que consuma ambas, y en cabecera de la petición indicar cuál se manda de la siguiente manera: *@Consumes({"text/plain", "application/json"})*). Crear, también, un nuevo servicio para GET que ahora devuelva (produzca) HTML. Recordar que habrá que crear una nueva URI, más concretamente, con la anotación @Path (Ej. @Path("/getHtml")) en el método, se añadirá un nuevo elemento a la URI base, la creada con la anotación @Path asociada a la clase (localhost:8080/*context-root*/webresources/*Path*/**getHtml**). Volver a probar el servicio.

Ejercicio 3. Añadir una operación POST. El tipo de dato a consumir da lo mismo. Probarle.

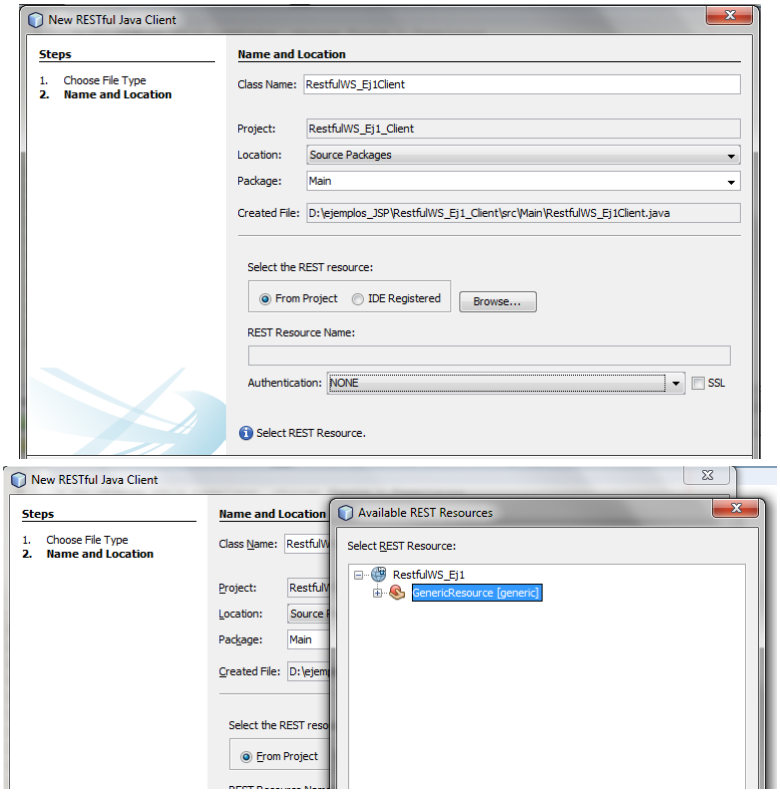
4. Creando un cliente JSE

Probamos esta opción por simplicidad. De igual manera se podría realizar un cliente Web o JEE.

Tras crear el contenedor (proyecto de tipo JSE), añadir un fichero "RESTful Java Client"



En el formulario siguiente se asigna el nombre a la clase y el paquete, seleccionando el recurso REST



Se crea la clase que nos permite acceder a los recursos del servicio web. En la figura siguiente se puede ver un ejemplo:

```
public class ClienteServicioWeb {
    private WebTarget webTarget;
    private Client client;
    private static final String BASE_URI = "http://localhost:8080/RESTful_pruebas15-16/webresources";

    public ClienteServicioWeb() {
        client = javax.ws.rs.client.ClientBuilder.newClient();
        webTarget = client.target(BASE_URI).path("prueba1");
    }

    public String getText() throws ClientErrorException {
        WebTarget resource = webTarget;
        return resource.request(javax.ws.rs.core.MediaType.TEXT_PLAIN).get(String.class);
    }

    public void putText(Object requestEntity) throws ClientErrorException {
        webTarget.request(javax.ws.rs.core.MediaType.TEXT_PLAIN).put(javax.ws.rs.client.Entity.entity(
    }

    public void editText() throws ClientErrorException {
        webTarget.request().post(null);
    }

    public String getTextHtml() throws ClientErrorException {
        WebTarget resource = webTarget;
        resource = resource.path("getTextHtml");
```

Las clases más importantes para poder consumir un servicio web RESTful son:

- La interface javax.ws.rs.client.Client, que es principal punto de entrada para realizar peticiones y consumir las respuestas. Como se especifica en la documentación de la clase de Oracle, es un objeto “pesado”, es decir, consume muchos recursos, por lo que no se deben usar muchos y se aconseja cerrarlos en cuanto no sean necesarios
- Interface javax.ws.rs.client.WebTarget. Mediante la clase Client, especificamos el recurso web a acceder, retornando un objeto de tipo WebTarget; es la “referencia” al recurso. Proporciona los metodos que permiten configurar la petición (elementos adicionales a la URI base) y realizar la petición (request) del recurso.

Ya podemos probarlo ejecutando los métodos getText() y putText() de esta clase. Ejemplo:

```
public class main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        ClienteServicioWeb servicio = new ClienteServicioWeb();

        // Ejecuto get
        String respuesta = servicio.getText();
        System.out.println("Respuesta del SW: " + respuesta);

        // Ejecuto put
        servicio.putText("Mensaje desde el cliente");
    }

}
```

Ejercicio 4. Crear un cliente, analizar el código creado y probar el servicio web creado en los ejercicios 1, 2 y 3. De momento probar sólo la opción de intercambiar Strings, que es la opción más sencilla).