Webservices and remote invocation of business objects

Rui Couto António Nestor Ribeiro

May 4, 2020

Contents

1 Introduction

This tutorial intends to refine the application developed so far by providing interoperability support. Such is achieved by the implementation of webservices in the Game Management System. The objective of this session is to present the process to implement both a RESTfull and a WSDL webservice to provide data to external applications.

2 GMS Facade

In this tutorial is proposed to provide, as webservices, the following functionalities:

1. List all games;

2. Search for a game;

As with previous tutorials, provided instructions are valid for NetBeans.

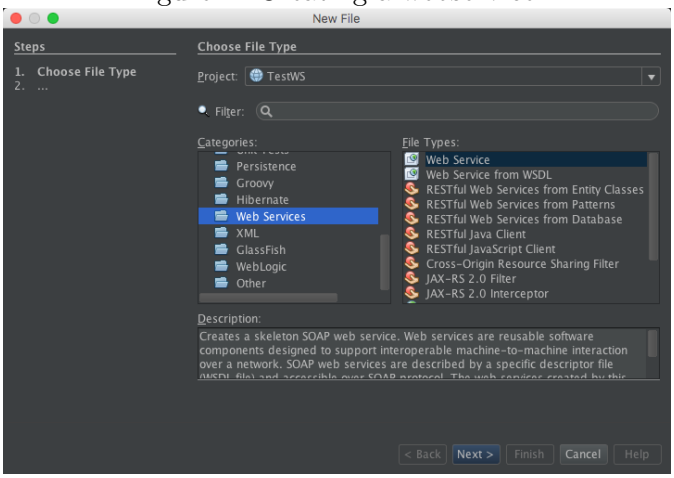
3 WSDL webservice

First, the creation of a WSDL SOAP webservice will be addressed.

3.1 Creating the webservice

Netbeans provides support for the creation of webservices, reducing the need to write code and deal with requests encapsulation. To create a webservice, right clicking a web application, and next `New > Other...` (see Figure 1).

Figure 1: Creating a webservice.



In the following window, it is possible to define the webservice name and end the process. A default method is presented in the class.

```
1 @WebService(serviceName = "MyWebservice")
2 public class MyWebservice {
3     @WebMethod(operationName = "hello")
4     public String hello(@WebParam(name = "name") String txt) {
5         return "Hello " + txt + " !";
6     }
7 }
```

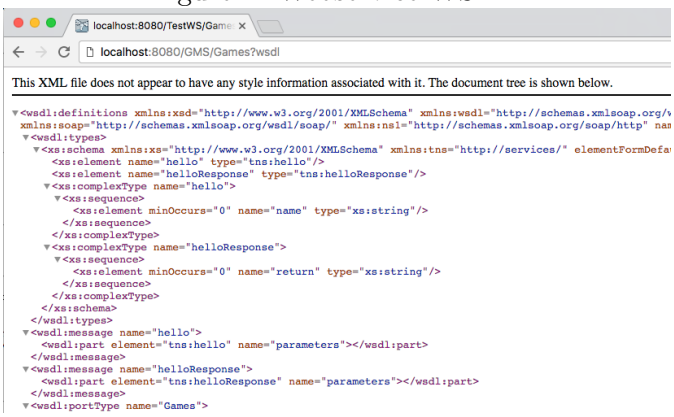
Note that line 1 is the declaration of the webservice name (`@WebService`), and line 3 is the declaration of a webservice method (`@WebMethod`). Webservice methods define also the name of the parameters (line 4, `@WebParam`).

After running the project, the new webservice should be available for use.

3.2 Testing the service

The webservice can be tested in several ways. The most direct one is to access its WSDL. To do so, access the URL corresponding to your webservice endpoint, for instance, `http://localhost:8080/GMS/Games?wsdl`. This should return the corresponding WSDL (c.f. Figure 2).

Figure 2: Webservice WSDL.



Another possibility is to resort to a third party application for testing the service itself, as is the case of SOAP UI[1]. These applications interpret the WSDL and provide an interface to interact with the methods.

Implementing an application to call the webservice is another possibility, as we will explore in the next section.

Tasks

1. Create a new webservice[2].
2. Implement the method to retrieve the list of existing games.
3. Implement the method to search for a game, for a given name.
4. Check the generated WSDL, in order to ensure that the service was deployed.

