Web Services

A simple FlexCorp example

Author: Jon Barnett
Document version: 1.0

Table of contents

1 Web services and FlexCorp 1-1
1.1 Using existing infrastructure 1-1
1.2 Basic web service implementation 1-2
1.3 Reading web service requests 1-2
1.4 Retrieving information 1-3
1.5 Generating the response document 1-4
1.6 A simple client 1-5
1.7 Web service document exchange 1-6
1.8 Demonstration service 1-7

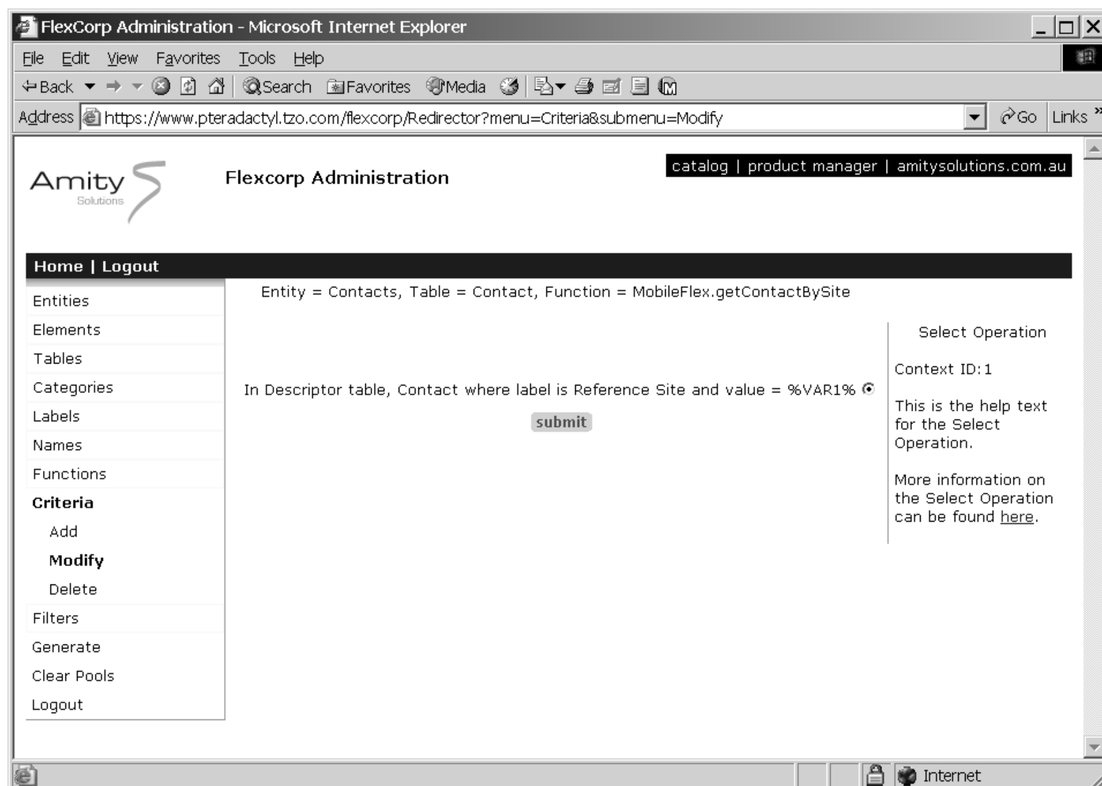
1 Web services and FlexCorp

FlexCorp as a data management framework controls the composition, storage and retrieval of information. The management of the data is particularly suited to generating XML elements. The advantage with using FlexCorp is that a business analyst can manage the data composition and the sequence of data without requiring reprogramming. We'll demonstrate the simplicity of the programming to deliver the web service and how FlexCorp controls the data.

