# CLOSING THE GAP BETWEEN WEB APPLICATIONS AND WEB SERVICES

Marc Jansen
*Ruhr West University of Applied Sciences, Computer Science Institute, Tannenstr. 43, 46240 Bottrop, Germany*

Keywords: Web Service, Web Application.

Abstract: The WWW is the killerapp of the internet. In recent years an enormously increasing number of Web Applications, as a means of human-to-computer interaction, showed up, that allows a visitor of a certain website to interact with the website. Additionally the approach of Web Services was introduced in order to allow computer-to-computer interaction on the basis of standardized protocols. This paper shows how the gap between Web Applications and Web Services can be closed by making Web Applications available to computer-to-computer interaction by a systematic approach.

## 1 INTRODUCTION

From the very early days of the internet, the Word Wide Web (WWW) was one of the most frequently used developments. According to (Hammerschall, 2005) Web Applications can be categorized at different levels. At the lowest level, simple static websites provide information to a visitor. The next step are interactive Web Applications that allow to receive data from a visitor and use this data in order to perform certain calculations with the result presented to the visitor. The last step of web technologies are Web Services. The major difference between a usual Web Application, that is a mean for human-to-computer interaction, is that Web Services are built to foster computer-to-computer interaction.

From an architectural point of view, formulars of a Web Application can be seen as interfaces to services in a service-oriented architecture (SOA). The major goal of this work, is to close the gap between Web Applications and Web Services by making Web Applications available for computer-to-computer interaction.

The developed prototype that is presented here achieves this goal by implementing a Web Service façade to Web Applications. This approach is explained in more detail in the following sections.

## 2 THE ARCHITECTURAL APPROACH

From an architectural point of view, the gap between a Web Application and a Web Service can be closed by providing a façade to the Web Application, according to the façade design pattern (Gamma, 1994). This façade allows to call the formular of a Web Application via a Web Service. The major goal of the façade design pattern is to provide an interface to a subsystem that allows to use the subsystem in an easier way. In this example the façade eases the use of Web Applications in different ways:

- The usage of a Web Service, as an interface to a Web Application allows to use standardized methods (like WSDL, SOAP and UDDI) in order to consume the service provided by the Web Application.
- The façade can decrease the number of parameters that need to be passed to the Web Application. This is described in more detail in section 4.
- Last but not least, the façade does not need to provide an external interface to all services of a Web Application but can be limited to certain services of interest.

To be able to provide a certain façade to a Web Application three major steps need to be performed. First of all the website, that provides access to the Web Application, needs to be parsed and the services (in their representation as forms) need to be extracted. In a second step the parameters of each of the forms need to be extracted and it needs to be determined which parameters are necessarily provided within the Web Service interface. Last but not least, the Web Service itself needs to be implemented, including all the descriptions that are necessary to deploy the Web Service to a usual application server. This step also includes the packaging of the Web Service itself and the necessary descriptions in a deployable package, e.g. a WAR archive.

Here, the first two steps (extracting the forms and extracting the necessary parameters for each form) can be done with the help of the pipes and filters design pattern (Buschmann and Rohnert, 1996). The approach presented here uses the results of these two filters to provide an XML description of the services provided by the Web Application that later-on allows to easily implement the Web Service (including its necessary descriptions) via an eXtended Stylesheet Language Transformation (XSL/T). An overview of the suggested architecture is provided in Figure 1:

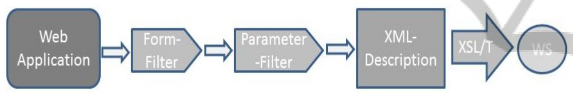

Figure 1: Proposed architecture for the creation of a Web Service facade to a Web Application.

The following provides a closer look to the solutions of the presented approach.

## 3 PARAMETER HANDLING

As described before, the parameters for the methods provided by the Web Service façade are basically the parameters provided by the form that represents the service of the Web Application. Basically, two types of parameters need to be distinguished to answer the question which parameters the Web Service façade needs. Parameters that need to be passed to a Web Application might be of two different types. On the one hand there are parameters that usually need to be provided directly by the user of a certain Web Application, e.g. the security identification number of a certain stock in a Web Application that provides stock values. On the other hand there are certain parameters that are provided by the Web Application itself, e.g. parameters used for session identification. The type of the parameter can easily be determined by an attribute of an input parameter of an HTML-Form: each input parameter might have an attribute called *type*. In case of the parameters that are provided by the Web Application itself, this attribute has the value *hidden*. All other parameters of an HTML-Form are usually parameters that need to be provided by the user of the Web Application in question. The Web Service façade to a certain Web Application obviously only needs to have these kind of parameters in its signature that usually need to be provided by the user of the Web Application and usually not the parameters that are provided by the Web Application itself.

In the developed prototype, for convenience purposes, all parameters that the Web Service provides are of type *String*. Of course it would not be much more difficult to provide different types of parameters, but this doesn't seem to be necessary for the development of a prototype.

## 4 RETURN VALUES

Beside the parameter handling of the provided Web Services, the handling of the return value is another topic of interest. The following two sections provide an overview about how the developed prototype deals with the return value.

### 4.1 Different Types of Return Values

Basically a Web Application might have different types of return values. At the first abstraction three different types of return values might be of interest:

- single – The Web Application might return a single value that is of interest. This is e.g. the case for a Web Application that provides stock values.
- list – Alternatively, the Web Application might provide more than one result. This type of return value is represented as a list. A popular example for this kind of return value might be the answer of a request to a search engine, that provides a list of URLs with information on a certain topic.
- void – Last but not least, a Web Application might not return data that is of interest to the user. This might e.g. be the case if the Web Application just starts a particular package of work. This kind of return value is quite unusual for Web Applications but might still be of interest to some Web Application, therefore it is also concerned in the developed prototype.

