CHAPTER 8

Integrating with the Enterprise Using Web Services

What Is a Web Service?

A Web service is a self-contained module of application logic that can be exposed to other applications over a network, usually the Internet. A Web service interface hides the technology used to implement the application logic, so that an application implemented in one particular technology can be accessed by using a Web service by another application implemented in a different technology, provided both adhere to the specification of the Web service interface.

There are several competing standards for describing Web services, but the best known is the Web Services Description Language (WSDL),[1] an XML format that describes XML-based services as an abstract set of methods, a binding of these abstract methods to a particular protocol (transport and data format used to invoke them), and the endpoint of the service. A commonly used binding extension to WSDL is SOAP, an XML protocol for invoking a service, usually (but not limited to) using HTTP.

1. For an introductory article on WSDL, refer to `http://www.ibm.com/developerworks/webservices/library/ws-soap/index.html?dwzone=ws`. The WSDL specification can be found at `http://www.w3.org/TR/wsdl`.

SOAP[2] supports two interaction styles: remote procedure call (RPC) and document literal. RPC style is similar to invoking an object's method to perform some procedural functionality. RPC style is suited to synchronous interactions with a published API. In document style, any XML document can be exchanged. The document need not be limited to the SOAP body specification, and need not be predefined. Document style is suited to asynchronous interactions, where a contract is not necessarily published beforehand.

In this chapter, we will use Web services that are modules of application logic invoked over the network, using SOAP in RPC style over HTTP and described with WSDL.

The benefits of Web services include:

- Messages to and from a Web service are easy to read.
- HTTP is almost always allowed through corporate firewalls.
- Web services promise to be ubiquitous. They are based on XML and HTTP, which are widely supported by a broad range of devices with varying capabilities.
- Just about every vendor supports Web services. Web services are becoming a common mechanism for integrating applications and systems, and tool support for creating them are now in their second generation of maturity.
- They are implementation-neutral. Web services can be implemented in any language on any platform. The server and the client are happily ignorant of the technology or operating system used at the other end of the interaction.

However, there are some limitations (although these will diminish over time):

- In practice, it is not yet straightforward to find a Web service and use it without knowing more about it than what is given in the WSDL file. Implementations vary between servers and clients. Writing a client for a particular Web service is still a matter of trial and error.
- Because they are based on XML, which is rather verbose and takes time and resources to process, Web services are not well suited to extremely constrained devices with limited processing and memory capabilities.

Web services is a vast field, and rapidly expanding. If you want to find out more, a good starting point is IBM's developerWorks (`http://www-106.ibm.com/developerworks/webservices/`).

2. To read the SOAP specification, refer to `http://www.w3.org/TR/SOAP/`.

In this chapter, we will look at how to access functionality provided by a Web service, both from Palm and PocketPC clients. To get an end-to-end understanding, we will go through the process of developing the Web service, deploying it on the server, and demonstrating several ways of accessing it from a PDA.

The ImageService Web Service

The ImageService Web service is a simple service designed to allow clients to retrieve binary images from the server. There are two methods a client can invoke:

- Get the names of available images. This method returns the names of images available on the server for retrieval.
- Get an image with a specified name. This method returns an object representing the image specified by the client by name.

Since SOAP is based on XML, a textual format, a means by which binary information such as an image can be sent using SOAP must be used. A common mechanism is to encode the image using Base64. Base64 is a standard originally developed for sending binary objects in emails and specified in the Multipurpose Internet Mail Extensions (MIME) standard.[3] Base64 provides a means to encode a binary object into a string, and also to decode the string back into a binary object. This is done by treating each set of three 8-bit bytes of binary data as a set of four 6-bit groups, and mapping each group of six bits to a printable subset of the ASCII character set. Each 6-bit number is used to look up the corresponding character in the Base64 alphabet. Table 8.1 shows the Base64 alphabet.

The characters thus obtained are concatenated to form the encoded string.[4] The string is decoded back into the binary object by the reverse process. If the decoder comes across a character in the string that is not in the Base64 alphabet, the decoder must ignore the character and continue with the next character.

The ImageService interface has two methods:

```
public String[] getNames(String extension);
public ImageValue getImage(String name);
```

3. For more information on Base64, refer to the MIME specification `http://www.faqs.org/rfcs/rfc2045.html`, section 6.8.

4. In MIME, the encoded string is broken into lines of no more than 76 characters each. In SOAP, this is not required and the encoded string can be continuous.

Table 8.1 The Base64 Alphabet[a]

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

a. This is Table 1 from RFC2045 at `http://www.faqs.org/rfcs/rfc2045.html`.

The `getImage` method returns an `ImageValue` object. This is partly to demonstrate how complex objects (versus primitive types) are sent across SOAP. The object has two attributes for the sake of demonstration: a `long`, representing the date the image was last modified, and a `String`, representing the Base64 encoded image.

For the purposes of SOAP serialization, the `ImageValue` supports a Java bean interface style. That is, it has a constructor with no arguments, and setters and getters for its attributes.

The `ImageValue` class is defined thus:

```
package com.javaonpdas.webservices;

public class ImageValue {
  public long dateAsLong;
  public String encodedImage;

  public ImageValue() {
  }

  public ImageValue(long dateAsLong, String encodedImage) {
    this.dateAsLong = dateAsLong;
    this.encodedImage = encodedImage;
  }

  public long getDate() {
    return this.dateAsLong;
  }

  public void setDate(long dateAsLong) {
    this.dateAsLong = dateAsLong;
  }

  public String getEncodedImage() {
    return this.encodedImage;
  }

  public void setEncodedImage(String encodedImage) {
    this.encodedImage = encodedImage;
  }
}
```

The `getNames` method returns a `String` array with the file names of image files with the specified extension in a particular directory. The directory name is hard-coded for the purposes of this example.