3.3 Invoking the webservice

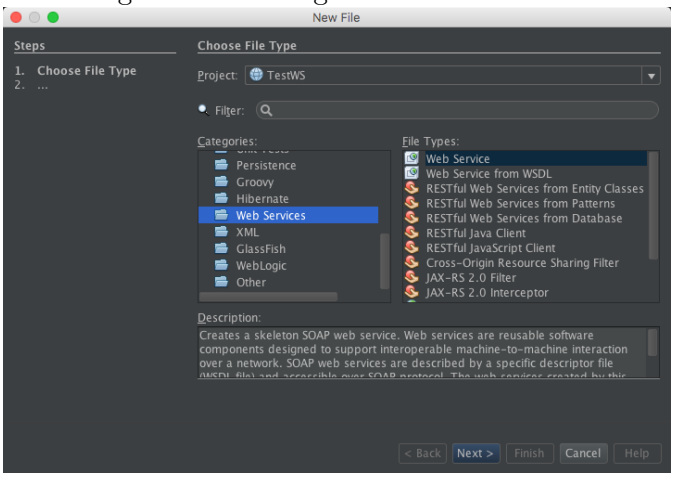
criar uma aplicação nova, cliente

NetBeans helps also in the invocation of the webservice. In an existing application, right clicking it, and then `New > Other`, `Web Services`, and finally `Web Service Client` presents an interface to configure a new webservice (c.f. Figure 3).

[1] `https://www.soapui.org/`

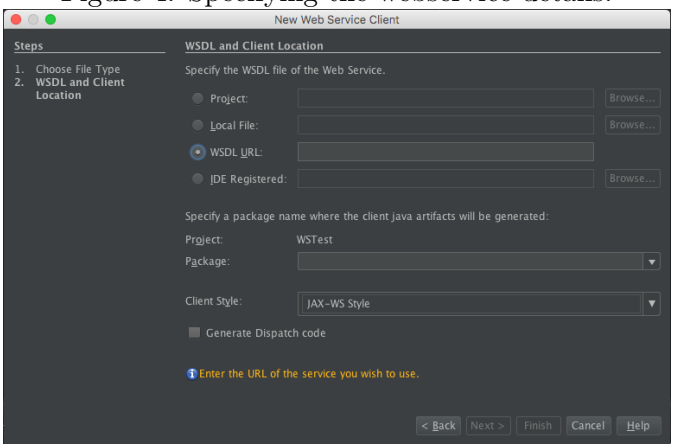
[2] it is suggested to create a new package for webservices.

Figure 3: Creating a webservice client.



In the presented window, the WSDL can be specified (c.f. Figure 4). Once `Finish` is selected, the corresponding source code to use the webservice is generated (notice a new folder under the project structure, `Generated Sources`).

Figure 4: Specifying the webservice details.



Now, the code to invoke the webservice can be added. See the following example. In line 1 the webservice port is instantiated. Next, webservice methods can be invoked as regular Java methods (c.f. line 2).

```
1 MyWebservice g = new MyWebservice_Service().getMyWebservicePort();
2 String r = g.hello("name");
3 System.out.println(r);
```

Finally, the code can be run. A `Clean and Build` is required prior to the first invocation, in order to compile the automatically generated sources.

Tasks

1. Create a new application to test the webservice.
2. Add the previously developed webservice to the application, and invoke it.

4 REST webservices

This section addresses the creation of REST webservices. Unlike the previous ones they rely on a simpler syntax. However, since they are not standardised sometimes they can be quite laborious to implement.

4.1 Creating the webservice

The implementation of a REST webservice will rely on a servlet. So, we start by creating a new servlet. This servlet will be the endpoint to receive REST requests. A typical servlet declaration is as follows:

```
1 @WebServlet(name = "MyServlet", urlPatterns = {"/MyServlet"})
2 public class MyServlet extends HttpServlet {
```

It is possible to make this servlet handle all requests for this url, by changing the `urlPatterns` parameters. For instance, as follows:

```
1 @WebServlet(name = "MyServlet", urlPatterns = {"/MyServlet/*"})
2 public class MyServlet extends HttpServlet {
```

This way, any request (e.g. `http://localhost:8080/GMS/MyServlet/test` will be handled by the servlet. In the servlet, we can parse the url in order to extract information. Further details can be provided in the HTTP request, such as parameters, or structured data (for instance using JSON).