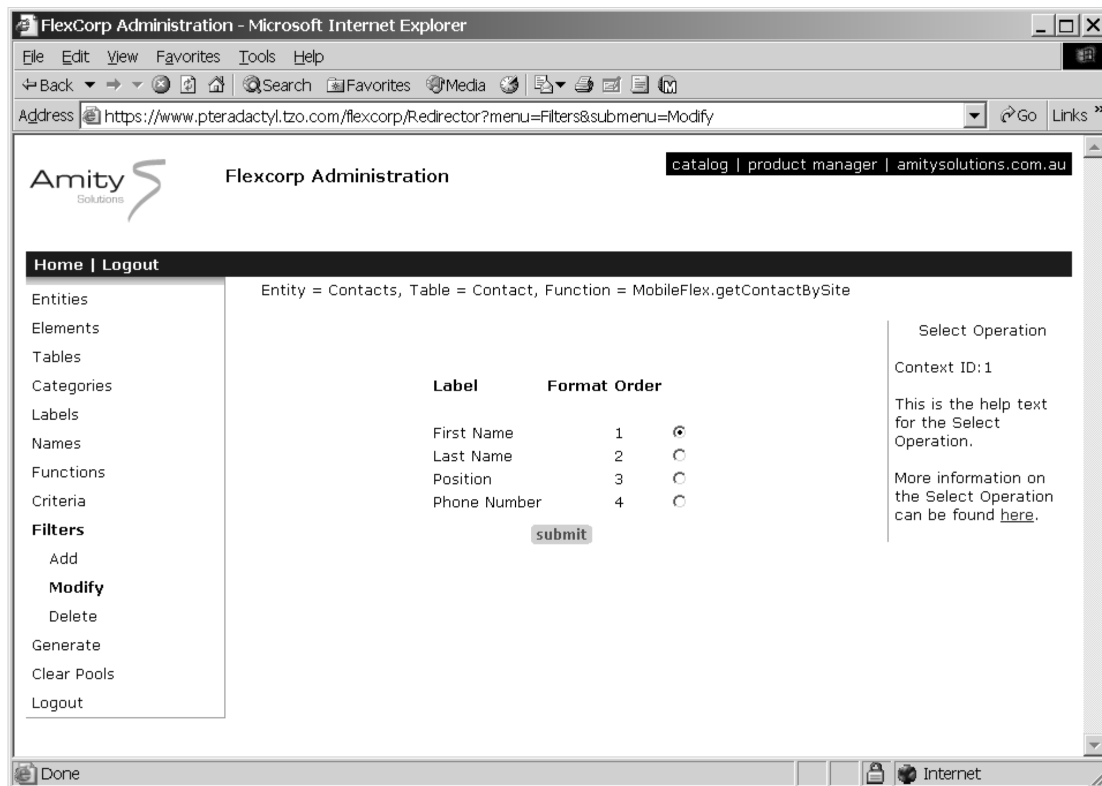
1.1 Using existing infrastructure

For the purposes of demonstrating the web service we will use existing infrastructure from MobileFlex, as web services do not often exist without other data interfaces such as existing applications. We will show how to assemble XML content for contacts related to a field service site.

We already have a predefined function for getting the list of contact information for a field service site as it is used in the MobileFlex. The cluster of data is related by the entity *Contacts*. The function is MobileFlex.getContactBySite. The criterion for the function to retrieve a list of contacts is that their reference site matches the first variable passed to the function. This is shown in the FlexCorp management centre below.



The sequence and composition of data returned by the function is also controlled within the FlexCorp management centre. We define the data required to be some basic contact details and we define the order in which they should be returned.



The mechanics of exchanging data is easier these days because of developments in web service infrastructure. We have adopted Axis as we are less interested in the actual protocols for transporting data. This saves us a lot of plumbing work.

Axis is already integrated into the IBM WebSphere Application Developer (WSAD) and the WebSphere Application Server (WAS) so you will find support for this web services infrastructure. We use Axis within a Tomcat framework and our development is conducted within NetBeans. The Ant make utility delivers adequate build and deploy services to get our web services running.

We assume that the reader is familiar with XML terminology relating to documents and elements.

1.2 Basic web service implementation

Axis gives a few choices in terms of the operation for generation of the response content for a web service. However, the best and most flexible fit for FlexCorp is the message-based service. The reason for this is that the content and order of data elements may be better controlled using the FlexCorp data organisation. These are controlled through the FlexCorp management centre, rather than requiring programmatic changes. The other methods of generating content via Axis require fixed, programmatic alterations of the web service.

1.3 Reading web service requests

In a message-based service, a bit more work is needed to read request information and deal with it. In our implementation the requester provides a text node that specifies the name of the site for which we are retrieving contacts.

