Meta Search Engine Web services with .NET & Java

ALEXANDRE DIBROV – FEBRUARY 21ST, 2003

PROFESOR – KARL ABERER
SUPERVISOR – PHILIPPE CUDRE-MAUROUX
# Table of Contents

## 1. Introduction
- 1.1. Structure 1
- 1.2. Prerequisites 2
- 1.3. Conventions 2
- 1.4. Support 2

## 2. Web Services & Their Frameworks
- 2.1. Web services 3
- 2.2. XML 4
- 2.3. HTTP 4
- 2.4. SOAP 4
- 2.5. WSDL 6
- 2.6. UDDI 7
- 2.7. .NET 8
- 2.8. Java 9

## 3. Meta Search Engine Web Service (MSEWS)
- 3.1. Overview 11
- 3.2. Global architecture 13
- 3.3. SEWS architecture 14
- 3.4. MSEWS architecture 20
- 3.5. Web client architecture 24
- 3.6. Implementation with .NET 25
- 3.7. Implementation with Java 27
- 3.8. UDDI registry administration 28

## 4. Performance and Interoperability Tests
- 4.1. Performance 31
- 4.2. Interoperability 34

## 5. Some Considerations Relative to Work Experience
- 5.1. Tips & tricks 36

## 6. Conclusion 37

## 7. Appendixes
- A – The wscompile tool 39
- B – The wsdeploy tool 42

## 8. References 45

## 9. Acknowledgment 47
Introduction

Are Web services just hype?

Many people are talking about Web services as the “FUTURE” of the Internet. This past year, a lot of articles and books were written on the subject; some of them describe Web services as a revolution in business-to-business (B2B) and e-business solutions.

One of the main goals of this thesis is to show that ‘Web services’ is not just another buzzword spinning around the Internet. The objective is to demonstrate, that beyond theories and concepts, it is possible to build Web services and real applications that integrate them. My intention is to exploit most of the features offered by Web services through the implementation of a Meta Search Engine Web Service on two different platforms: Java and .NET.

If someone is still skeptic as far as the real added value of the Web services, he may change his opinion, after reading this document, and understand the reasons why companies like Microsoft re-built their entire strategy around Web services.

1.1 Structure

This paper is organized into nine chapters. The first six chapters are not independent one from the others and should be read in that order. However, Chapter 2 may be skipped if you are already familiar with Web services and their frameworks. If you are more interested in my work around (Meta) Search Engine Web Services, you should focus on Chapter 3. If you are a developer who wants to build Web services, you should be more concerned about the following chapters: 4, 5, 6 and 7. Chapter 4 deals with performance and interoperability between Web services running on two different platforms. Chapter 5 contains a very useful section called ‘Tips & tricks’ that should be read before starting to write Web services. You can read my general conclusion about the project and Web services in Chapter 6. Last but not least, if you want to develop Web services using the Java technology, you will find a helpful documentation for the tools required to build and deploy your services.
1.2 Prerequisites

The reading of this document does not assume any previous background in Web services. However, understanding some listings may require some basic knowledge of XML Schema and object-oriented programming. These listings are not essential for the global comprehension; they are nevertheless useful for the details of the implementation.

1.3 Conventions

To help differentiate between the different kinds of information of this thesis, I used diverse styles of text and layout. First of all, you will find many listings containing either XML documents or fragments of code; these are displayed within a grey box that will look like that:

```xml
<?xml version="1.0" encoding="utf-8"?>
```

Sometimes in these listings, you will see portions of text in bold when I want to highlight an element:

```xml
<?xml version="1.0" encoding="utf-8"?>
```

You will also find several figures because it’s sometimes simpler to visualize things than to have them explained. Both listings and figures have a description at the bottom of the containing frame. This explanation is occasionally very useful to understand their representation if it’s not clear enough.

From time to time the text contains references represented with a number between brackets (example [2]). All references are listed in Chapter 8 for an easy consultation.

1.4 Support

Please feel free to contact me if you want to get the source code of the different objects that were created around this project. You can also get in touch with me if you have any questions about this thesis or if you need further information on the subject. I can be reached via email at one of the following addresses:

- alex@utopix.ch
- alexandre.dibrov@epfl.ch
Web services & their frameworks

What are exactly Web services and how does it work?

This chapter will start by giving the definition of the Web services. We will see that they are based on a set of standard protocols, technologies and formats that will also be described so we have a better understanding of how everything is interacting. Finally, I will introduce the two platforms that were used to develop and deploy the Meta Search Engine Web Service (MSEWS) described in details in Chapter 3.

If you are already familiar with Web services, you can skip this chapter; otherwise you can use it as a glossary for the next chapters.

2.1 Web services

They can be defined as services offered via the Web. The meaning is pretty much implicit because everybody knows what a service is, and has a more or less clear definition in mind of the Web. Maybe, that’s why a lot of people talk about them without really understanding what is behind. A common example of a Web service is that of a stock quote service; there is a request for the current price of a specified stock to which the Web service responds with the price. Basically, the service receives a request, processes it, and returns a response. You could say that a simple Java servlet behaves exactly the same, but the difference is behind the scene and that’s what makes the big asset Web services have over all other servlet-like components.

First advantage, the format of the messages exchanged between a client and a Web service is specified by a standard called SOAP. This means that Web services are not platform dependant. You could have, for example, a Web service written in c# and deployed on the .NET Framework, communicating with a Java client running on a Linux; all this can happen because they both speak the same language: SOAP.

Then, every Web service is deployed with a WSDL file that acts like an instruction manual; in case someone wants to use a particular Web Service, he simply has to look at that WSDL file to learn how to communicate with the corresponding service and use it.
Finally, you can find a specific Web service using UDDI; that’s the yellow pages of Web services where you can search by different criteria to find the appropriate service and its access points (URL).

All this is simply an exchange of XML documents over HTTP. Now that we have the big picture, let’s take a closer look at those different components.

2.2 XML

Stands for Extensible Markup Language. XML is an industry-standard protocol, administered by the World Wide Web Consortium (W3C) who defines it as a “simple and very flexible text format”. Amongst all advantages, the most important one for the Web services is that XML is not platform dependent and it allows easy data processing and exchange between different applications. The major difficulty with XML is that the envelope is often heavier than the content itself; we could imagine a particular flavor of Web services exchanging information in a more condensed format but we would certainly loose the portability provided by XML Web services.

More information about XML can be found on the W3C website [1].

2.3 HTTP

Is the acronym for Hypertext Transfer Protocol. HTTP is a W3C standard defined as an application-level protocol for distributed, collaborative, hypermedia information systems that can be built independently of the data being transferred. HTTP has been in use by the World-Wide Web global information initiative since 1990. Thus, we can say that HTTP is the most popular transfer protocol and it’s supported on almost all platforms. By implementing this standard, combined with XML, Web services remove almost all frontiers between platforms.