Convention

We will adopt the following convention `http://url/Application/Endpoint/Target`, where:

url The url of the application server;

Application The Java web application name;

Endpoint The endpoint (i.e., servlet);

Target The target action;

A specific example is `http://localhost:8080/GMS/MyServlet/games`. This would be the endpoint for game related actions. In the servlet, it is then possible to retrieve the target as well as the action to perform. Note that other conventions are also valid, as there is no standard for the implementation of REST webservices.

A set of HTTP methods[3] exist to specify the objective of the request. Two are of special interest, namely `GET` for url encoded parameters and `POST` for methods providing data in the request body.

Next follows the code to extract both the target and the method.

```
1 protected void processRequest(HttpServletRequest request, HttpServletResponse response)
2           throws ServletException, IOException {
3   //...
4   String[] url = request.getRequestURI().toString().split("/");
5   String target = url[url.length-1];
6   String method = request.getMethod();
7   //...
8 }
```

In lines 4 and 5 the url is retrieved, and the target extracted (i.e. `games`). In line 6 the method is extracted (i.e. `GET` or `POST`). By default web browsers will make `GET` requests. Third party applications can be used to test other requests[4]. URL encoded parameters can also be extracted from the request: `request.getParameter("name")`.

Tasks

1. Create a servlet to provide the rest webservice, related with games.
2. Extract and print both the target and method in the page, when open.

4.2 Request data

As stated, `GET` requests provide URL encoded data, as for instance `http://localhost:8080/GMS/MyServlet/games?name=Sonic&platform=megadrive`. Passing data this way, while simpler, has several drawbacks, as is the case of exposing the data in the url and the limit of 2000 characters. Furthermore, the data is not structured.

On the other hand, `POST` requests can have arbitrary length bodies, where structured information can be transmitted. A popular format is JSON[5], a lightweight object notation for data representation. Briefly, JSON data is represented as follows:

```
1 {
2   "key1" : "value1",
3   "key2" : "value2",
4   ...
5 }
```

The data corresponds to a set of `key/value` pairs, delimited by `"`, and inside `{ }`. The data to search a game, could be for instance:

[3] List of HTTP methods `https://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html`
[4] `https://insomnia.rest/`
[5] `https://www.w3schools.com/js/js_JSON_intro.asp`

```
{
  "name" : "Sonic",
  "platform" : "megadrive"
}
```

In the servlet side, the post body can be retrieved from a `BufferedReader`, in the request.

```
String data = "";
if(method.equals("POST")) {
  StringBuilder sb = new StringBuilder();
  String line;
  while ((line = request.getReader().readLine()) != null) {
    sb.append(line);
  }
  data = sb.toString();
}
```

In the example above, the if the request is a `POST` request, the request body is extracted into `data`.

Finally, being JSON a standard format, several libraries support their usage. An example is Jackson[6]. This library supports the automatic transformation of JSON strings into a Java `Map`, or into a corresponding Java object.

An example of usage is as follows:

```
//Decode JSON string
ObjectMapper mapper = new ObjectMapper();
Map<String,Object> JSON = mapper.readValue(data, Map.class);

//Encode an object as a JSON string
List<String> list = new ArrayList<>();
list.add("Value 1");
String sJSON = mapper.writeValueAsString(list);
```

Tasks

1. Add to the servlet the capability to extract the body of `POST` requests.
2. Parse the provided JSON into a Map, and verify that the data is being correctly read[7].
3. Implement, for the same target, the possibility to:
   - `GET` request: return the list of games (as JSON);
   - `POST` request, with `name` as JSON: return the details of a game with a matching name (as JSON).

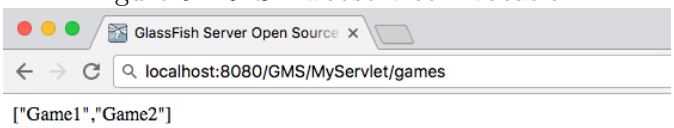
[6] `http://repo1.maven.org/maven2/com/fasterxml/jackson/core/`

[7] Jackson requires the `core`, `databind` and `annotations` jars.