The `ImageService` class is listed in the following.

```
package com.javaonpdas.webservices;

import java.util.Date;
import java.io.File;
import java.io.FileInputStream;
import java.io.FilenameFilter;
```

```
public class ImageService {
  private String directory =
    "C:\\JavaOnPDAs\\Desktop\\resources";
  public ImageValue getImage(String name) {
    String encodedImage = null;
    long timeStamp = 0;
    FileInputStream fis = null;
    try {
      // read the file into a byte array
      File imageFile = new File(directory + "\\" + name);
      if (imageFile.exists()) {
        timeStamp = imageFile.lastModified();
        fis = new FileInputStream(imageFile);
        int length = fis.available();
        byte[] rawImage = new byte[length];
        fis.read(rawImage);
        // encode the byte array into a Base64 string
        encodedImage = org.apache.axis.encoding.Base64.encode(
          rawImage);
      }
    }
    catch (Exception e) {
      System.out.println("Error:" + e);
    }
    finally {
      try { if (fis != null) fis.close(); }
      catch (Exception e) {}
    }
    ImageValue image = new ImageValue(timeStamp, encodedImage);
    return image;
  }

  public String[] getNames(String extension) {
    final String ext = extension;
    FilenameFilter filter = new FilenameFilter() {
      public boolean accept(File dir, String name) {
        return name.endsWith(ext);
      }
    };
    File dir = new File(directory);
    String[] files = dir.list(filter);
    return files;
  }
}
```

Some important points about this code:

- Since the Web service will be deployed on Axis, we will make use of the built-in `Base64` class with a static encode method, `org.apache.axis.encoding.Base64`.
- This is a "normal" Java class definition. Apart from making use of the Axis `Base64` class (which could be replaced by any Base64 encoder), there is nothing here to suggest that this class will be deployed as a Web service. The publication of this class's methods as a Web service interface is completely separate to the class definition.

Now that we have a class that will serve as our Web service, the next step is to write a Web service deployment descriptor (WSDD). Web service deployment descriptors conform to a standard XSD.

For our `ImageService` class, the WSDD is quite simple. The WSDD describes:

- The name of the Web service.
- The class of the Web service.
- The methods of the class that are allowed to be accessed.
- That the service uses the built-in bean serializer for converting an object to a SOAP representation, by accessing its attributes and converting those base types to SOAP format.
- The name of the class to serialize, using the bean serializer.

The WSDD for the `ImageService` Web service is `deploy-ImageService-AXIS.wsdd`, and looks like this:

```
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
       xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

   service name="ImageService" provider="java:RPC">
      parameter name="className"
         value="com.javaonpdas.webservices.ImageService"/>
      <parameter name="allowedMethods" value="*"/>
      <beanMapping qname="ns:ImageValue"
         xmlns:ns="urn:BeanService"
         languageSpecificType=
           "java:com.javaonpdas.webservices.ImageValue"/>
   </service>

</deployment>
```

Now we're ready to set up Axis and Tomcat, which we will use to deploy and test the `ImageService` Web service.

Setting Up Axis and Tomcat

Axis (`http://xml.apache.org/axis/index.html`) is an Apache XML open source project, and is the successor to Apache SOAP, one of the earliest SOAP implementations. Axis is a Java implementation of a SOAP server and client. Axis operates within an application server or servlet engine, and one of the most popular servlet engines is another Apache open source project, Tomcat (`http://jakarta.apache.org/tomcat/index.html`). Tomcat is the reference implementation of the Java Server Pages (JSP) 1.2 and Servlet 2.3 specifications. Axis can use just about any servlet engine, provided it supports version 2.2 (or greater) of the servlet specification.

Download Axis from `http://xml.apache.org/axis/releases.html`. The examples in this book use Axis 1.0.

Download Tomcat from `http://jakarta.apache.org/builds/jakarta-tomcat-4.0/release/v4.1.12/`. The examples in this book use Tomcat 4.1.

Tomcat Installation[5]

Unzip the Tomcat binary package into a convenient location. We will refer to this location as `${tomcat-base}`. Change to the directory `${tomcat-base}\bin`, and run the batch file `startup.cmd` to start Tomcat.

By default, Tomcat runs on port 8080. As Tomcat is starting, you should see information similar to the following:

```
16/11/2002 13:36:29 org.apache.commons.modeler.Registry loadRegistry
INFO: Loading registry information
16/11/2002 13:36:30 org.apache.commons.modeler.Registry getRegistry
INFO: Creating new Registry instance
16/11/2002 13:36:31 org.apache.commons.modeler.Registry getServer
INFO: Creating MBeanServer
16/11/2002 13:36:32 org.apache.coyote.http11.Http11Protocol init
INFO: Initializing Coyote HTTP/1.1 on port 8080
Starting service Tomcat-Standalone
Apache Tomcat/4.1.10
16/11/2002 13:36:43 org.apache.coyote.http11.Http11Protocol start
INFO: Starting Coyote HTTP/1.1 on port 8080
16/11/2002 13:36:43 org.apache.jk.common.ChannelSocket init
INFO: JK2: ajp13 listening on tcp port 8009
16/11/2002 13:36:43 org.apache.jk.server.JkMain start
```

5. This section assumes that you have already installed JDK 1.4, and that `JAVA_HOME` is pointing to it.

```
INFO: Jk running ID=0 time=10/90  config=C:\jakarta-tomcat-
4.1.10\bin\..\conf\jk2.properties
```

Once Tomcat has started, test that it is working by pointing your browser to `http://localhost:8080/`. You should see something similar to Figure 8.1.

To shut down Tomcat, run the `shutdown.cmd` batch file.

Axis Installation

Unzip the Axis ZIP file into a convenient location. We will refer to this location as `${axis-base}`. In the directory `${axis-base}\webapps`, there is a directory called axis. This directory contains the Axis Web application, which can be copied over to the `webapps` directory of the servlet engine (Tomcat, in this case). So, copy `${axis-base}\webapps\axis` to `${tomcat-base}\webapps`.