More information about HTTP can be found on the W3C website [2].

2.4 SOAP

The Simple Object Access Protocol (SOAP) is defined by the W3C as a lightweight protocol for exchange of information in a decentralized, distributed environment. SOAP is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses.

Below is an example of SOAP messages exchanged between a client and a Web service. The first message (LISTING 2-1) is a request sent to the Meta Search Engine Web Service (MSEWS) calling the ‘GetAvailableSEWS’ function without parameters. The Web service response (LISTING 2-2) includes the encoded string resulting from the previous call. This Web service is described in detail in chapter 3.

More information about SOAP can be found on the W3C website [3].
LISTING 2-1: SOAP request to the MSEWS, invoking the GetAvailableSEWS function.

```xml
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
   xmlns:tns="http://alex/MSEWS"
   xmlns:types="http://alex/MSEWS/encodedTypes"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body
      soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
     ">
      <tns:GetAvailableSEWS />
   </soap:Body
   
</soap:Envelope
   >
```

LISTING 2-2: SOAP response from the MSEWS

```xml
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
   xmlns:tns="http://alex/MSEWS"
   xmlns:types="http://alex/MSEWS/encodedTypes"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <soap:Body
      soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/
     ">
      <types:GetAvailableSEWSResponse>
         <GetAvailableSEWSResult xsi:type="xsd:string">&lt;?xml
            version="1.0" encoding="utf-16" standalone="yes"?&gt;
            &lt;availableSEWS&gt;
               &lt;SEWS&gt;
                  &lt;name&gt;altavista.java&lt;/name&gt;
                  &lt;binding&gt;http://localhost:8080/SEWSaltavista/SEWS?WSDL&
                     &lt;/binding&gt;
                  &lt;/SEWS&gt;
                  &lt;/availableSEWS&gt;
               </GetAvailableSEWSResult>
            </types:GetAvailableSEWSResponse>
   </soap:Body
   
</soap:Envelope
   >
```

5
2.5 WSDL

Web Service Description Language (WSDL) defines an XML grammar for describing network services as a set of endpoints that accept messages containing either document-oriented (style="document") or procedure-oriented (style="rpc") information. The endpoint is defined by a network protocol and a message format, however, the extensible characteristic of WSDL allow the messages and endpoints being described regardless of what message formats or network protocols are being used to communicate. In other words, a WSDL file is an XML document that describes a set of SOAP messages and how the messages are exchanged. Some very good tools for WSDL file processing can be found on Internet [10]; going from verification of WSDL files to automatic generation of proxy classes or SOAP request/response messages out of them.

To illustrate all this, please refer to LISTING 2-3 the WSDL document of the altavista Search Engine Web Service (SEWS) described in details in the next chapter. You can distinguish five distinct parts of a WSDL file on that example: <types>, <message>, <portType>, <binding> and <service> that are linked together.

More information about WSDL can be found on the W3C website [4].

```xml
<?xml version="1.0" encoding="utf-8"?>
<definitions
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://alex/SEWS"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
targetNamespace="http://alex/SEWS"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types />
  <message name="SearchForSoapIn">
    <part name="String_1" type="s:string" />
    <part name="int_2" type="s:int" />
  </message>
  <message name="SearchForSoapOut">
    <part name="SearchForResult" type="s:string" />
  </message>
  <portType name="SEWSSoapPort">
    <operation name="SearchFor">
      <input message="tns:SearchForSoapIn" />
      <output message="tns:SearchForSoapOut" />
    </operation>
  </portType>
</definitions>
```
LISTING 2-3: WSDL document of the altavista SEWS.

2.6 UDDI

Universal Discovery Description and Integration (UDDI) is the yellow pages of Web services. A UDDI directory entry is an XML file that describes a business and the services it offers. Both can be categorized and have keys so one can find a provider or a service by different ways. Web services are also defined through a document called a Type Model (tModel) that describes their interface. A tModel is simply a WSDL file without the <service> tag; it contains information to generate the different proxy classes or SOAP messages to communicate with the Web service but it does not specify the access point of the service.

Some public UDDI Business Registry Nodes, where you can publish your own Web services or search for existing ones, are available on the Internet. If you do not want to distribute openly your services, you can install your own UDDI server. Independently of the choice, there are two ways of accessing a UDDI registry. The
first one is through a web browser interface that provides form-based access to register or search for a Web service. The second one is programmatically using standardized API's; the main advantage of that solution is that it can be used dynamically at runtime.

More information about UDDI is available on the UDDI website [6].

2.7 .NET

Also known as Microsoft® .NET, is defined as set of software technologies designed to connect your world of information, people, systems, and devices. It enables a high level of software integration through the use of Web services.

Web services are the core of .NET; it’s the glue that holds and connects everything together (see FIGURE 2-1).

To be able to build, deploy and run Web services, Microsoft® has put in place the .NET Framework. The same environment is also used for building other Web-based and smart client applications. It includes the common language runtime, class libraries and a multiple-language support that enables developers to use the appropriate language for a specific component with the capacity to combine them within a single application.

The Microsoft® .NET Framework Software Development Kit (SDK) including the .NET Framework, as well as everything you need to write, build, test, and deploy Microsoft® .NET–connected applications and technologies, can be downloaded at http://msdn.microsoft.com/netframework/downloads/.
Microsoft® Internet Information Services (IIS) is required to be able to run your Web services. Besides, Visual Studio .NET can optionally be installed for rapidly building and integrating Web services.


2.8 Java

Sun couldn’t let all the Java developers without a solution to build, deploy and run Web service so they released the Java Web Service Developer Pack (WSDP). Based on the Java 2 SDK, this toolset adds new API including XML Messaging (JAXM), XML Processing (JAXP), XML Registries (JAXR), XML-based RPC (JAX-RPC) and the SOAP with Attachments (SAAJ). It comes also with the Apache Tomcat container and the famous Ant Build Tool; therefore, all you need to build your first Web service is your favorite text editor.

The main advantage Java WSDP has over the .NET Framework is that it is supported not only on the Windows® platforms but also on Solaris™ 2.9 and RedHat Linux® 7.2.

More information about the Java WSDP can be found on the Java website [19].

Other tools exist on the market for the development of Web services using Java; the latest version of JBuilder includes the Borland Web Services Kit supporting the JAX-RPC standards and the Apache Axis project. JBuilder is, like Visual Studio .NET, a development tool but with the benefit of being cross-platform (available on Windows®, Solaris™, Linux® and Max® OS X). JBuilder can be downloaded

More information about JBuilder can be found on the Borland website [7].

Another example is GLUE, a Web Services platform that allows you to publish any Java object as a Web service without modification. With GLUE, any Web service can also be accessed as if it were a local Java object (no stub generators or command line tools are necessary).