4.3 Testing the service

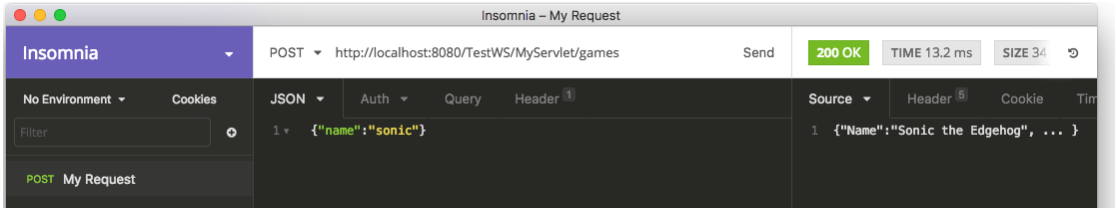
Testing REST webservices is fairly easier, since requests and answers tend to be made in human-readable formats. In this case, simply opening the browser in the correct URL (`http://localhost:8080/GMS/MyServlet/games`) should present the results in JSON (see Figure 5).

Figure 5: REST webservice invocation.



`POST` requests can be performed resorting to third party applications.

Figure 6: Invocation of a POST REST webservice in Insomnia.



4.4 Invoking the webservice

Invoking REST webservices consists in performing an HTTP request with the appropriate parameters. Although such is possible resorting only to Java, using libraries greatly eases the process. An example of a library to perform http requests is OkHttp[8]. Making a POST request resorting to that library is as follows:

```
1 OkHttpClient client = new OkHttpClient();
2 MediaType mediaType = MediaType.parse("application/JSON");
3 RequestBody body = RequestBody.create(mediaType, "{\"name\":\"sonic\"}");
4 Request request = new Request.Builder()
5   .url("http://localhost:8080/GMS/MyServlet/games")
6   .post(body) //post request
```

[8] `http://square.github.io/okhttp/`

```
  .addHeader("content-type", "application/JSON")
  .build();

Response response = client.newCall(request).execute();
//example of how to print result
Scanner sc = new Scanner(response.body().byteStream());
while(sc.hasNextLine()) {
  System.out.println(sc.nextLine());
}
```

and the corresponding GET request:

```
Request request = new Request.Builder()
  .url("http://localhost:8080/GMS/MyServlet/games")
  .get()
  .build();
Response response = client.newCall(request).execute();
```

Tasks

1. Invoke the new implemented webservice in the test application. Invoke both POST and GET requests.
2. Develop a minimal terminal based client to search and list games.

5 Remote Enterprise Java Beans

In the previous tutorial `Local` EJBs were presented. It is worth, however to present the code to perform a remote EJB invocation. First, in the bean declaration, the interface should be marked as remote:

```
@Remote
public interface RemoteGamesBeanRemote {
    List<String> listPlatformsNames();
     //...
}
```

Next, the interface should exist both in server side and client side. Having the interface it is then possible to perform the invocation of the bean. So, in the client, the first step consists in creating a domain to perform the lookup. At this stage is defined the server IP and port.

```
Hashtable<String, String> config = new Hashtable<String, String>();
config.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.sun.enterprise.naming.SerialInitContextFactory");
//server IP
config.put("org.omg.CORBA.ORBInitialHost", "192.168. ...");
//server port. 3700 is the default
```

```
config.put("org.omg.CORBA.ORBInitialPort", "3700");
Context context = new InitialContext(config);
```

After that, the lookup should be performed. The lookup string follows the format: `java:global/<Application>/<Bean>!<package>.<Interface>`. In this case, the lookup is performed as follows:

```
RemoteGamesBeanRemote remoteBean = (RemoteGamesBeanRemote)
    context.lookup("java:global/GMSWeb/RemoteGamesBean!bean.RemoteGamesBeanRemote");
```

Finally, the bean methods can be invoked as usual Java methods, for instance:

```
List<String> list = remoteBean.listPlatformsNames();
```

6 Next steps

Tasks

1. Implement a method to retrieve the user details. Note that authentication should be provided[9]. Use a REST approach.
2. Implement the previous method resorting to a WSDL approach. Authentication can be provided as parameters of the method.
3. Improve the previous webservice, by implementing token mechanism. Instead of providing login data every call, a webservice to perform login should exist.
    - The webservice receives the username and password, and returns a unique token.
    - In future calls to other webservices, the token should be provided. The token identifies the user, and avoids the need to authenticate again.

[9] See Basic authentication for a possible approach `https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication`