Web Service Portal

Simone Spaccarotella {spa.simone@gmail.com}
Carmine Dodaro {carminedodaro@gmail.com}

LaTeX

Abstract

WS Portal is a web application that allows users to choose a service, generate a client, and use it. For this purpose it was necessary to create an easy to use GUI that allows users to do these tasks all in one, just typing/pasting the URL of the WSDL file in an input text field, and then pressing the "Generate" button. This operation creates a new web service client, and shows the service and its operations into a dedicated panel, placed on the left side of the editor. In order to test the functionality of WS Portal, was also created a web service that communicates with Twitter, the well-known social network.

The entire project was developed with Java. WS Portal is a web project and it's made up of two sections: the client-side - for which it has been used GWT and Ext-GWT libraries - and the server-side - implemented using the JSP/Servlet.

Contents

1 Issues addressed **3**
- 1.1 Automatic generation of client . . . 3
- 1.2 Invocation of a generic web service . . . 4
- 1.3 Client/Server communication . . . 4
- 1.4 Countermeasures to the SQL injection . . . 5
- 1.5 Secure connection with Tomcat . . . 5

2 Use Cases **6**
- 2.1 Use Case 1 . . . 6
- 2.2 Use Case 2 . . . 7
- 2.3 Use Case 3 . . . 8

3 Classes description **9**
- 3.1 Server side . . . 9
  - 3.1.1 Generator.java . . . 9
  - 3.1.2 Compiler.java . . . 10
  - 3.1.3 GenericClient.java . . . 10
  - 3.1.4 WsdlInfoRetreiver.java . . . 11
  - 3.1.5 DataSourceConnection.java . . . 11
  - 3.1.6 DataSourceConnection.java . . . 12
  - 3.1.7 WsLibGenerator.java . . . 13
- 3.2 Client side . . . 13
  - 3.2.1 WSPortal.java . . . 14
  - 3.2.2 Controller.java . . . 15

Chapter 1

Issues addressed

The problems faced during the realization of this project have been a lot. Below are described the spots where there were major problems and how they were resolved.

1.1 Automatic generation of client

The generation of the client, together with the invocation of a generic service (covered in the next paragraph), was the most problematic aspect, especially during the debugging.

From the user point of view, in order to add a new service to the profile, the user can simply type the URL of the WSDL file in a specific text field and then press the "Generate" button. On the client side, a listener to which the button has been previously registered, retrieves the URL, validates it and sends it asynchronously to the server. On the server-side instead, a servlet processes the request making a series of operations that are listed below:

1. retrieves the URL of the WSDL file from the query string;
2. checks that this service has not been generated before (by querying the underlying database);
3. if the web service doesn't exist, it's generated and a new tuple is added to the *service* table;
4. if the user doesn't have this service, a tuple is added in the *user service* table, otherwise it sends a warning message.

After receiving the response from the server, a callback function handles the event. It parses the string received from the server (this string contains information about the added service and its methods), and updates the

GUI. Where the user already has this service in its list, a warning message is shown.

The client is created in two phases. In the first phase the source files are generated. For this purpose, we used `org.apache.axis.wsdl.toJava.Emitter` class, the superclass of `org.apache.axis.wsdl.WSDL2Java` class. It was made this choice because `WSDL2Java`, after generating the *java* files, stops the execution of the Virtual Machine, whereas `Emitter` doesn't do this. In the second phase however, the *java* files are compiled, and pasted into the classpath of Tomcat. The compilation is provided by the *javac* command, invoked by a sub-process managed by `java.lang.ProcessBuilder`. Once the library is generated, the service can be used on the fly, without rebooting the server.

1.2 Invocation of a generic web service

This step involves the invocation of a generic web service's operation, whatever it is. It was necessary to perform introspection of classes in order to create a generic client, independent of the particular service. For this purpose it was used Java reflection, through which you can know which and how many operations a service provides, which and how many input parameters the operation accepts, and what's the return parameter type. In this way, the user simply enters the required datas, using a form made available by the GUI (also generated at runtime) and press the "Send" button.

1.3 Client/Server communication

Client and server must be able to communicate without the hassle of updating the web page every time. To do so, the client must make asynchronous calls to the server, thanks to the XMLHttpRequest object, an API used by the various web scripting languages (in our case is JavaScript) that lets you communicate with the server also asynchronously over HTTP/HTTPS.