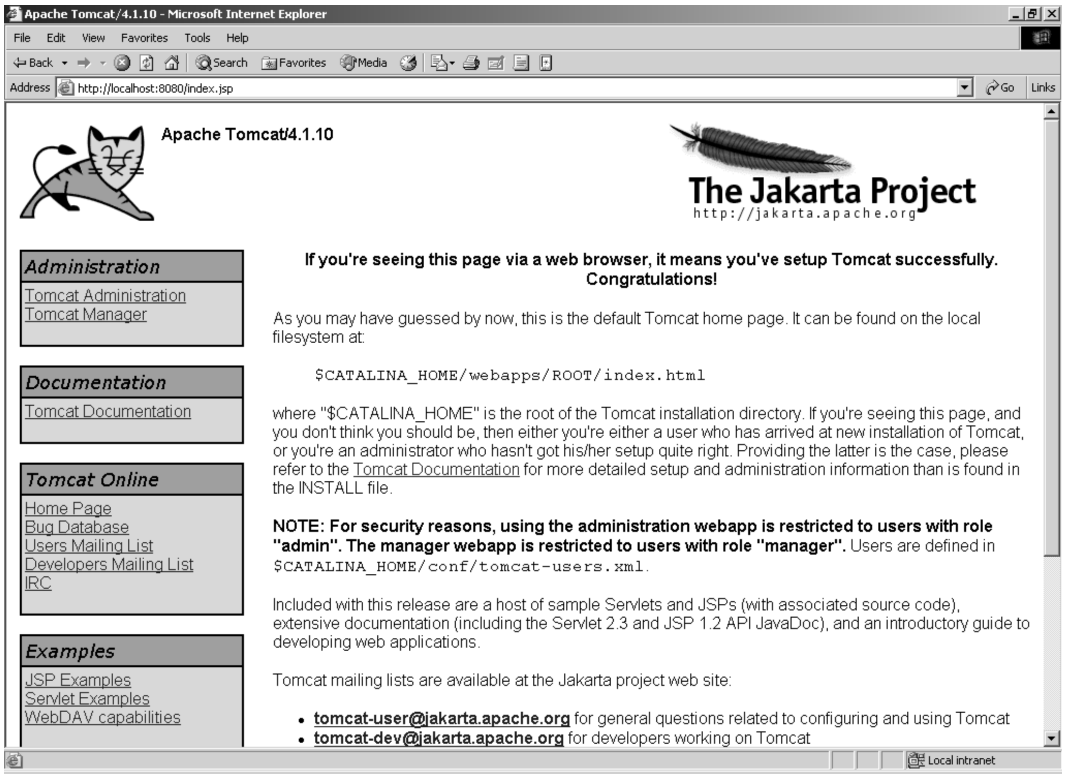

Figure 8.1 The Tomcat test page

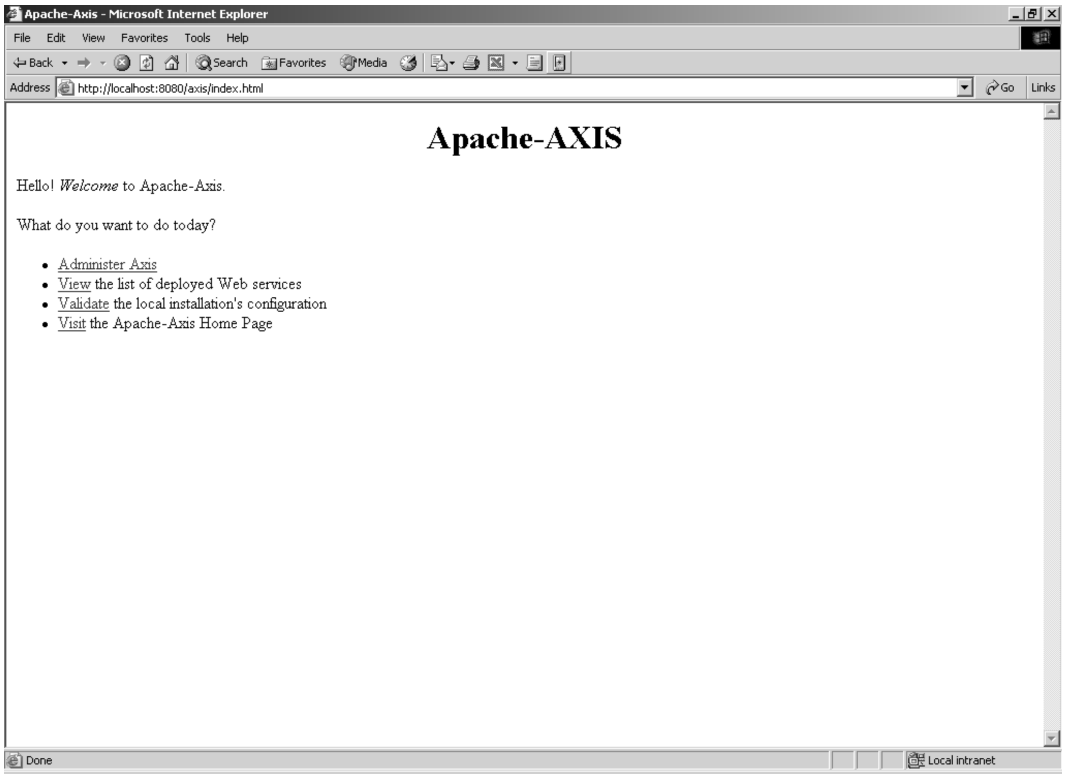

Figure 8.2 The Axis test page

To test the installation, point your browser to `http://127.0.0.1:8080/axis/`. You should see a page similar to Figure 8.2.

Check whether AXIS is properly configured and that it can find all the components it requires by clicking on the Validate link. You should see a page like that shown in Figure 8.3.

Note that the page should say that all the needed components can be found.

Now we are ready to deploy the Web service.

Deploying the ImageService Web Service

In Tomcat, each Web application has a `WEB-INF` subdirectory, in which class files or JAR files specific to the application are located. This is done so that each Web application can have a set of components (classes or JAR files) independent of other Web applications on the same instance of Tomcat. Axis will