Again, just for the sake of lower complexity, the return value within the Web Service façade is of type *String* (respectively *String[]*) for the first two kinds of return values, and of type *void* for the last one.

## 4.2 Determing the Return Value

One major challenge during the implementation of the prototype was the determination of the actual return value. Usually Web Applications provide their result within a webpage. Therefore it is necessary to parse the webpage and determine the result of interest. In order to be able to do so, the position where the information of interest is located, has to be described. Usually an absolute description of the position, e.g. the information is placed between the n-th and m-th character doesn't work, since Web applications usually provide the information of interest in specially designed websites where the absolute position of the information changes dramatically. Here another approach was necessary. Within the developed prototype we found that a contextual representation of the position of the information worked very well. Therefore we designed regular expressions to describe what kind of information is of interest to us on the webpage that the Web Application returns. For example, in the case of a certain search engine, the regular expression for the return type looks like this:

```
<span class="url">.*</span>
```

Here, the . represents any character. Therefore in this example the return value is an array of type String with all character sequences that are within the left border (*<span class="url">*) and the right border (*</span>*).

## 5 XML REPRESENTATION

As already shown in the architectural overview in Figure 1, the implemented approach produces an intermediate XML representation of the service of interest on its way to the final Web Service. The following two sections provide an overview first on the XML representation and the structure itself. Afterwards we provide a solution how this XML representation is used in order to implement the Web Service, including its necessary descriptions, with the help of XSL/T.

## 5.1 An XML Datastructure for the Definition of Web Applications

The here described XML representation of the methods finally implemented in the resulting Web Service is a minimal set of information necessary in order to be able to implement the Web Service later on. The Document Type Definition (DTD) that defines the grammar of the XML representation looks like this:

```
<!ELEMENT forms (form*)>
<!ATTLIST forms name CDATA #REQUIRED>

<!ELEMENT form (parameter*)>
<!ATTLIST form action      CDATA          #REQUIRED
          leftBorder  CDATA               #REQUIRED
          rightBorder  CDATA              #REQUIRED
          method      (get|post)          #REQUIRED
          result   (single|list|void) #REQUIRED
          methodName  CDATA              #REQUIRED>

<!ELEMENT parameter EMPTY>
<!ATTLIST parameter name   CDATA         #REQUIRED
                    value  CDATA         #IMPLIED
                    hidden (true|false) #REQUIRED>
```

The root element of a certain XML representation is a *forms* element. This one is basically necessary in order to encapsulate the different services that a Web Application provides. As an attribute, the *forms* element provides the possibility of a *name* that might be interpreted as an acronym for the provided service.

On the next level the XML representation consists of a number of *form* elements. Each *form* element describes a single service of the Web Application on a syntactical level. Therefore it provides attributes that represent the URL under which the service is available (*action*), the left and the right border for the determination of the return values (*leftBorder*, *rightBorder*), the HTTP method used to invoke the service (*method*), the result type as explained in section 5.1 (*result*) and the name that the method of the Web Service façade (*methodName*) should have.

At the lowest level, each of the *forms* elements consists of potentially several *parameter* elements. Each parameter element consists of potentially three attributes that describe the parameter: first of all, each parameter can be identified (*name*), potentially parameters might have a value (*value*) that can be interpreted as the default value of this parameter and, last but not least, as already described in section 4 some parameters might not be visible to the end-user of the service (*hidden*).

This XML representation of a certain service is enough to implement the corresponding Web Service façade automatically, as describe in the next section.

### 5.2 Transforming Web Applications to Web Services Via XSL/T

The XML datastructure described in the last section is used to automatically implement the Web Service façade including the related descriptions. Therefore we developed a eXtended Stylesheet Language Transformation (XSL/T) script that on the one hand implements a Java class with the necessary annotations to expose the methods as Web Services according to Java Specifiction Request 181 (JSR 181) (Mullendore, 2009). The following represents an extract of this XSL script:

```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/forms">

import …;

@WebService

public class <xsl:value-of select="@name" />Service {

<xsl:for-each select="form">

@WebMethod

public String <xsl:value-of select="@methodName"/>
/>(<xsl:copy-of select="$parameters" />) {

…

}

}

</xsl:for-each>

</xsl:template>

</xsl:stylesheet>
```

On the other hand to deploy the Web Service later-on to a usual application server like JBoss or Glassfish the deployment description for the Web Service needs to be implemented in form of a *web.xml* file. This *web.xml* file is also automatically generated from the described XML representation with a similar XSL script as mentioned above for the creation of the Web Service source code.

Last but not least, the developed prototype compiles the created Web Service classes and packages these classes, together with the created *web.xml* file, into a deployable web archive (.war) file. This .war file is now ready for deployment in a standardized application server like JBoss or Glassfish.

## 6 CONCLUSIONS AND OUTLOOK

As shown in this paper, the gap between Web Applications and Web Services can be closed at least semi automatically. Especially the determination of the position of the Web Applications return value, as describe in section 5.2, needs to be defined manually. In the next research step, a completely automatically determination also of the return value of the Web Application can be concerned.

Another interesting idea would be to deploy the presented prototype in a Cloud Computing environment. This would allow to flexibly deploy and scale the resulting Web Service façade according to the demands of a certain customer.

## REFERENCES

Hammerschall. Verteilte Systeme und Anwendungen, 2005, *Pearson Studium*

Gamma, Helm, Johnson. Elements of Reusable Object-Oriented Software. 1994, *Addison-Wesley*

Buschmann, Meunier, Rohnert, Sommerland. A System of Patterns: Pattern-Oriented Software Architecture, 1996, *John Wiley & Sons*

Mullendore. JSR 181: Web Service Metadata for the Java Platform, 2009, *Java Community Process*