The issue addressed was the synchronization of the client state with the server state. Since all functions relating to GUI are delegated to the client, as well as the management of the events triggered by it, there was the problem of loss of data after every page refresh. The client needs to communicate with the server, which maintains persistent the user state, throughout the session. In this way the problem is solved, because every page refresh, simply retrieves the information by calling the server. Solved this problem, it rises another one. We have a client-side scripting language and a server-side programming language, so you can not be simply return an object (perhaps a bean) and extract the information from it. Therefore, we created a mapping between the two languages. The server sends a simple concatenated string.

It is splitted by the client that rebuilds the JavaScript object and sets it with the received information. A kind of object serialization.

1.4 Countermeasures to the SQL injection

SQL injection is a type of security attack that affects all web applications that provide interaction between the user - through form - and the underlying database, without performing a validity check on the input. In absence of these controls, a shrewd user can easily bypass the authentication service and access management service (eg. during the login), injecting malicious *ad-hoc* code.

To overcome this problem, we implemented an input validation at login, in order to avoid that the user can injects invalid and potentially harmful text. In particular, the invalid characters are escaped, especially those used by the SQL language.

1.5 Secure connection with Tomcat

For the client/server communication, was set up a SSL tunnel with the exchange of self-signed certificates (web of trust).

Chapter 2

Use Cases

At design time, it has been chosen to adopt the MVC pattern. It allowed us to create a flexible and easy to modify and integrate web application. Also, it allowed us to develop the application into parallel tasks.

For the design, very useful was the use of the Use Case Diagrams, illustrating the main functionalities that the system itself must provide. Below are the main scenarios for user and administrator. Particularly, a user can:

- load a service;
- use a service;

An administrator can perform the same operations of a normal user, and more, can delete a service in the database.

2.1 Use Case 1

Name: Loading Web Service
Primary actor: User
Stakeholders and interests:

- **User:** enters the URL of the WSDL file.
- **System:** generates the corresponding web service.

Pre conditions: the system works.
Success guarantee: the system loaded the services.
Main success scenario:

1. the user enters the service URL and sends the string to the server
2. the system gets the URL from the query string and generates the service

3. the system stores the service information in the database
4. the system warns the user that the service has been added and shows it into the GUI
5. the user can start again from step 1
6. the user finishes and the use case ends

Extensions:

- the service URL isn't valid and the system warns the user
- the system isn't able to loads the service and warns the user
- the system isn't able to saves the service and warns the user

2.2 Use Case 2

Name: Using Web Service
Primary actor: User
Stakeholders and interests:

- **User:** use the services provided by the system. He doesn't want to enter incorrect data, and get incorrect results. Do not want to know in detail how the system would interface with services made available for him.
- **System:** provides a GUI to the user.

Pre conditions: the user is logged into the system and the system contains the services to be provided.
Success guarantee: the user chooses the service, enter the necessary input data, and the system returns the result.
Main success scenario:

1. the user requests the display of services
2. the system shows the available services
3. the user chooses a service between those available
4. the system shows the public interface of available services
5. the user enters the necessary data
6. the system processes the request and returns the result
7. the user starts from step 1 until it considers that it has completed its work
8. the user decides that his work is completed and the use case ends

Extensions: if the user enters invalid datas, the system warn him

2.3 Use Case 3

Name: Removing Web Service
Primary actor: Administrator
Stakeholders and interests:

- **User:** want to remove a service.
- **System:** want to shows all the service datas to enable the administrator the choice of service to delete.

Pre conditions: the administrator is logged into the system and is displaying all the services.
Success guarantee: the service was chosen and the user can enter the input datas.
Main success scenario:

1. the system shows all available services
2. the administrator deletes the selected service and the use case ends

Non-functional requirements: the system must have at least one service.

Chapter 3

Classes description

This chapter describes the most important classes of the project. WS Portal is divided into two main parts: the client side and server side.

3.1 Server side

The server side is divided into 4 packages: the main package, db, servlet util. In the main package there are the most important classes that implement the server functionality. In the *db* package there are two classes that provide the connection to the database, through a datasource and implement the queries. In the *servlet* package there are all the classes registered as Java Servlet, that provides an interface between the client and the database and web services servers. Last, in the *util* package there are the utility classes that provide a specific functionality.