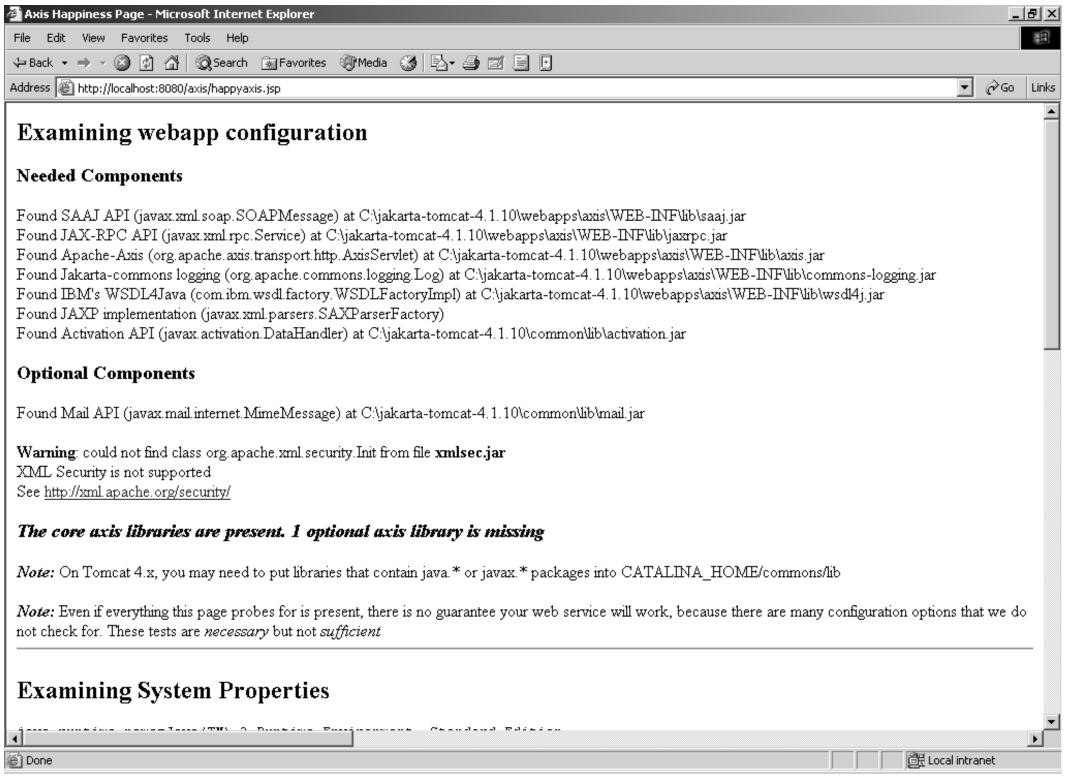

Figure 8.3 The Axis configuration page

need to find our Web service class, so we need to bundle a JAR file and put it in our Web application's `lib` directory. This is done with the `CompileDesktop` target in the Ant build file:

```
<target name="CompileDesktop" depends="Init">
  <!-- Compile the Desktop source code -->
  <javac
    srcdir="${desktopsrc}"
    destdir="${desktopdest}"
    debug="off">
    <classpath>
      <pathelement path="${desktopdest}"/>
      <path refid="axis.path"/>
    </classpath>
  </javac>
  <jar jarfile="${desktoplib}\javaonpdas-desktop.jar"
```

```
        basedir="${desktopdest}"
        includes="**\*.class"
    />
    <copy file="${desktoplib}\javaonpdas-desktop.jar"
          todir="${axis-webapp}" />
    <copy file="${desktoplib}\javaonpdas-desktop.jar"
          todir="${proxy-webapp}" />
  </target>
```

This target compiles all the source files under the desktop directory, creates the JAR `javaonpdas-desktop.jar`, and copies it to the Axis `webapp` directory. Stop Axis and start it again to make sure it has seen the new JAR file.

Now that Axis can see the classes that implement the Web service, we can deploy it.

Deploying the Web Service

In this step we make use of the deployment descriptor we prepared previously. Axis has a deployment client that we will use, which is `org.apache.axis.client.AdminClient`. The Ant build target `DeployImageService` performs this task:

```
  <target name="DeployImageService" depends="CompileDesktop">
    <!-- Run the ImageService web service -->
    <java
      classname="org.apache.axis.client.AdminClient"
      dir="."
      fork="true"
      failonerror="true">
      <classpath>
        <pathelement path="${desktopdest}"/>
        <path refid="axis.path"/>
      </classpath>
      <arg line="-l${webserviceadmin} ${desktopsrc}\com\javaonpdas\↵
      webservices\deploy-ImageService-AXIS.wsdd"/>
    </java>
  </target>
```

Running this task will tell Axis about the new Web service:

```
C:\JavaOnPDAs>ant DeployImageService
Buildfile: build.xml

Init:

CompileDesktop:
```

```
DeployImageService:
     [java] - Processing file .\Desktop\src\com\javaonpdas\
webservices\deploy-ImageService-AXIS.wsdd
     [java] - <Admin>Done processing</Admin>

BUILD SUCCESSFUL
Total time: 8 seconds
```

If you see any exceptions thrown during this step, it is likely that Axis cannot find a class referenced in the deployment descriptor. If you do see such an exception, check that the JAR file is in the Axis webapp `lib` directory, and try restarting Axis and redeploying the Web service.

Testing

The first step is to check whether Axis thinks that the Web service has been properly deployed. Point your browser at the main Axis page (`http://127.0.0.1:8080/axis/index.html`), and click the `View` link. This will display all the Web services deployed on this server, as shown in Figure 8.4.

Click on the WSDL link to display a description of the service and how to invoke it. While the WSDL may seem a bit daunting, it describes the operations (i.e., methods) of the Web service that can be invoked (`getNames` and `getImage`), the transport to use to invoke them (HTTP), and the location of the service (`http://127.0.0.1:8080/axis/services/ImageService`).

The next step is to test the Web service from a client. To do so from the desktop, the command line application `com.javaonpdas.client.ImageServiceClient` calls the Web service and calls both methods, displaying information about what is returned.

Using the Axis client (as we will see later, other SOAP clients have slight variations), the steps to calling a Web service are:

1. Set up the `Call` object. This involves telling the `Call` object the end point URL, the name of the service, and the method to call.

```
Service service = new Service();
Call getNamesCall = (Call)service.createCall();
getNamesCall.setTargetEndpointAddress(endPointURL);
getNamesCall.setOperationName(
  new QName("ImageService", "getNames"));
getNamesCall.addParameter("extension",
  org.apache.axis.encoding.XMLType.XSD_STRING,
  ParameterMode.IN);
getNamesCall.setReturnType(XMLType.SOAP_ARRAY);
```