More information about GLUE can be found on The Mind Electric website [8].
A good example of what can be done with Web services.

In the previous chapter, we introduced the various standards, protocols and platforms that are used by Web services. Now it’s time to see all this theory in action by implementing a Meta Search Engine Web Service (MSWES).

We will start with an overview of how this MSEWS is working; next we will take a look at its architecture with a description of the different components. Finally, we will illustrate the implementation of this service on both platforms.

3.1 Overview

A Meta search engine is a search engine that collects results from other search engines, and then presents a summary of that information as the results of a search. The MSEWS is, as you would expect, a Web service that offers such functionality.

In practice, one can adopt different approaches to achieve the same goal; One could build a unique Web service that fulfils all tasks and does everything (see FIGURE 3-1). However, this solution is not very interesting for our research because it doesn’t really demonstrate the advantages of the Web services.

FIGURE 3-1: Flow of the result information for a MSEWS implemented with one Web service.
With a smaller granularity, we can imagine a service for each search engine and an extra one that will use all the other to retrieve the results and compile them (see FIGURE 3-2).

FIGURE 3-2: Flow of the result information for a MSEWS implemented with multiple Web services.

A solution like this has many advantages; first of all, the different services can be used independently. You could integrate the Yahoo Search Engine Web Service (SEWS) for example, or any of the other services, to another project; their lives are independent from each other.

Then, with such an implementation, we will have to deal with the communication between Web services. Combined with the fact that everything will be developed on two platforms, we will be able to test one of the most important features of Web services: interoperability.

Dividing the Meta search engine in multiple Web services introduced a hierarchy. We have now two types of service, MSEWS (Meta Search Engine Web Services) and SEWS (Search Engine Web Services). Without SEWS, the existence of MSEWS has no sense because the MSEWS need SEWS to collect results to be able to process them and return something. The more SEWS are connected to the MSEWS, the most interesting the result is; and because the SEWS can implement the same interface, we can create a tModel and register every SEWS bind to it in a UDDI registry. That way, the MSEWS will be able to query the registry for all Web services that implement that tModel and retrieve the SEWS access points. Other SEWS can be developed and added dynamically, without changing the MSEWS, by registering them to that UDDI registry.

This design will verify the major advantages Web services have over other distributed systems.
3.2 Global architecture

To illustrate what has been said in the overview section, the following schema (FIGURE 3-3) shows the different components acting in this Meta Search Engine Service and how they are connected together.

FIGURE 3-3: Global architecture of the Meta search engine. The client [A] is using the MSEWS [B] who looks first for the registered SEWS [D] in the UDDI registry [C]. The available SEWS [D] are then used to retrieve the different results that will be processed, compiled and sent back to the client [A].

The client is often identified with the Microsoft Internet Explorer logo in the various figures and symbolizes a browser. For this project, there has been implemented only an access point for browsers; we could, of course, connect other clients to the different Web services (MSEWS or SEWS), but this would be out of scope. We could imagine and build an application running on Palm OS integrating those services, or an access point for WAP clients; all this without altering the Web services.

Another important characteristic of this architecture (already approached in the overview) is that the MSEWS can use any Web service published in the UDDI registry as an implementation of the tModel representing the SEWS interface. No matter who developed the SEWS and on what platform it is running.

In the next sections we will start by presenting the SEWS architecture because the MSEWS are using them. We will also see how to build a Web client that connects to the MSEWS. It’s only in a later section called ‘UDDI registry’ that we will see how to publish and find a Web service.
3.3 SEWS architecture

Most search engines available on the Web provide only a browser-based interface; however, because Web services start to be successful, some of those search engines offer also an access to their information through Web services. The pioneer in this field is the famous Google (http://www.google.com) who created a project named Web APIs service [9]. When those lines were written, there was only a beta version of this service; the limitation is that it requires a Key and it is limited in the number of queries per day. The good thing is that it’s free and it works fine.

As a result of what has just been said, we may conclude by saying that there are two flavors of SEWS; one that acts like a wrapper for the HTML pages returned by the search engine. The other one is build upon the Web service offered by the search engine (see FIGURE 3-4).

![FIGURE 3-4](image)

FIGURE 3-4: The left SEWS is parsing HTML pages returned by the Yahoo website [A] to extract the result for the posted query. On the other hand, the right SEWS is using the Google APIs service [B].

Of course, it’s much simpler to build a SEWS that uses another Web service than a wrapper for an HTML document. Besides, the structure of a Web page may change over time and the wrapper risks to become useless, while new versions of a Web service will still support older functionalities because clients may still use them.

At this point, a pertinent question may arise: why doesn’t the MSEWS use directly the Google APIs service? That is because there are no standards for the format of a search result. If Altavista builds a Web service to search its database, it won’t offer the same operations Google proposes. So, we’ll have to hardcode in the MSEWS the different cases; this is the same unattractive situation we saw on FIGURE 3-1.
By inserting a SEWS between the MSEWS and the Google APIs service, we keep the implementation of the MSEWS dynamic (no static references & special cases).

We saw that there are two types of SEWS, but this difference is visible only when looking at the internal processing of the service. From the outside, MSEWS or any other consumer of these types of service would not be able to distinguish them because they implement the same interface. Let’s first have a look at this interface and then we’ll explain the internal differences.

The WSDL file of a SEWS was already presented in Chapter 2 (see LISTING 2-3); we saw furthermore that by removing the <service> tag from that document, we obtain a tModel representing the interface of the Web service. Right now, only the <message> and the <portType> tags of the tModel (see LISTING 3-1) are interesting to understand what operations are offered by the service.

```
<message name="SearchForSoapIn">
  <part name="String_1" type="s:string" />
  <part name="int_2" type="s:int" />
</message>
<message name="SearchForSoapOut">
  <part name="SearchForResult" type="s:string" />
</message>
<portType name="SEWSSoapPort">
  <operation name="SearchFor">
    <input message="tns:SearchForSoapIn" />
    <output message="tns:SearchForSoapOut" />
  </operation>
</portType>
```

LISTING 3-1: Input and output messages definition for the ‘SearchFor’ operation in the SEWS tModel.

Operations are described within the <portType> element; SEWS propose only the ‘SearchFor’ function that requires a string and an integer for the input message and returns a string in the output message. The names of the input parameters are not very explicit because they are generated automatically by the wscompile tool (see Appendix A) that creates the WSDL file from the SEWS Java interface (see LISTING 3-2).

```java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface SEWSSoap extends Remote {
  public String SearchFor(String sXMLquery, int deepness)
    throws RemoteException;
}
```
LISTING 3-2: Java interface of the SEWS used by the wscompile tool to generate the WSDL file.