3.1.1 Generator.java

This class generates a library that will be used by the client to interact with its web services. The generation is divided into two phases: the creation of the source files *(.java)* and compilation. The generation of java files is delegated to the `org.apache.axis.wsdl.toJava.Emitter` class (the superclass of `org.apache.axis.wsdl.WSDL2Java`) of the Apache Axis libraries. Once generated the sources, they are given in input to the *Compiler* class. This class creates a new process that acts as a wrapper of the command "javac". Once completed, the class files are copied to the WEB-INF/classes folder of the web application, whereas the source files are deleted from the working directory. The most important method is *run()*:

```java
/**
 * Generates the java library that uses the WS described by the WSDL file.
 * @throws Exception
 */
public void run() throws Exception {
    if (isInvalidPackage()) {
        generatePackageName(null);
    }
    emitter.run(wsdlFile);
    compiler.setSourceDir(emitter);
    compiler.compile();
}
```

3.1.2 Compiler.java

```java
public void compile() throws IOException, FileNotFoundException, InterruptedException {
    // create the command in order to compile with "javac"
    ArrayList<String> command = new ArrayList<String>();
    command.add("javac");
    // if javac options file is not empty, add its name as a parameter
    if (fillJavacOptions()) {
        command.add("@" + JAVAC_OPTIONS_FILE);
    }
    if (fillJavacSourceFiles()) {
        command.add("@" + JAVAC_SOURCES_FILE);
    }
    // initialize a process builder, in order to compile the command
    processBuilder = new ProcessBuilder(command);
    /*
     * Any error output generated by subprocesses,
     * subsequently started by this object's start() method,
     * will be merged with the standard output, so that both can be read
     * using the Process.getInputStream() method.
     */
    processBuilder.redirectErrorStream(true);
    // set the working directory of this new process (is the directory which contains the parameter files
    processBuilder.directory(new File(workingDirectory));
    // start the sub process
    Process process = processBuilder.start();
    process.waitFor();
    process.destroy();
}
```

3.1.3 GenericClient.java

The most important method of this class is *invokeMethod*. It invoke a service operation dinamically. It receives the name of the method and its input parameter and returns a string that represent the web service response. If the result is a generic `java.lang.Object` it must implements the *toString* method.

```
public String invokeMethod(String packageName, MethodEnvelope envelope, String portName) throws IOException {
    Object result = "nothing";
    try {
        // locator class
        Class locatorClass = getClassByPackage(packageName, "Locator.class");
        // locator object
        Object locator = locatorClass.newInstance();
        Method getService = locatorClass.getMethod("get" + portName, new Class[]{});
        Object service = getService.invoke(locator, new Object[]{});
        Class serviceClass = Class.forName(service.getClass().getCanonicalName());
        ArrayList<TypeValue> parameters = envelope.getParameters();
        Class[] formalParameters = new Class[parameters.size()];
        Object[] actualParameters = new Object[parameters.size()];
        for (int i = 0; i < formalParameters.length; i++) {
            formalParameters[i] = Class.forName(parameters.get(i).getType());
            actualParameters[i] = parameters.get(i).getValue();
        }
        Method method = serviceClass.getMethod(envelope.getMethodName(), formalParameters);
        result = method.invoke(service, actualParameters);
    } catch (ClassNotFoundException ex) {
        result = "Class not found";
    } catch (IllegalArgumentException ex) {
        result = "Illegal argument";
    } catch (InvocationTargetException ex) {
        result = "Invocation Target exception";
    } catch (InstantiationException ex) {
        result = "Instantiation exception";
    } catch (IllegalAccessException ex) {
        result = "Illegal acces";
    } catch (NoSuchMethodException ex) {
        result = "No such method";
    } catch (SecurityException ex) {
        result = "Security exception";
    }
    return result.toString();
}
```

3.1.4 WsdlInfoRetreiver.java

Is a utility class that provides information about a given WSDL file, such as:

- *getServiceUrl*
- *getServiceName*
- *getServiceName*
- *getPortName*
- *getPackageName*