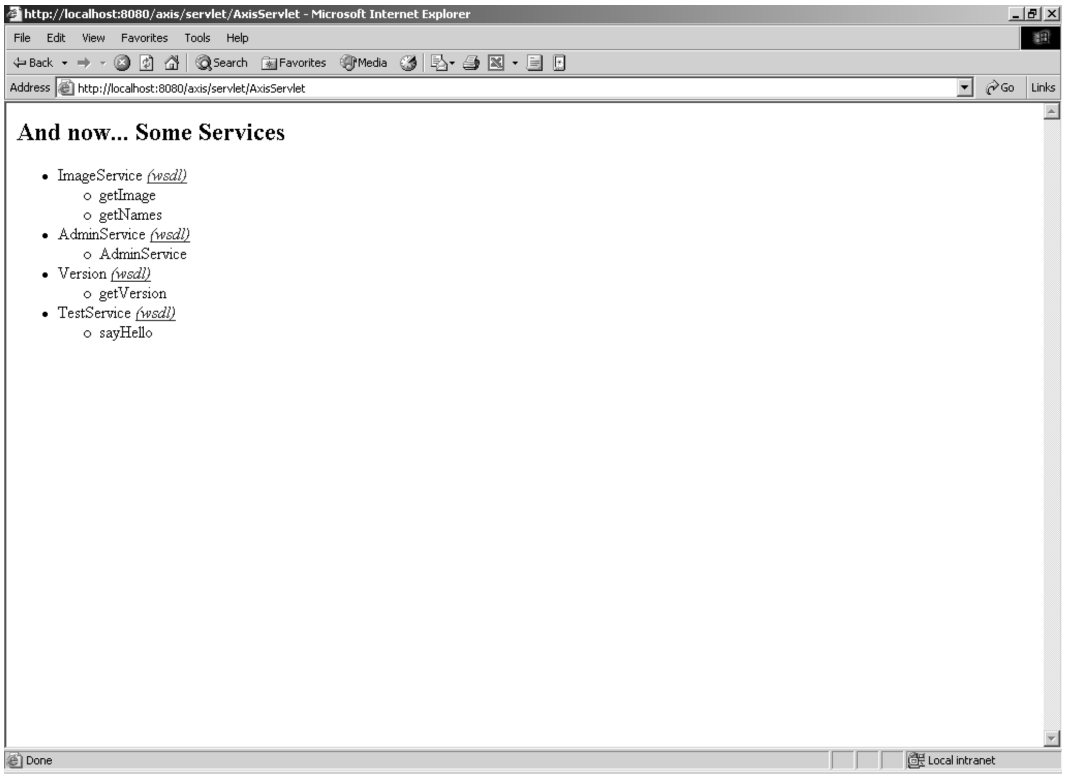

Figure 8.4 Web services deployed on Axis

2. Invoke the call. In this case, the `getNames` method returns an array of `String`, so we need to cast the invoke method to `String[]`.

```
String[] names = (String[])getNamesCall.invoke(
  new Object[] { ".png" });
```

3. Process the response.

```
if (names == null) {
   System.out.println("The array of names is null");
}
else {
   System.out.println("Image names:");
   for (int i=0; i<names.length; i++)
      System.out.println("  " + names[i]);
}
```

4. Catch exceptions during these calls. Axis will throw an `AxisFault` exception if something goes wrong, so we need to make sure we catch it. In this case, we will just send it to `System.out`.

```
catch (AxisFault fault) {
   System.err.println("Generated fault: ");
   System.out.println("  Fault Code   = " +
     fault.getFaultCode());
   System.out.println("  Fault String = " +
     fault.getFaultString());
}
```

The Ant build file has a target to run this test client, called `RunImageServiceClient`. It appears like this:

```
<target name="RunImageServiceClient" depends="CompileDesktop">
  <java
     classname="com.javaonpdas.client.ImageServiceClient"
     dir="."
     fork="true"
     failonerror="true">
     <classpath>
       <pathelement path="${desktopdest}"/>
       <path refid="axis.path"/>
     </classpath>
     <arg line="${testendpoint}"/>
  </java>
</target>
```

Run the test client by typing `ant RunImageServiceClient` on the command line:

```
C:\JavaOnPDAs>ant RunImageServiceClient
Buildfile: build.xml

Init:

CompileDesktop:

RunImageServiceClient:
     [java] Image names:
     [java]   kookaburra.png
     [java]   kookaburra-bw.png
     [java]   kangaroo-bw.png
     [java]   kangaroo1.png
     [java]   kangaroo2.png
     [java]   kangaroo3.png
```

```
     [java] Image returned is kangaroo1.png
     [java]   image date is Mon, 04 Nov 2002 21:08:50
     [java]   image is 1458 bytes long

BUILD SUCCESSFUL
Total time: 7 seconds
```

Monitoring HTTP Traffic

Axis comes with a very useful debugging tool, called `TcpMon`. `TcpMon` allows you to view the request sent from the client and the response received from the server. The way it works is to create a TCP/IP "tunnel" between a dummy port to which the client is pointed, and the real port on which the server is running. For example, we can tell `TcpMon` to listen on port 5555 and to connect clients to port 8080 (where Tomcat listens), and to display all traffic going between the two ports, such as our Web service requests and responses.

To start `TcpMon`, the Ant build file has a target called `RunTcpMon`. It starts on the port `${test-axis-port}`, which is set to 5555. The Ant target looks like this:

```
<target name="RunTcpMon">
  <java
     classname="org.apache.axis.utils.tcpmon"
     dir="."
     fork="true"
     failonerror="true">
     <classpath>
       <path refid="axis.path"/>
     </classpath>
     <arg line="${test-axis-port} localhost ${axis-port}"/>
  </java>
</target>
```

To start it, type `ant RunTcpMon` on the command line. When it starts, `TcpMon` looks like the window shown in Figure 8.5.

To tell the `ImageServiceClient` that we want to connect to port 5555, make sure that the Ant build file has the following value for the `testendpoint` property:

```
<property name="testendpoint"
value="http://localhost:${test-axis-port}/axis/servlet/AxisServlet"/>
```

When you run the client, you should see `TcpMon` display the request and response, as shown in Figure 8.6.