For interoperability reasons, the name of all input parameters of Web services implementing that SEWS tModel must match (see the ‘Tips & tricks’ section of Chapter 5 for more details).

The Java interface (LISTING 3-2) is more explicit but still it does not tell everything; in fact, the string is the query we want to submit in XML format and the integer is the deepness of the result. The deepness represents the maximum number of result that will be returned; the query string (LISTING 3-4) must be valid according to the above XML Schema (LISTING 3-3); see the ‘Tips and tricks’ section of Chapter 5 for XML document validation.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="query">
    <xs:complexType>
      <xs:choice>
        <xs:element ref="or" />
        <xs:element ref="and" />
        <xs:element ref="not" />
        <xs:element ref="item" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="or">
    <xs:complexType>
      <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element ref="or" />
        <xs:element ref="and" />
        <xs:element ref="not" />
        <xs:element ref="item" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
  <xs:element name="and">
    <xs:complexType>
      <xs:choice minOccurs="2" maxOccurs="unbounded">
        <xs:element ref="or" />
        <xs:element ref="and" />
        <xs:element ref="not" />
        <xs:element ref="item" />
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```
Because a known problem of serialization is present when using dynamic proxies with JAX-RPC (see the 'Tips & tricks' section of Chapter 5 for more details on this problem), the SearchFor function returns a XML formatted string (see LISTING 3.6 for an example) valid according the following XML Schema (LISTING 3.5). If we were using only .NET, or if this error didn’t occur with Java, the SearchFor function could have returned a more complex object than the primitive string. Hopefully, with XML, we can encapsulate our list of results within a string; the only problem is that we loose some processing time in transforming back and forth a result in its XML representation.
LISTING 3-5: XML Schema of the string resulting from the SearchFor function
Now, that we are familiar with all that is required in order to use a SEWS, let’s take a look at what is inside the black box. We saw that the string parameter of the SearchFor function contain the query enclosed in an XML document, so the first thing to do for the SEWS is to transform that XML string into another query string compatible with the specific search engine. Then, depending if it is a SEWS acting as a wrapper or using another Web service, two different behaviors are adopted. The wrapper will parse the pages returned by the search engine for the posted query to extract the different results and store them in a list until no new results are returned or if the deepness has been reached. The SEWS using another service will have to create first an instance of the service and invoke the appropriate function to retrieve the results. In both cases, the list will be traversed to build the XML string that will be returned by the SearchFor function.

The future sections will present the implementation of the SEWS with the two technologies. For the moment, let’s take first a look at the MSEWS architecture that is thoroughly connected to the SEWS.
3.4 MSEWS architecture

As we did for the SEWS, we will start to take a look at the WSDL file (LISTING 3-7) to discover what operations are available and how to access them.

```xml
<?xml version="1.0" encoding="utf-8"?>
<definitions
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://alex/MSEWS/MSEWS"
xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/
targetNamespace="http://alex/MSEWS/MSEWS" name="MSEWS"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types />
  <message name="MSEWSIF_GetAvailableSEWS" />
  <message name="MSEWSIF_GetAvailableSEWSResponse">
    <part name="result" type="s:string" />
  </message>
  <message name="MSEWSIF_SearchFor">
    <part name="String_1" type="s:string" />
    <part name="int_2" type="s:int" />
    <part name="String_3" type="s:string" />
  </message>
  <message name="MSEWSIF_SearchForResponse">
    <part name="result" type="s:string" />
  </message>
  <portType name="MSEWSIF">
    <operation name="GetAvailableSEWS">
      <input message="tns:MSEWSIF_GetAvailableSEWS" />
      <output message="tns:MSEWSIF_GetAvailableSEWSResponse" />
    </operation>
    <operation name="SearchFor" parameterOrder="String_1 int_2 String_3">
      <input message="tns:MSEWSIF_SearchFor" />
      <output message="tns:MSEWSIF_SearchForResponse" />
    </operation>
  </portType>
  <binding name="MSEWSIFBinding" type="tns:MSEWSIF">
```
LISTING 3-7: WSDL file of the Java MSEWS.

The `<portType>` element indicates that MSEWS offer two operations; one is GetAvailableSEWS that takes no parameters and returns a string. The other is SearchFor that takes two strings and an integer as parameters and returns also a string.
The names of these functions are quite explicit, so, we’ll focus our attention on their parameters and results; GetAvailableSEWS will return a list of SEWS defined by their name and binding (access point), all this encapsulated in an XML string. LISTING 3-8 and 3-9 show respectively the XML Schema and an example of response for this operation.

```xml
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="availableSEWS">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="SEWS"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="SEWS">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="binding"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="binding" type="xs:string"/>
</xs:schema>

LISTING 3-8: XML Schema of the XML string returned by the GetAvailableSEWS function.

```xml
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<availableSEWS>
  <SEWS>
    <name>google.net</name>
    <binding>http://localhost/SEWSgoogle/SEWS.asmx?WSDL</binding>
  </SEWS>
</availableSEWS>

LISTING 3-9: Example of result returned by the GetAvailableSEWS function.
The SearchFor function must be invoked with three parameters; the first one is the query string that will be simply forwarded as is to the SEWS. The second one is the depthness, an integer that will also be forwarded without any modification to the SEWS. Finally, the third parameter is an XML string that encapsulates the list of access points (URL) of the SEWS that we want to use; LISTING 3-10 and 3-11 show respectively the XML Schema and an example of that parameter.

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="SEWStoUse">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="binding" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="binding" type="xs:string" />
</xs:schema>
```

LISTING 3-10: XML Schema of the third parameter for the SearchFor operation in MSEWS

```
<?xml version="1.0" encoding="utf-8" ?>
<SEWStoUse>
  <binding>
    http://localhost:8080/SEWSgoogle/SEWS?WSDL
  </binding>
  <binding>
    http://localhost/SEWSgoogle/SEWS.asmx?WSDL
  </binding>
</SEWStoUse>
```

LISTING 3-11: Example of a valid third parameter XML string for the SearchFor operation in MSEWS

Because the SearchFor operation requires specifying at least one SEWS access point to use, if you don't know any SEWS location, you can first use the GetAvailableSEWS function that will look in a UDDI registry for all known Web services that implement the SEWS tModel and return their name and URL. When calling the SearchFor function, all 'to use' SEWS indicated in the third parameter will be queried with the two first parameters. Next, the results will be compiled and ordered according to the rank given by the following formula (LISTING 3-12). The XML string returned by the MSEWS SearchFor function is formatted according the same XML Schema used for the SEWS SearchFor response (see LISTING 3-5).