3.1.5 DataSourceConnection.java

Provides a connection to the database.

```
public class DataSourceConnection {

    private DataSource dataSource;

    /**
     *
     * @throws NamingException
     */
    public DataSourceConnection() throws NamingException {
        InitialContext initialContext = new InitialContext();
        dataSource = (DataSource) initialContext.lookup("java:/comp/env/jdbc/wsportal");
    }

    /**
     * Get the connection to the database.
     *
     * @return a java.sql.Connection object
     * @throws SQLException
     */
    public Connection getDatabaseConnection() throws SQLException {
        return dataSource.getConnection();
    }
}
```

3.1.6 SqlStatement.java

This class implements the SQL queries that the client submits to the database. For example:

login

```
public boolean logIn(String username, String password) {
    boolean result = false;
    try {
        username = username.toLowerCase();
        username = SqlCleaner.clean(username);
        statement = connection.prepareStatement("select * from user where username = ?");
        statement.setString(1, username);
        resultSet = statement.executeQuery();
        if (resultSet.next()) {
            String dbPassword = resultSet.getString("password");
            String dbSalt = resultSet.getString("salt");
            String cryptoPassword = Hash.sha1(username + password + dbSalt);
            if (dbPassword.equals(cryptoPassword)) {
                result = true;
            }
        }
    } catch (NoSuchAlgorithmException ex) {
    } catch (UnsupportedEncodingException ex) {
    } catch (SQLException ex) {
    } finally {
        closeResources();
    }
    return result;
}
```

signUp

```
public boolean signUp(String username, String password, String name, String surname) {
    boolean result = false;
    username = username.toLowerCase();
    // check if the username doesn't have strange character
    if (SqlCleaner.isClean(username)) {
        try {
            // generate the random string
            String salt = generateSalt();
            // rigenerate the hashed password
            password = Hash.sha1(username + password + salt);
            // prepare the statement
            statement = connection.prepareStatement("insert into user values(?,?,?,?,?)");
            // set the parameters
            statement.setString(1, username);
            statement.setString(2, password);
            statement.setString(3, name);
            statement.setString(4, surname);
            statement.setString(5, salt);
            // execute the update - if a tuple with the same key exists, this statement
            // will throws an SQLException and the result will be false
            statement.executeUpdate();
            result = true;
        } catch (NoSuchAlgorithmException ex) {
        } catch (UnsupportedEncodingException ex) {
        } catch (SQLException ex) {
        } finally {
            closeResources();
        }
    }
    return result;
}
```

3.1.7 WsLibGenerator.java

This is a Servlet and it's the most important one. This class generate the client library for a web service, just using its WSDL definition file.

```
@Override
protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException, FileNotFoundException {
    this.request = request;
    String wsdlFile = getWsdlFile();
    if (isNotGeneratedYet(wsdlFile)) {
        generateLib(wsdlFile);
    }
    boolean addedToTheUser = addServiceToTheUser();
    response.setContentType("text/xml");
    response.setCharacterEncoding("UTF-8");
    MessageEncoder encoder = new MessageEncoder(response.getWriter());
    if (!addedToTheUser) {
        encoder.encode("The user already has the " + service.getName());
    } else {
        addServiceOperations();
        encoder.encode(service);
    }
}
```

3.2 Client side

In order to develope the client side, it has been used the MVC pattern. The *view* package contains the GUI components, the *model* package contains the

classes that represent the underlying state of the GUI, and in the end, the *controller* package contains a class that manage the events of the GUI. More, there is a package called *callback* that contains the callback function that handle the asynchronous events (client/server communication, GUI updates, etc.). These are java file, that are compiled in javascript functions by the GWT module.

3.2.1 WSPortal.java

This is the main Class that contains the entry point of the application. When the client starts, the GWT module is loaded and the *onModuleLoad* method is called. This method initialize the GUI and the model.

```
@Override
public void onModuleLoad() {
    // instantiates the viewport
    wsportalViewport = new WsPortalViewport();
    // add the viewport into the web page
    RootPanel.get("wsportal").add(wsportalViewport);
}
```

The GUI is added in a HTML page that represent the home page, and is rendered after the GWT module loading. The home page looks like this:

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <%
        Object username = session.getAttribute("username");
        if (username == null) {
    %>
            <jsp:include page="/WEB-INF/pages/loginSignup.jsp" />
    <%
        } else {
    %>
            <jsp:include page="/WEB-INF/pages/home.jsp" />
    <%
        }
    %>
</html>
```

Its include fragments of HTML code so, if the user is logged in, the application will show the WS Portal editor, otherwise it will show the login/signup page:

login/signup fragment page