The code is set up to get the first child of the request document. We expect this to be a text node so we read the node value and this will be the name of the site for which we are gathering contact information. We also take the opportunity to set up the response document. In Axis, we return an array of elements that gets transformed by Axis into a correct SOAP body response. But in order to properly create the element structure, we need to create the document.

We also sneak in the retrieval of the contact information once we have obtained the site reference.

The fragment of code shown below prepares the way to create the response for this message-based service. The service method getContacts, expects an array of XML elements and returns a set of XML elements. It is very basic to illustrate the mechanics of operation and assumes that elements are enclosed in the request document. This code can be embellished to deal better with unexpected conditions.

Source Editor [ContactService]

```
        // Produce an accessor to the FlexCorp EJB infrastructure
        accessor = new EJBAccessor(new ReferenceFactory());
    }

    public Element[] getContacts(Element[] elements)
    {
        int i;
        int j;
        Classifier[][] specs = null;
        Element detail = null;
        Element contact = null;
        Document document = null;
        String site = "";
        String text = "";
        String tag = "";
        // Elements to be returned to the client by this service
        Element[] e = new Element[1];
        // List of contacts
        Contact[] contacts = new Contact[0];
        try
        {
            // Get the first child of the first element which we expect to be the site reference
            Node node = elements[0].getFirstChild();
            if (node != null)
                site = node.getNodeValue();
            // Create a new document
            DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
            document = builder.newDocument();
            // Get the list of contacts for a site
            contacts = accessor.Contacts(site);
        }
        catch(Exception f)
        {
            f.printStackTrace();
            return new Element[0];
        }
        // Make the first element to be returned the site element of the XML document
```

107:1 INS

1.4 Retrieving information

FlexCorp simply implements the retrieval by passing the function name and the argument to a J2EE infrastructure call. The retrieval will conform to the configuration defined in the FlexCorp management centre. There are some control services to exercise use of the sequencing filter and retrieval of binary information.

For our requirements we do not need the binary data but we do wish to apply the ordering filter. The Boolean values in the framework call, getContacts shown below emphasises this configuration. Note that we make reference to the getContactBySite function to select the information retrieval mode we want for the contacts.

```
    public final Contact[] Contacts(String site)
    {
        if (site == null)
            site = "";
        String[] args = new String[] {function(TEMPLATENAME, "getContactBySite"), site};
        return getContacts(args, null, true, false);
    }
```