```
Ranking = Pos1 + … + PosY + (X – Y) * (MaxNumberOfResults + 1)
```

LISTING 3-12: Formula giving the overall rank of a result that was found by Y SEWS. Pos1 to PosY represent the rank according to the different SEWS that returned it, X represent the number of SEWS requested and MaxNumberOfResults is the maximum number of results returned by the SEWS. (Example: if a result was returned by google and yahoo, respectively in 2nd and 5th position, but was not found by altavista who returned 24 results (the maximum), Y=2, X=3, Pos1=2, Pos2=5, MaxNumberOfResult=24.)
3.5 Web client architecture

To use the MSEWS and the SEWS, we need to build a client; because I wanted to be able to run this client on most computers, I decided to create a Web client that is accessible through any Web browser. We could also build a Windows Forms client, a Java application or any other client that can consume Web services.

As we know what type of client we want, let’s design it; for that, we need to decide what functionalities we want to provide to the user. I wanted to give the user full control over the Web service; this implies that he will be able to specify what SEWS he want to use from a list of available SEWS. He will also be given the possibility to define the deepness of the result (the deepness is the maximum number of results we want to retrieve). Finally, a text box must be provided to enable the user to enter his XML query validated by the XML Schema available on LISTING 3-3. Obviously, it isn’t very user-friendly to have to enter the query in an XML format, but it wasn’t the purpose of this project to study the parsing for transforming a more easy to use string entered by the user in its corresponding XML tree. A very good idea for future work would be to improve that functionality of the client.

![Screen capture of a browser window displaying the Web client of the .NET MSEWS.](image)

**FIGURE 3-5:** Screen capture of a browser window displaying the Web client of the .NET MSEWS. On the left of the page are the different fields of the form used to parameterize the search. On the right, you can see the returned list of results in an HTML table. The little chronometer on the left gives the time elapsed for retrieving the result.
3.6 Implementation with .NET

In this section we will examine how have been implemented the different Web services and the Web clients using the Microsoft .NET technology and VisualStudio.NET. Because it is be rather boring to deal with the syntax of a loop in c# (there are several good books written on the subject), I will focus on the presentation of the diverse APIs and strategies used to build the various services.

First of all, we saw in the previous sections that there is a lot of XML processing; as expected, the .NET framework offers two very intuitive libraries System.Xml and System.Xml.XPath used respectively to Read/Write and parse XML documents. An example of usage of the first library, to write an XML string is presented in LISTING 3-13.

```csharp
StringWriter sw = new StringWriter();
XmlTextWriter writer = new XmlTextWriter(sw);
writer.WriteStartDocument(true);
writer.WriteStartElement("SEWStoUse");
writer.WriteElementString("binding",sURL);
writer.WriteEndElement();
writer.WriteEndDocument();
string sToUse = sw.ToString(); //string containing an XML doc
```
LISTING 3-13: How to build an XML string in c#

The second library supports the W3C XML Path Language (XPath) Version 1.0 Recommendation [5]; an example of usage is presented on LISTING 3-14.

```csharp
ArrayList alResult = new ArrayList();
XPathDocument doc =new XPathDocument(new StringReader(sXML));
XPathNavigator nav = doc.CreateNavigator();
XPathNodeIterator nodes = nav.Select("//SEWStoUse/binding");
while (nodes.MoveNext())
{
    alResult.Add(nodes.Current.Value);
}
```
LISTING 3-14: Example of c# code demonstrating how to build an ArrayList containing all the nodes selected by the path ‘//SEWStoUse/binding’ of the XML string sXML.

For parsing HTML documents (used by the wrapper style SEWS), I found a useful API on the outstanding Microsoft website related to .NET (www.gotdotnet.com). It’s a SGMLReader that can be used to convert HTML into well-formed XML. After that the above presented XML libraries can be exploited to extract what is needed from that document. More information about the SGMLReader can be found on the www.xmlforasp.net website [13].
Consuming Web services with .NET is also rather simple; all you need to do, is to add a so called ‘Web reference’ using a very intuitive browser (see FIGURE 3-6 for a screen capture) that helps you locating online Web Services and using them in your applications. This tool creates the different stubs from a specified WSDL file; then these stubs can be instantiated as if they were local objects (see LISTING 3-15 for an example of how to invoke functions on Web services).

```csharp
alex.tmodel.SEWSSoap proxy = new alex.tmodel.SEWSSoap();
proxy.Url = "http://localhost/SEWSgoogle/SEWS.asmx?WSDL";
string tempXMLresult = proxy.SearchFor(sXMLquery, deepness);
```

LISTING 3-15: Invocation in c# of the SearchFor function offered by the google SEWS.

To query a UDDI registry, I used the UDDI .NET SDK v.1.76 Beta available on the msdn.microsoft.com website [14]; this SDK offers some user-friendly libraries to query and publish services in UDDI directories.

Everything else in the development of those Web services was only basic c# programming. As far as the client is concerned, I used the Web Forms components available within ASPX pages (the successor of ASP).
3.7 Implementation with Java

This section describes how the different SEWS and the MSEWS have been implemented with the Java technology. No Web client was build using this technology because the client developed with .NET was also exploitable for the Java Web services; moreover, building clients on diverse platforms wasn’t really the main target of this project.

Because the Java language is not very different of c#, almost the same strategies have been adopted on this platform. However, the libraries are not really identical; therefore it’s very instructive to take a look at those divergences.

XML processing in Java to perform exactly the same operations we performed with .NET, requires, at least, twice as more lines of code; without speaking of the fact that the Java libraries are a lot less intuitive to use. LISTING 3-16 shows how to build the same XML string created with .NET on LISTING 3-13; on both listing, you can see in bold the same operation!

```
org.w3c.dom.Document xmldoc= new DocumentImpl();
org.w3c.dom.Element root = xmldoc.createElement("SEWstoUse");
org.w3c.dom.Element e = xmldoc.createElementNS(null, "binding");
org.w3c.dom.Node n = xmldoc.createTextNode(sURL);
e.appendChild(n);
root.appendChild(e);
xmldoc.appendChild(root);
StringWriter sw = new StringWriter();
XMLSerializer serializer = new XMLSerializer(sw,null);
serializer.asDOMSerializer();
serializer.serialize( xmldoc.getDocumentElement() );
String sResult = sw.toString();//string containing an XML doc
```

LISTING 3-16: How to build an XML string in Java

Parsing HTML documents with Java is again not as elegant as it was with .NET; I used helper classes coming from two libraries javax.swing.text.html.parser and javax.swing.text.html. More information about the Swing HTML parser can be found on the java.sun.com website [25].

To take advantage of Web services in Java, I used essentially the javax.xml.rpc library. LISTING 3-17 shoes the Java implementation of the .NET code of LISTING 3-15.