```
<head>
    <link type="text/css" rel="stylesheet" href="resources/css/gxt-all.css">
    <link type="text/css" rel="stylesheet" href="resources/css/wsportal.css">
    <title>WS Portal Home Page</title>
</head>
<body>
    <!-- this is the autogenerated script by Ext-GWT -->
    <script type="text/javascript"
            src="it.unical.inf.wsportal.WSPortal/it.unical.inf.wsportal.WSPortal.nocache.js">
    </script>
    <!-- this is the point where the wsportal is added in -->
    <div id="wsportal"></div>
</body>
```

GWT module

```
<module>
    <!-- imports the GWT API-->
    <inherits name="com.google.gwt.user.User" />
    <!-- imports the Ext-GWT API -->
    <inherits name="com.extjs.gxt.ui.GXT" />
    <!-- in order to use the RequestBuilder class -->
    <inherits name="com.google.gwt.http.HTTP" />
    <!-- the entry point class -->
    <entry-point class="it.unical.inf.wsportal.client.WSPortal"/>
    <!-- the java class that will be compiled in javascript -->
    <source path="client" />
    <source path="shared" />
</module>
```

3.2.2 Controller.java

This class manage the GUI events. It handle the event and delegate it to the related callback function.

```
@Override
public void handleEvent(ComponentEvent event) {
    String id = event.getComponent().getId();
    int eventCode = event.getType().getEventCode();

    if (id.startsWith(ComponentID.INVOKE_BUTTON)) {
        invokeMethod(event.getComponent());
    } else if (id.equals(ComponentID.ADD_WSDL_BUTTON)) {
        showLoadWsdlDialog();
    } else if (eventCode == Events.OnDoubleClick.getEventCode()) {
        showInputPortlet();
    }
}
```

invokeMethod

```
private void invokeMethod(Component component) {
    InputPortlet portlet = (InputPortlet) WSPortal.getInputPortletContainer().getPortlet(component);
    ListStore<BaseModelData> store = portlet.getInputGrid().getStore();
    MethodEnvelope envelope = new MethodEnvelope();
    envelope.setServiceUrl(portlet.getServiceUrl());
    envelope.setMethodName(portlet.getHeading());
    for (int i = 0; i < store.getCount(); i++) {
        String type = store.getAt(i).get("type").toString();
        Object value = store.getAt(i).get("param");
        envelope.addParameter(type, value);
    }
    XMLSerializer serializer = new XMLSerializer(envelope);
    String params = "xml=" + URL.encode(serializer.serialize());
    AsynchRequest request = new AsynchRequest(AsynchRequest.POST, "../invokeMethod", params);
    request.setCallback(new InvokeMethod(portlet.getHeading()));
    try {
        request.send();
    } catch (RequestException ex) {
        Toolkit.showException(ex);
    }
}
```

showLoadWsdl

```
private void showLoadWsdlDialog() {
    MessageBox messageBox = MessageBox.prompt("Load new Service", "WSDL url");
    messageBox.addCallback(new ShowWsdlDialog());
}
```

showInputPortlet

```
private void showInputPortlet() {
    WsTree tree = WSPortal.getWsTree();
    WsTreeItem item = tree.getSelectedItem();
    if (item.getType() == WsTreeItem.OPERATION) {
        // create a new portlet
        InputPortlet portlet = new InputPortlet(item.getName());
        // get the service url from the service item id
        String url = tree.getServiceOf(item).getId();
        // set the service url to the portlet
        portlet.setServiceUrl(url);
        // get the item id
        String id = item.getId();
        // if the id is equals to void, it means that the operation doesn't
        // accept any parameter, otherwise
        if (!id.equals("void")) {
            // split the string by ":"
            String[] split = id.split(":");
            // and for each element
            for (int i = 0; i < split.length; i++) {
                // get the input parameter and set it to the portlet input text field
                portlet.addParameter(split[i], "param" + i);
            }
        }
        portlet.setButtonAlign(HorizontalAlignment.CENTER);
        portlet.addButton(new InvokeButton());
        WSPortal.getInputPortletContainer().add(portlet);
    }
}
```