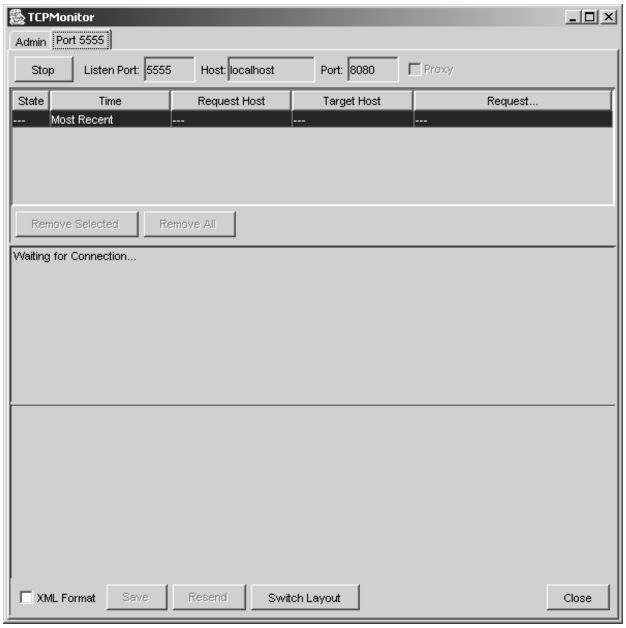

Figure 8.5 The TCPMonitor window

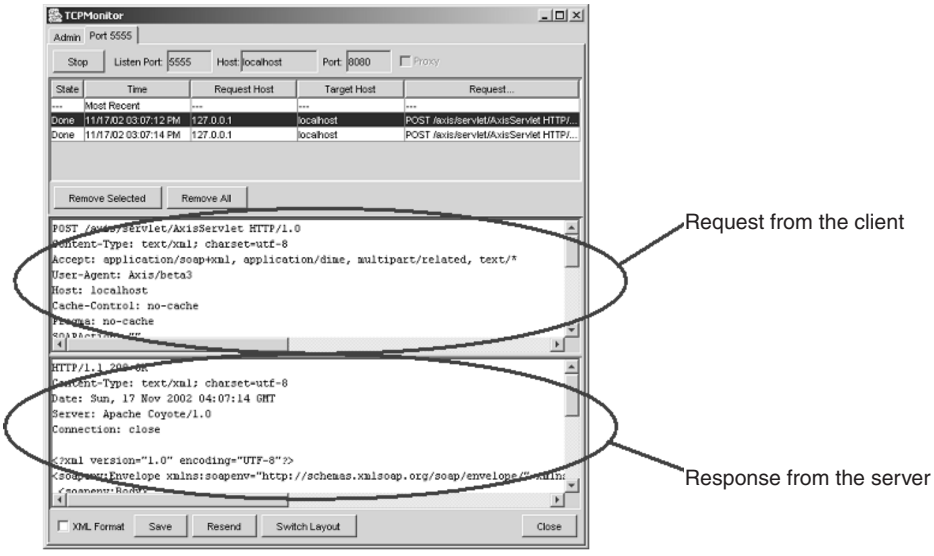

Figure 8.6 The TCPMonitor window showing an example session

The `ImageServiceClient` sends two separate requests to the server. The first is to retrieve the list of image names, and it looks like this:

```
POST /axis/servlet/AxisServlet HTTP/1.0

Content-Type: text/xml; charset=utf-8

Accept: application/soap+xml, application/dime, multipart/related,
text/*

User-Agent: Axis/beta3

Host: localhost

Cache-Control: no-cache

Pragma: no-cache

SOAPAction: ""

Content-Length: 446


<?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:getNames soapenv:encodingStyle="http://schemas.xmlsoap.org/
soap/encoding/" xmlns:ns1="ImageService">
   <extension xsi:type="xsd:string">.png</extension>
  </ns1:getNames>
 </soapenv:Body>
</soapenv:Envelope>
```

And the response looks like this:

```
HTTP/1.1 200 OK

Content-Type: text/xml; charset=utf-8

Date: Sat, 07 Dec 2002 12:21:33 GMT

Server: Apache Coyote/1.0

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>
  <ns1:getNamesResponse soapenv:encodingStyle="http://
schemas.xmlsoap.org/soap/encoding/" xmlns:ns1="ImageService">
   <getNamesReturn xsi:type="soapenc:Array"
soapenc:arrayType="xsd:string[4]" xmlns:soapenc="http://
schemas.xmlsoap.org/soap/encoding/">
    <item>kookaburra.png</item>
    <item>kookaburra-bw.png</item>
    <item>kangaroo-bw.png</item>
    <item>kangaroo1.png</item>
   </getNamesReturn>
  </ns1:getNamesResponse>
 </soapenv:Body>
</soapenv:Envelope>
```

Undeploying the Web Service

To undeploy the Web service, we again use the Axis administration client. The Ant build file defines a target to undeploy the image service, called `UndeployImageService`. The target is defined as follows:

```
  <target name="UndeployImageService">
    <!-- Undeploy the ImageService web service -->
    <java
       classname="org.apache.axis.client.AdminClient"
       dir="."
       fork="true"
       failonerror="true">
       <classpath>
         <pathelement path="${desktopdest}"/>
         <path refid="axis.path"/>
       </classpath>
       <arg line="-l${webserviceadmin}↵
${desktopsrc}\com\javaonpdas\webservices\undeploy-ImageService-↵
AXIS.wsdd"/>
    </java>
  </target>
```

Typing the command `ant UndeployImageService` should get the following response:

```
C:\JavaOnPDAs>ant UndeployImageService
Buildfile: build.xml

UndeployImageService:
     [java] - Processing file
.\Desktop\src\com\javaonpdas\webservices\undeploy-
ImageService-AXIS.wsdd
     [java] - <Admin>Done processing</Admin>

BUILD SUCCESSFUL
Total time: 14 seconds
```

To verify that Axis has indeed removed the service, point your browser back to the main Axis page at `http://127.0.0.1:8080/axis/index.html` and click on the `View` link. The `ImageService` should not be listed as a service that Axis knows about. Note that the `undeploy` command merely removes the Web service definition from Axis—it does not remove the JAR file we put in the `webapp lib` directory.

The ImageViewer Client Application

Now that we have tested the ImageService with a desktop client, we will write a PDA client. We will start with a Palm version, and then move on to a PocketPC version. In each case the client will allow the user to

- View a list of names of images available on the server.
- View a list of names of images available on the client.
- Select an image name and view the image.
- Store a remote image on the client.

Before we start writing the Palm version of the `ImageViewer` application, we will explore some options for accessing a Web service from the Palm.

Web Service Access from a Palm Device

Calling a Web Service Directly

In this option, the Palm calls the Web service directly as shown in Figure 8.7.

We will need a SOAP client for J2ME. There are several popular clients available, such as kSOAP (`http://www.ksoap.org`) and Wingfoot (`http://`

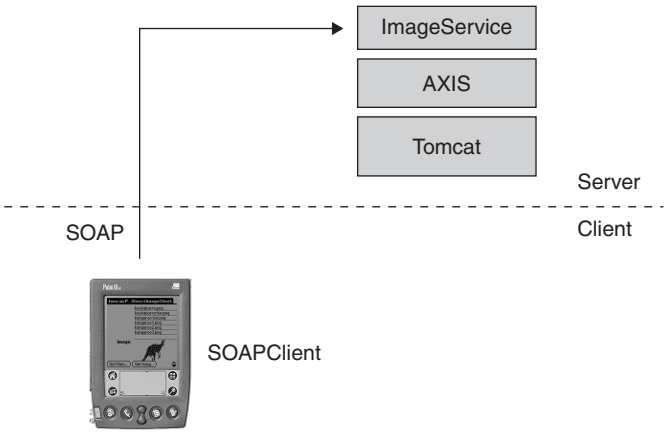

Figure 8.7 Accessing a Web service directly from the Palm device

`www.wingfoot.com`). For this client, we will choose the Wingfoot SOAP client. It also has a J2SE version, which we will use for the PocketPC client.

Download the latest Wingfoot SOAP client from `http://www.wingfoot.com/`. Note that you will need to register by providing your email address and your name. The examples in this book use Wingfoot 1.03.