```
```
URL sewsWsdlUrl = new URL("http://localhost/SEWSgoogle/SEWS.asmx?WSDL");
ServiceFactory serviceFactory = ServiceFactory.newInstance();
Service SEWSservice = serviceFactory.createService(
    sewsWsdlUrl, new QName("http://alex/SEWS", "SEWS"));
SEWSproxy myProxy = (SEWSproxy) SEWSservice.getPort(new QName("http://alex/SEWS", "SEWSSoapPort"),
    MSEWS.SEWSproxy.class);
String tempXMLresult = myProxy.SearchFor(sXMLquery, deepness);

LISTING 3-17: Invocation in Java of the SearchFor function offered by the google SEWS.

Querying UDDI directories in Java is done using the JAXR API. It wasn’t a surprise to notice that this part of the implementation was also less convivial in Java than in .NET. Hopefully, I found a good article covering the subject on the www.javaworld.com website [24].

**3.8 UDDI registry administration**

Because we wanted to be able to add or remove dynamically the SEWS that are used by the MSEWS, the easiest and most elegant solution was to make use of a UDDI registry. SEWS are public services, so I decided to use the public Microsoft test UDDI registry [15].

All you need to publish Web services on that business registry node is to open an account or use your regular .NET Passport connection if you have one. There are two levels of publisher accounts for UDDI; the initial level allows you only to 4 Business Services per Business Entity but you can straightforwardly upgrade to the unlimited level by sending you request to uddiask@microsoft.com.

Before publishing services, you must create a Provider; then you will be able to add your different services and bind them to their access points. You can also manage tModels separately than can be added as Instance info for an access point.

Every time you create an object in a UDDI registry, a unique key is automatically generated to identify that component. You have also the possibility to classify your elements according to a categorization scheme; a predefined set of categories, derived from an internal or external hierarchy.

All these descriptions of a Web service match with different approaches a user may use for querying the UDDI registry to find the right service. For example, we saw in a previous section that MSEWS look for all Web services that are implementing the SEWS tModel identified by its key (uuid:aaad0ab9-bcf1-4823-a716-e098402d0a57).
The below figure shows the browser-based interface of the Microsoft UDDI registry test site.

FIGURE 3-7: Screen capture of a browser window opened on the Microsoft UDDI registry test site. On the left frame, you can see all the diverse components that can be created. The right panel shows the details for the SEWS tModel.
Performance and interoperability tests

One of the main interests in studying Web services on two platforms was to determine whether they are interoperable or not, and to determine what platform is the best in terms of performance.

4.1 Performance

SEWS are not a very good example of services on which you can run performance tests. Their time of response depends of external factors that cannot be controlled; such as the network traffic or the response time of the search engine that is used. Besides, the implementations are not the same with the technologies, so we would judge more the implementation than the platform.

For those reasons, I decided to create a Web service that returns the number of prime numbers smaller than a certain integer that will be parameterized. You can find below the implementations of this Web service in Java (LISTING 4-1) and c# (LISTING 4-2).

```java
public int CountLessThan(int iNumber)
{
    ArrayList primes = getPrimesLessThan(iNumber);
    return primes.size();
}