1.5 Generating the response document

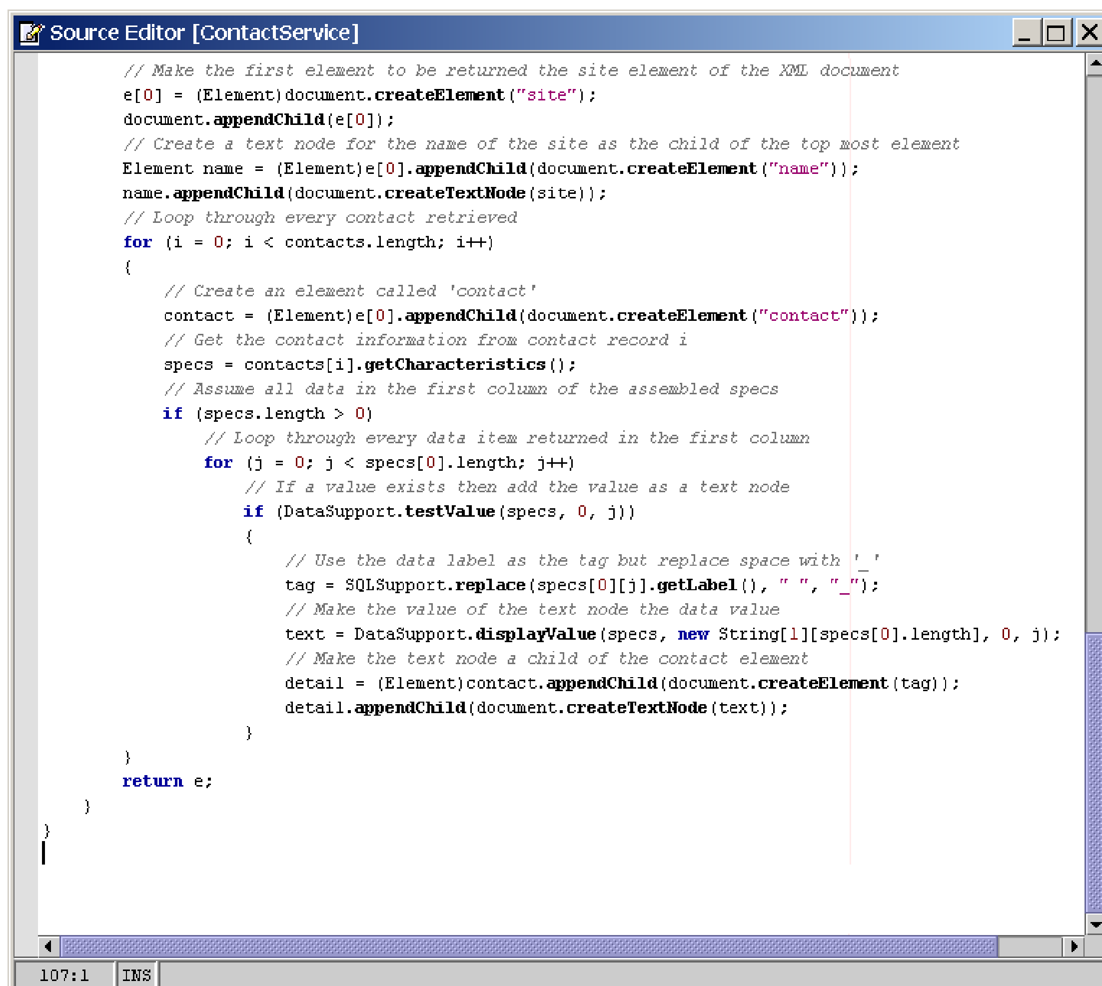
The generation of the document is straightforward. FlexCorp returns the information related to each site as a multi-dimensional array. We could make the generation algorithm very general by stepping through the whole of the array. However, we know the pertinent information is contained in the first column of the array, so we only step through and display the returned elements of that column.

Since we use FlexCorp to control the data elements returned and the order in which they are returned, the generation algorithm does not have to contain too much in the way of interpretation intelligence. We simply keep going through, converting the FlexCorp data label to an XML tag and the FlexCorp data value into a text node value. We replace any space in the FlexCorp data label with an underscore to satisfy the requirements for well formed XML.

The algorithm creates a document structure with 'site' as the root. We add a text node that labels the name of the site. Each contact is made a child of the root and all the data for a contact is stored as text nodes for the contact.

The structure and order of the data, as specified by the FlexCorp function MobileFlex.getContactBySite is first name, last name, position and phone number, independent of the programming used to generate the document.

The code below shows the implementation for the document structure we have described. The response service could be improved by creating a different document should no contacts exist for the search.