Unzip the downloaded ZIP file into a convenient location. The ZIP will contain two JAR files: one for J2ME named `kvmwsoap_1.03.jar` and the other for J2SE named `j2sewsoap_1.03.jar`. Copy the J2ME JAR to the `${palm-base}\lib` directory.

In the example application, `com.javaonpdas.webservices.clients.wingfoot.SOAPClient`, the application has two buttons: one to retrieve the names and the other to retrieve the image, specified by a hard-coded name.

When the "Get Names" button is pressed, we create an `Envelope` object and use it to create a `Call` object. The `Call` object represents the service invocation we want to make, so we need to specify the name of the service to invoke and the service's method name.

```
Envelope requestEnvelope = new Envelope();
requestEnvelope.setBody("extension", ".png");
Call call = new Call(requestEnvelope);
call.setMethodName("getNames");
call.setTargetObjectURI("ImageService");
```

Next we set up the transport by telling it the SOAP endpoint. Note that the hostname in the URL is not `localhost`, as the Palm will interpret that as meaning the Palm device rather than the machine where the Web service resides.

```
HTTPTransport transport =
  new HTTPTransport("http://192.168.0.1:8080/axis/servlet/|↵
AxisServlet", null);
transport.getResponse(true);
```

The transport object is used to invoke the call, and the result is assigned to an `Envelope` object.

```
Envelope responseEnvelope = call.invoke(transport);
```

Then the envelope is queried to process its contents. If an error occurred, `isFaultGenerated` will return true. In this case we need to retrieve the fault from the envelope and, in this case, display it on the screen. Otherwise, the response to the Web service invocation is in the 0th parameter, represented as an array of `Object`. These are the names of the images available on the server, so we insert them at the end of the text field and add a new line character.

```
if (responseEnvelope != null) {
   if (responseEnvelope.isFaultGenerated()) {
      Fault f = responseEnvelope.getFault();
      textField.insert("Error: " + f.getFaultString(),
      textField.size());
   }
   else {
      textField.setString(null);
      Object[] parameter =
        (Object[])responseEnvelope.getParameter(0);
      for (int i=0; i<parameter.length; i++)
         textField.insert(parameter[i] + "\n",
           textField.size());
   }
}
```

The code to handle the "Get Image" button press is similar but there are some important extensions. The first difference is in the set up of the envelope, where we set the parameter for the method we want to call.

```
Envelope requestEnvelope = new Envelope();
requestEnvelope.setBody("name", "kangaroo1.png");
```

When we set up the `Call` object, there are some other differences compared to the "Get Names" case above. In this case, the Web service method we want to invoke returns an object we defined, rather than a standard object (i.e., `String`) as was the case in "Get Names." The ImageService's `getImage` method returns an `ImageValue`, so we need to specify that in setting up the `Call`

object using a `TypeMappingRegistry`. The `TypeMappingRegistry` tells the SOAP client how to deal with the `ImageValue` type returned by the `getImage` method.

```
Call call = new Call(requestEnvelope);
call.setMethodName("getImage");
call.setTargetObjectURI("ImageService");
TypeMappingRegistry registry =
  new TypeMappingRegistry();
registry.mapTypes("urn:BeanService", "ImageValue",
  new ImageValue().getClass(),
  new BeanSerializer().getClass(),
  new BeanSerializer().getClass());
call.setMappingRegistry(registry);
```

The `registry.mapTypes` method creates a new entry in the registry that maps between the custom SOAP type `ImageValue` in the namespace `urn:BeanService` and the bean serializer and deserializer class `ImageValue`. The client then knows that the `ImageValue` class is used to serialize and deserialize the type `ImageValue`.

The transport is set up as before.

```
HTTPTransport transport = new HTTPTransport("http://↵
192.168.0.1:8080/axis/servlet/AxisServlet", null);
transport.getResponse(true);
```

And the service is invoked.

```
Envelope responseEnvelope = call.invoke(transport);
```

Next we process the response in a similar way to the "Get Names" case, except this time we are expecting an `ImageValue` instance.

```
ImageValue imageValue = (ImageValue)responseEnvelope.getParameter(0);
```

Now that we have the instance of `ImageValue`, containing the image in encoded form as received from the server, we need to decode the Base64-encoded string back into an image. The Wingfoot SOAP client JAR includes a Base64 encoder and decoder, so we will use that.

To decode the encoded string, we use the `String` constructor of the class `Base64`, and then the getBytes method to retrieve the byte array. The `Image` class has a static method `createImage` that takes a byte array, which we use to create an immutable image for the `ImageItem` on the main screen.

```
Base64 encodedImage =
  new Base64(imageValue.getEncodedImage());
```

```
imageItem.setImage(Image.createImage(
  encodedImage.getBytes(), 0,
  encodedImage.getBytes().length));
```

To run the application, type `ant SOAPClient`. The emulator will start and the screen will look like the screen in Figure 8.8.

Pressing the "Get Names" button will result in the invocation of the `ImageService` Web service, and the Palm will display on the screen the names of images on the server, as shown in Figure 8.9.

Pressing the "Get Image" button will cause the `getImage` method of the `ImageService` service to be invoked, and the image `kangaroo1.png` is displayed on the screen, similar to Figure 8.10.

Using a Proxy with HTTP to Access the Web Service

Having a SOAP client on the Palm device means that there is less space for your application, and valuable processing resources are used to generate the SOAP request and parse the response. Often in constrained environments, we need to find a way to off-load processing from the device if at all possible. In this section, we look at one such alternative. Off-loading processing from the

Figure 8.8 The SOAPClient window

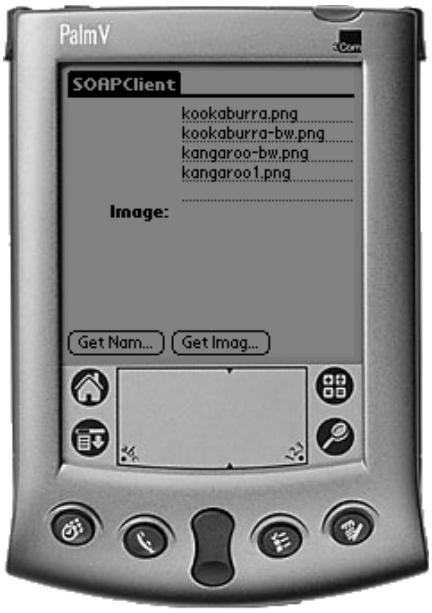

Figure 8.9 Image names returned by the Web service

Figure 8.10 An image returned from the Web service

device means that we need to do some more processing on the server side. The server side is relatively unconstrained in processing resources, so it makes sense to do more there than on the constrained mobile device. We need some processing to be done on the server on behalf of the Palm device. The processing interacts with the Web service, and the Web service is unchanged. The results of this processing are sent to the device in a form that it can process with a minimum of effort. The logic to perform this processing is called a proxy.