private ArrayList getPrimesLessThan(int iNumber)
{
    ArrayList alResult = new ArrayList(iNumber);
    for (int i=0;i<iNumber;i++)
    {
        alResult.add(new Integer(i+1));
    }
    int iPointer = 1;
    while (iPointer<alResult.size())
    {
```
int iBase = ((Integer)alResult.get(iPointer)).intValue();
for (int k = alResult.size()-1; k > iPointer; k--)
{
    if (((Integer)alResult.get(k)).intValue()%iBase)==0)
        alResult.remove(k);
    iPointer++;
}
return alResult;

LISTING 4-1: Java implementation of the PrimeNumbers Web service

@WebMethod
[SoapRpcMethod(Binding="PrimeSoapPort")]
public int CountLessThan(int int_1)
{
    ArrayList primes = getPrimesLessThan(int_1);
    return primes.Count;
}

private ArrayList getPrimesLessThan(int iNumber)
{
    ArrayList alResult = new ArrayList(iNumber);
    for (int i=0; i<iNumber; i++)
    {
        alResult.Add(i+1);
    }
    while (iPointer<alResult.Count)
    {
        int iBase = (int)alResult[iPointer];
        for (int k = alResult.Count-1; k > iPointer; k--)
        {
            if (((int)alResult[k]%iBase)==0) alResult.RemoveAt(k);
        }
        iPointer++;
    }
    return alResult;
}

LISTING 4-2: C# implementation of the PrimeNumbers Web service
You can see that there is exactly the same number of operations; moreover the only object used with both implementations is an ArrayList that stores the prime numbers.

First experiment I did, was to look at the difference in the time of response of the two services depending on the number of operations. This number is proportional to the parameter of the function used: getPrimesLessThan(int iNumber).

You can see on the chart of FIGURE 4-1 that Java is slower than .NET for a significant number of operations.

![FIGURE 4-1: Chart showing the time of response of the PrimeNumbers Web service that determines all prime numbers smaller than a certain parameter](image)

I wanted also to test which platform has a faster response for N simultaneous requests. Unfortunately, this wasn’t possible because the version of IIS included with Windows XP Professional is not as versatile or powerful as the one shipped with server editions of Windows. One of the main limitations is that it allows you create only one Web site, and it allows a maximum of 10 simultaneous TCP connections.
4.2 Interoperability

This feature was easy to test since we designed our MSEWS to be able to use any Web service implementing the SEWS tModel.

Of course, because each platform has its preferences, there are some rules to follow if one wants to build interoperable services.

Most important, it is better to build services that return simple types like strings or integers. If your results are complex, you can use XML to encapsulate them the way we did it for our diverse Web Services.

In a near future, because a lot of people are concerned by interoperability, all platforms will fully support complex types as well. Web sites and forums are dedicated specially to this; as an example, sun proposes a loosely organized forum of SOAP vendors working towards a testbed for interoperability testing between SOAP implementations [26].

To conclude this section, I want to point out that Web services are the most interoperable pieces of software I had to work with. It is not just a theory, it works! And the best example that illustrates it is the Meta search engine that was developed.
Some considerations relative to Work experience

In this chapter I would like to share with you some personal thoughts related to work environment and the experience I acquired by going through this extremely interesting research and development project.

Before starting the project, my knowledge of c# and the .NET framework was very superficial compared to my Java background. As a result, I feared experiencing odd situations in the Microsoft environment. This was the reason why I decided to start the project by implement first the .NET part. Everything went very smoothly and to my surprise I discovered that VisualStudio.NET [16] is a very handy and intuitive tool for developers. I was used to code with editors such as UltraEdit which brings only a little bit of color in your code. Compared to that, VisualStudio.NET is fabulous! It warns you on the fly if the syntax of your code is incorrect. You are not supposed to know by heart all the APIs when programming with VisualStudio.NET. You just have to remember what class offers the functionality you are looking for. Last but not least is the deployment. A Web service can be build and deploy with only one mouse click in VisualStudio.NET. There is no need to take care of any CLASSPATH or to create thousand build.xml files. The list of user-friendly features of VisualStudio.NET can go on and on but if I were to resume, in VisualStudio.NET, everything you need for development and deployment is integrated in only one and the same tool. In theory, there are also such tools for the Java language and one of them is the famous Borland JBuilder. However, from what I could experience until now, they are not so well integrated to application servers.

Another significant aspect related to work environment is the documentation and support. The lack of documentation and support available for the Java Web Service Developer Pack was sometimes misleading, extremely time consuming and stress raising. I spent a lot of time on different forums related to Web services, including the one available on the java.sun.com website [23]; Not to mention that some of the developers posting on those forums were even considering migrating their work on other platforms due to the lack of support from Java. To resume, I would say that the work performance has been affected all along the project by this absence of support and documentation on the Java Web Service Developer Pack.
It goes without saying that the work environment is something very personal and a lot of people may find VisualStudio.NET a little bit too automatic because they want to know exactly what is behind that one-click built feature.

### 5.1 Tips & tricks

In this section, you will find miscellaneous tips and tricks that I discovered in my research around Web services. It might be very useful for future Web service developers.

XML documents can be validated against a W3C XML Schema using the XSD Schema Validator available online [12]. You can also check with the same application if an XML document is well-formed and if a schema is valid.

Other useful tools, specially dedicated to Web services, are available on the same site [10]. One of them is for WSDL verification; another one is called the Webservice Studio and allows you to invoke web methods interactively. The user can provide a WSDL endpoint from which the tool will generate .NET proxy and displays the list of methods available. The user can choose any method and provide the required input parameters. A SOAP request is after that generated and sent to the server; the response is parsed to display the return value.

At this time, there is no workaround to consume Web services using a dynamic proxy that returns complex types because of a known serialization problem. You can use a static client stub instead or encapsulate your object in an XML string.

The names of the function parameters are automatically transformed by the wscompile tool when generating the WSDL file. The name becomes “type_n” where type is the parameter type (“string”, “int”) and n its position. Because this can not be parameterized with Java, if you want to create a tModel used by services developed on the two technologies, start from the Java WSDL and adapt the .NET Web services that can be parameterized to match the tModel settings.

Always check that the access point (URL) of a Web service is correct before calling a Web method in .NET. When adding the reference, the port information is deleted from the URL; port 80 is used by default.

If you encounter a problem whose solution isn’t listed in the above paragraphs of this section, you should search on Internet, especially on forums related to Web services; someone else encounter certainly the same difficulties.
Conclusion

The research and development work performed around the Web services was extremely interesting and sometimes unexpectedly astounding. I was pleased to discover that Web services is a very elegant solution, probably the most elegant today, for designing, building and deploying distributed systems. This project gave me the opportunity to design, develop and deploy Web services by using two leading technologies, Java and .NET. The project provided me the possibility to put theory into practice and to experiment myself that Web services are independent of any programming language, platform or operating system. The satisfaction to deploy and integrate the Meta search engine developed using Web services made me think of many applications embedding this technology.

We saw in the previous chapters that there are strategies to adopt if we want to build Web services that are exploitable from any platform; also called interoperability. You may ask yourself what technology to choose for your project. Is the answer straightforward!

We saw in the previous chapters that performance wise there is no obvious difference between the two technologies. Platform wise, the main advantage Java WSDP has over the Microsoft .NET Framework is that it is supported not only on the Windows platforms but also on Solaris 2.9 and RedHat Linux 7.2.

Programmatically speaking, we saw that a major difference between the two technologies is the length of code, as already shown in the implementation section of Chapter 3. To do the same operation in Java, you will have to write at least twice more code than you would do with .NET. This is probably due to the fact that c# is more recent and it could lean a lot from the Java history and experience. Inspired by the Java API, the c# libraries took only the best and improved the other things.

Personally, if I need to implement a Web service and I have the possibility to choose, I would choose to go with .NET because I know that I will do it faster and without pain which would mean that the cost of development would be more adapted to market’s expectations.

This may or may not help you in making your choice of the best and most appropriate technology.
Appendixes

In this chapter, you can find a draft of the documentation for two valuable applications packaged with the Java Web Services Developer Pack and used to compile (A) and deploy (B) Web service. This documentation was written by Dale Green and posted on the ‘Java Technologies for Web Services’ forum; I only adapted and formatted it to fit this paper.

The official documentation is very modest for developing Web services using the Java technology; a lot of people are unhappy and show their frustration within different discussion groups where a lot of support can be found because developers are helping each other.

More helpful information can be found on the various forums available on the Java website [23].

A – The wscompile tool

This application generates stubs, ties, serializers, and WSDL files used in JAX-RPC clients and services. The tool reads as input a configuration file and either a WSDL file or an RMI interface that defines the service.

wscompile [options] <configuration-file>

LISTING 7-1: The wscompile tool syntax.

By convention, the configuration file is named ‘config.xml’, but this is not a requirement. The following table lists the wscompile options. Note that exactly one of the -import, -define, or -gen options must be specified.

-classpath <path> specify where to find input class files
-cp <path> same as -classpath <path>
-d <directory> specify where to place generated output files
-define read the service's RMI interface, define a service
-f:<features> enable the given features (See the below table for a list of features. When specifying multiple features, separate them with commas.)
-features:<features> same as -f:<features>
-g generate debugging info
-gen same as -gen:client
-gen:client generate client artifacts (stubs, etc.)
-gen:server generate server artifacts (ties, etc.) and the WSDL file (If you are using wsdeploy you do not specify this option.)
-gen:both generate both client and server artifacts
-httpproxy:<host>:<port> specify a HTTP proxy server (port defaults to 8080)
-import read a WSDL file, generate the service’s RMI interface and a template of the class that implements the interface
-keep keep generated files
-model <file> write the internal model to the given file
-nd <directory> specify where to place non-class generated files
-O optimize generated code
-s <directory> specify where to place generated source files
-verbose output messages about what the compiler is doing
-version print version information

LISTING 7-2: Options description for the wscompile tool.

datahandleronly always map attachments to the DataHandler type
explicitcontext turn on explicit service context mapping
infix=<name> specify an infix to use for generated serializers
nodatabinding turn off data binding for literal encoding
noencodedtypes turn off encoding type information
nomultirefs turn off support for multiple references
novalidation turn off full validation of imported WSDL documents
searchschema search schema aggressively for subtypes
serializeinterfaces turn on direct serialization of interface types

LISTING 7-3: Features (delimited by commas) that may follow the -f option.
The wscompile tool reads the configuration file (config.xml), which contains information that describes the web service.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://java.sun.com/xml/ns/jax-rpc/ri/config">
  <service> or <wsdl> or <modelfile>
</configuration>
```

Listing 7-4: Basic structure of the config.xml file.