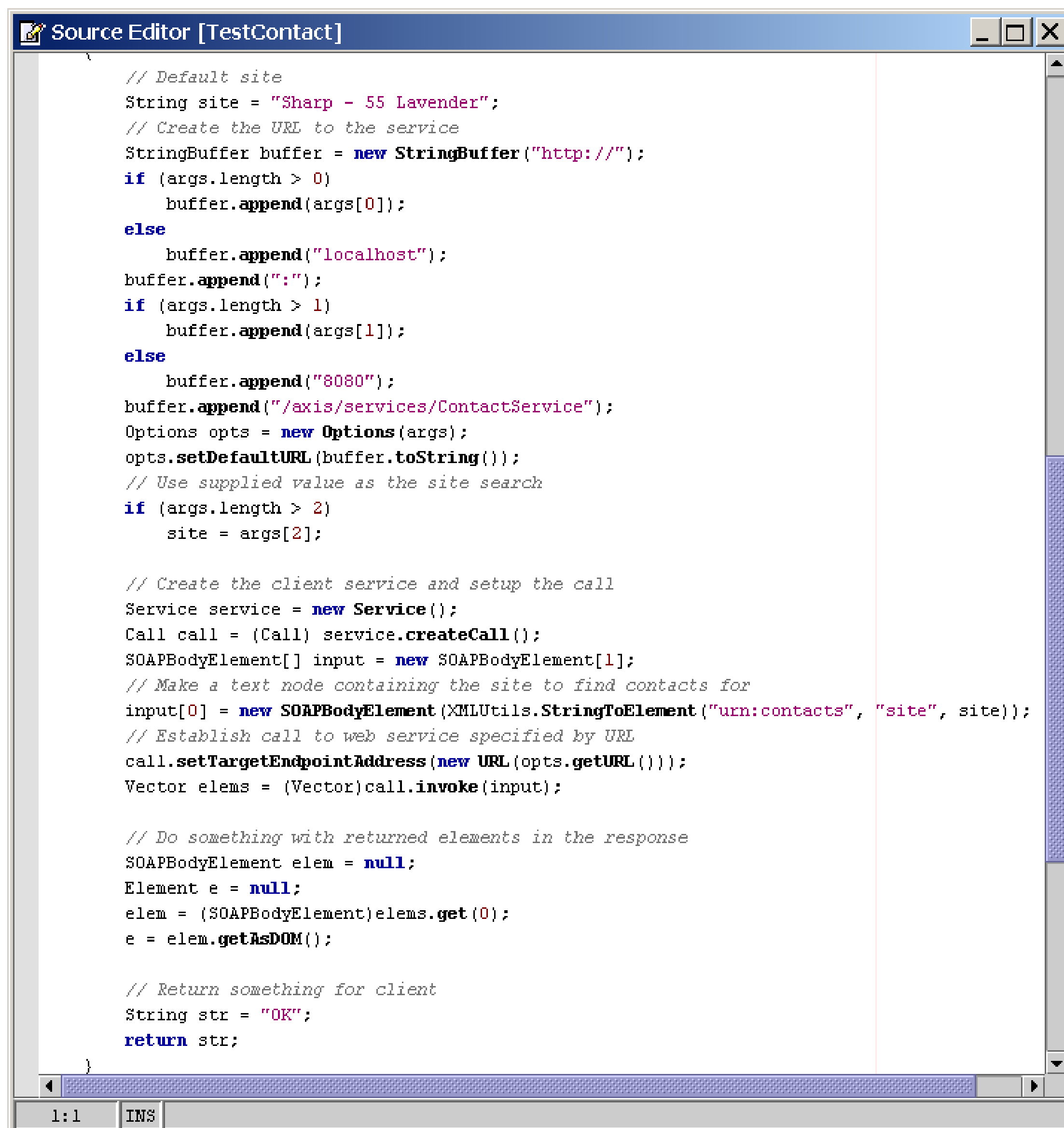
1.6 A simple client

For test purposes, we have written a simple client for message-based web services. The client itself does not do too much other than create a request and retrieve the response. It does not process the response in any way. For the purposes of working through the concepts, we are more concerned with the information transferred across the wire than we are with using the returned information. We leave that for the reader to extend as necessary.

The client code we present here takes some simple command line information for determining the target or endpoint for the request and the nominated site for which we are locating contact information.

Since this is a message-based service we are calling, you will find the code different from those normally presented in the Axis examples. We borrowed most of this code from the message examples provided with the Axis documentation. Note that we send the site information using some of the Axis-specific constructors.

The good thing about Axis is that we do not need to deal with the packaging of the full SOAP document. This leaves us free to concentrate on the actual information we want to send and deal with processing the information when Axis passes us the response.




```
        // Default site
        String site = "Sharp - 55 Lavender";
        // Create the URL to the service
        StringBuffer buffer = new StringBuffer("http://");
        if (args.length > 0)
            buffer.append(args[0]);
        else
            buffer.append("localhost");
        buffer.append(":");
        if (args.length > 1)
            buffer.append(args[1]);
        else
            buffer.append("8080");
        buffer.append("/axis/services/ContactService");
        Options opts = new Options(args);
        opts.setDefaultURL(buffer.toString());
        // Use supplied value as the site search
        if (args.length > 2)
            site = args[2];

        // Create the client service and setup the call
        Service service = new Service();
        Call call = (Call) service.createCall();
        SOAPBodyElement[] input = new SOAPBodyElement[1];
        // Make a text node containing the site to find contacts for
        input[0] = new SOAPBodyElement(XMLUtils.StringToElement("urn:contacts", "site", site));
        // Establish call to web service specified by URL
        call.setTargetEndpointAddress(new URL(opts.getURL()));
        Vector elems = (Vector)call.invoke(input);

        // Do something with returned elements in the response
        SOAPBodyElement elem = null;
        Element e = null;
        elem = (SOAPBodyElement)elems.get(0);
        e = elem.getAsDOM();

        // Return something for client
        String str = "OK";
        return str;
    }
```

1.7 Web service document exchange

Axis provides a way to view the actual document exchanges through the tcpmon utility. WSAD incorporates the tcpmon utility within the development environment, allowing you to perform testing interactively. However, you can manually operate the system quite easily.

The utility is probably the most useful in helping you debug your web service. You send your service request to tcpmon which is set to listen on a particular port. In terms of configuration, tcpmon is a proxy for your request. Your request is forwarded by tcpmon to the actual web service and the web service sends it response back to tcpmon. Finally, tcpmon returns the response to your client. In the meantime, tcpmon has captured the actual XML documents exchanged between your client and the actual web service.

For this particular example the request we send from the web services client contains the text node with the site for which we are finding contacts. Axis creates the SOAP envelope and header for us. The request details are shown in the following tcpmon extract.

```
POST /axis/services/ContactService HTTP/1.0
Content-Type: text/xml; charset=utf-8
Accept: application/soap+xml, application/dime, multipart/related, text/*
User-Agent: Axis/1.1RC2
Host: poseidon.hsys.com.au
Cache-Control: no-cache
Pragma: no-cache
SOAPAction: ""
Content-Length: 331

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <soapenv:Body>
         <ns1:site xmlns:ns1="urn:contacts">Sharp - 55 Lavender</ns1:site>
      </soapenv:Body>
   </soapenv:Envelope>
```

Based on the request, our web service generates a response that lists the contacts for that site if any contacts exist. Again, Axis deals with the SOAP envelope and headers, leaving us free to deal with only the body. The structure and contents of the response is shown in the following diagram. Note that the order of data elements for the contact information is in exactly the order we have specified in the FlexCorp management centre. Should we require new contact information in the future or change the order of the information, we are able to add this without changing the programming in the service.

```
HTTP/1.1 200 OK
Date: Sat, 22 Mar 2003 15:59:50 GMT
Server: Apache Coyote/1.0
Content-Type: text/xml; charset=utf-8
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
   <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
                     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
      <soapenv:Body>
         <site>
            <name>Sharp - 55 Lavender</name>
            <contact>
               <First_Name>Brian</First_Name>
               <Last_Name>Hennessy</Last_Name>
               <Position>Regional Operations Manager</Position>
               <Phone_Number>02 9090 8000</Phone_Number>
            </contact>
            <contact>
               <First_Name>Hasan</First_Name>
               <Last_Name>Odemis</Last_Name>
               <Position>Building Services Coordinator</Position>
               <Phone_Number>02 9954 4684</Phone_Number>
            </contact>
            <contact>
               <First_Name>Paul</First_Name>
               <Last_Name>Birch</Last_Name>
               <Position>Northern Area Supervisor</Position>
               <Phone_Number>99224429</Phone_Number>
            </contact>
         </site>
      </soapenv:Body>
   </soapenv:Envelope>
```

1.8 Demonstration service

For the purposes of testing the actual service yourself, you can run this yourself by downloading the package from our web site at http://www.amitysolutions.com.au/pages/downloads.html. To make things very simple, we have packaged all the Axis support libraries into the contact.jar distribution so you don't need to include all the libraries manually. However, this makes the download fairly large.

You will require a Java runtime installed on your machine. The first thing is to run the tcpmon utility. We suggest that you set tcpmon to listen locally on your machine on port 8080. Redirect received requests to port 80 of www.pteradactyl.tzo.com. At the command line, run the following:

java –cp contact.jar org.apache.axis.utils.tcpmon 8080 www.pteradactyl.tzo.com 80

Now in another command line window, you can run the client with the following:

java –jar contact.jar localhost 8080 "Sharp – 55 Lavender"

If you have set up everything correctly, you should see an exchange of information as your client communicates through tcpmon to our demonstration service.