The `<configuration>` element may contain exactly one `<service>`, `<wsdl>`, or `<modelfile>` element.

If you specify this element, wscompile reads the RMI interface that describes the service and generates a WSDL file.

In the `<interface>` subelement, the `name` attribute specifies the service's RMI interface, and the `servantName` attribute specifies the class that implements the interface.

```xml
<service name="CollectionIF_Service"
  targetNamespace="http://echoservice.org/wsdl"
  typeNamespace="http://echoservice.org/types"
  packageName="stub_tie_generator_test">
  <interface name="stub_tie_generator_test.CollectionIF"
    servantName="stub_tie_generator_test.CollectionImpl"/>
</service>
```

Listing 7-5: Example of a `<service>` element.

If you specify the `<wsdl>` element, wscompile reads the service's WSDL file and generates the service's RMI interface.

The `location` attribute specifies the URL of the WSDL file, and the `packageName` attribute specifies the package of the classes generated by wscompile.

```xml
<wSDL location="http://tempuri.org/sample.wsdl"
  packageName="org.tempuri.sample" />
```

Listing 7-6: Example of a `<wsdl>` tag

The `<modelfile>` element is for advanced users. If config.xml contains a `<service>` or `<wsdl>` element, wscompile generates a model file that contains the internal data structures that describe the service. If you've already generated a model file in this manner, then you can reuse it the next time you run wscompile.

```xml
<modelfile location="mymodel.xml.gz"/>
```

Listing 7-7: Example of a `<modelfile>` element
B – The wsdeploy tool

This application reads a WAR file and the jaxrpc-ri.xml file and then generates another WAR file that is ready for deployment.

Behind the scenes, wsdeploy runs wscompile with the -gen:server option. The wscompile command generates classes and a WSDL file which wsdeploy includes in the generated WAR file.

```
wsdeploy <options> <input-war-file>
```

LISTING 7-8: The wsdeploy tool syntax.

```
-classpath <path> specify an optional classpath
-keep keep temporary files
-o <output-war-file> specify where to place the generated war file
-tmpdir <directory> specify the temporary directory to use
-verbose output messages about what the compiler is doing
-version print version information
```

LISTING 7-9: The wsdeploy tool’s options (Note that the -o option is required).

Typically, you create the input WAR file with a development tool or with the ant war task. Here are the contents of a simple input WAR file:

```
META-INF/MANIFEST.MF
WEB-INF/classes/hello/HelloIF.class
WEB-INF/classes/hello/HelloImpl.class
WEB-INF/jaxrpc-ri.xml
WEB-INF/web.xml
```

LISTING 7-10: The contents of a simple input WAR file.

In this example, HelloIF is the service’s RMI interface and HelloImpl is the class that implements the interface. The web.xml file is the deployment descriptor of a web component. The jaxrpc-ri.xml file is described in the next section.

```
<?xml version="1.0" encoding="UTF-8"?>
```
<endpoint name="MyHello" displayName="HelloWorld Service" description="A simple web service" interface="hello.HelloIF" implementation="hello.HelloImpl"/>
<endpointMapping endpointName="MyHello" urlPattern="/hello"/>
</webServices>

LISTING 7-11: The jaxrpc-ri.xml file for a simple HelloWorld service

The <webServices> element must contain one or more <endpoint> elements. In this example, note that the interface and implementation attributes of <endpoint> specify the service's interface and implementation class. The <endpointMapping> element associates the service port with an element of the endpoint URL path that follows the urlPatternBase.

For developers who are familiar with WSDL, SOAP, and the JAX-RPC specifications, below are some advanced topics.

Namespace Mappings

```
schemaType="ns1:SampleType"
xmlns:ns1="http://echoservice.org/types"
```

LISTING 7-12: Example of a schema type name

When generating a Java type from a schema type, wscompile gets the class name from the local part of the schema type name.

To specify the package name of the generated Java classes, you define a mapping between the schema type namespace and the package name.

You define this mapping by adding a <namespaceMappingRegistry> element to the config.xml file.

```
<service>
  ...
  <namespaceMappingRegistry>
    <namespaceMapping namespace="http://echoservice.org/types" packageName="echoservice.org.types"/>
  </namespaceMappingRegistry>
  ...
</service>
```

LISTING 7-13: Example of a <namespaceMappingRegistry> element.
A handler accesses a SOAP message that represents an RPC request or response. A handler class must implement the javax.xml.rpc.handler interface. Because it accesses a SOAP message, a handler can manipulate the message with the APIs of the javax.xml.soap package.

* Encryption and decryption
* Logging and auditing
* Caching
* Application-specific SOAP header processing

LISTING 7-14: Example of handler tasks

A handler chain is a list of handlers. You may specify one handler chain for the client and one for the server. On the client, you include the <handlerChains> element in the jaxrpc-ri.xml file. On the server, you include this element in the config.xml file.

```xml
<handlerChains>
    <handler className="acme.MyHandler" headers ="ns1:foo ns1:bar"/>
    <property name="property" value="xyz"/>
  </handler>
  </chain>
</handlerChains>
```

LISTING 7-15: Example of the <handlerChains> element in config.xml

For more information on handlers, see the SOAP Message Handlers chapter of the JAX-RPC specifications.
References


[26] SOAP Builders @ Sun (about interoperability): http://java.sun.com/wsinterop/sb/index.html
Acknowledgment

First of all, I want to express my gratitude for Professor Karl Aberer (karl.aberer@epfl.ch), head of the Distributed Information Systems Laboratory (LSIR) at EPFL, who created the project especially for me. He is also the one who introduced me to distributed information systems, during his ‘Conception of Information Systems’ course, and gave me the desire to do my diploma project in this field.

Special thanks as well to my supervisor Philippe Cudré-Mauroux (philippe.cudre-mauroux@epfl.ch) who directed and helped me during all stages of the project. He is the person who motivated me to study more for finding a solution to different problems I encountered.

I want also to thank my second supervisor Jie Wu (jie.wu@epfl.ch) who presented to Professor Karl Aberer my interest in doing my diploma project around Web services.

My recognition goes furthermore to Manfred Riedo, my predecessor in the development of a distributed Meta search engine. My work is inspired by the Web services he build for the Sun ONE platform.

Thanks also to Jean-Pierre Dupertuis (jean-pierre.dupertuis@epfl.ch), system manager at the Laboratory for computer Communications and Applications (LCA). He is the person that installed all the applications and configured the machine I needed to fulfill my work.

Finally I want to thank Dale Green, the creator of the documentation for the tools used to compile and deploy my Java Web services, and other authors of messages posted on the ‘Java Technologies for Web Services’ forum where I found a lot of answers to my questions.