One approach to building a proxy to interact with the `ImageService` on the Palm's behalf is to use a servlet. A servlet provides classes that make it very easy to create server-side logic accessed with HTTP as shown in Figure 8.11.

The Palm will invoke the servlet using HTTP GET and a URL, and retrieve the information as text in a Web page. The URL will embed some parameters that form the protocol between the Palm and the proxy. The first parameter is the service endpoint (the name is "service-end-point"), the value of which is a URL that tells the proxy where to find the `ImageService` Web service. The second parameter named "action" tells the proxy which `ImageService` method to invoke. The values will be "`getNames`" and "`getImage`." If the action is "`getImage`," another parameter named "`name`" has the value of the image to retrieve.

The first thing is to create a class that extends `HttpServlet`:

```
public class ImageServiceProxy extends HttpServlet {

}
```

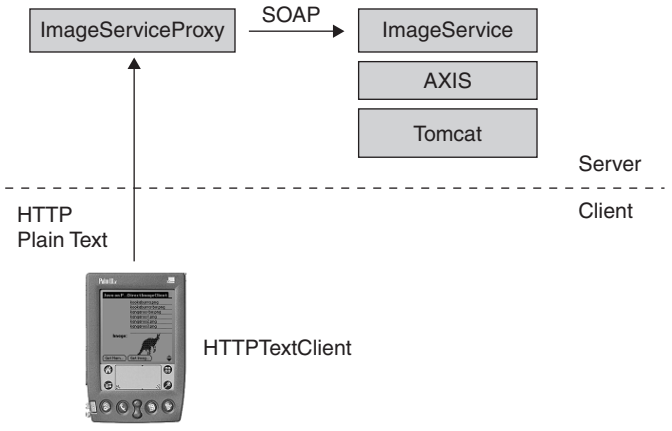

Figure 8.11 Accessing the Web service via a server-side proxy

Next we will implement the doGet method. This method gets the values of the parameters and invokes the ImageService methods accordingly. It then writes the response to the servlet's output stream.

```
public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws IOException,
    ServletException {
   URL endPointURL = new URL(request.getParameter("service-end-point"));
   String action = request.getParameter("action");
   if (action.equalsIgnoreCase("getNames")) {
      String[] names = getNames(endPointURL);
      response.setContentType("text/plain");
      PrintWriter out = response.getWriter();
      if (names != null) {
         for (int i=0; i<names.length; i++)
            out.println(names[i]);
      }
   }
   else if (action.equalsIgnoreCase("getImage")) {
      String name = request.getParameter("name");
      ImageValue imageValue = getImage(endPointURL, name);
      if (imageValue == null) {
         System.out.println("imageValue is null");
      }
      else {
         response.setContentType("text/plain");
         StringBuffer buffer = new StringBuffer();
         buffer.append(""+imageValue.getDate()+"\n");
         buffer.append(imageValue.getEncodedImage()+"\n");
         response.setContentLength(buffer.length());
         PrintWriter out = response.getWriter();
         out.println(buffer.toString());
      }
   }
   else {
      // action not recognised
   }
}
```

The doGet method makes use of some helper methods, for accessing the ImageService Web service. The purpose of these methods is to separate the Web service access from the main servlet logic, as they are logically distinct.

```
private String[] getNames(URL endPointURL) {
   String[] names = null;
   try {
```

```
        Service service = new Service();
        Call call = (Call)service.createCall();
        call.setTargetEndpointAddress(endPointURL);
        call.setOperationName(new QName("ImageService", "getNames"));
        names = (String[])call.invoke(new Object[] {});
    }
    catch (AxisFault fault) {
        System.err.println("Generated fault: ");
        System.out.println("  Fault Code   = " + fault.getFaultCode());
        System.out.println("  Fault String = " + fault.getFaultString());
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
    return names;
}

private ImageValue getImage(URL endPointURL, String name) {
    ImageValue imageValue = null;
    try {
      // Set up the SOAP Service Object
      Service service = new Service();
      Call call = (Call)service.createCall();
      call.setTargetEndpointAddress(endPointURL);
      call.setOperationName(new QName("ImageService", "getImage"));
      call.addParameter("name",
        org.apache.axis.encoding.XMLType.XSD_STRING,
        ParameterMode.IN);
      QName qn = new QName("urn:BeanService", "ImageValue");
      call.registerTypeMapping(ImageValue.class, qn,
        new BeanSerializerFactory(ImageValue.class, qn),
        new BeanDeserializerFactory(ImageValue.class, qn));
       call.setReturnType(qn);

        imageValue = (ImageValue)call.invoke(new Object[] { name });
    }
    catch (AxisFault fault) {
        System.err.println("Generated fault: ");
        System.out.println("  Fault Code   = " + fault.getFaultCode());
       System.out.println("  Fault String = " + fault.getFaultString());
    }
    catch (Exception e) {
        System.out.println(e.toString());
    }
    return imageValue;
}
```

To deploy the servlet to run on Tomcat, we will need to set up a new Web application. We will call the Web application "javaonpdas," and so we need to create a new directory under ${tomcat-base}\webapps called javaonpdas. In the javaonpdas directory, we need a WEB-INF directory, which in turn should contain a lib directory for the JAR files for the Web application.

In the directory ${tomcat-base}\webapps\javaonpdas\WEB-INF we need to put a web.xml file that describes the new Web application. The web.xml file describes the servlet class that implements the Web application, as well as the URL pattern to be used to access the servlet. The servlet section describes this. Normally we do not want the servlet accessed with a long-winded URL that includes the fully qualified class name—we can use a shorthand name instead. The servlet-mapping section sets this up.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
  <servlet>
    <servlet-name>ImageServiceProxy</servlet-name>
    <display-name>ImageServiceProxy</display-name>
    <servlet-class>
        com.javaonpdas.proxy.ImageServiceProxy
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>ImageServiceProxy</servlet-name>
    <url-pattern>/servlet/ImageServiceProxy</url-pattern>
  </servlet-mapping>

</web-app>
```

With these mappings in place, we can access the new servlet with the following URL:

```
http://192.168.0.1:8080/javaonpdas/servlet/ImageServiceProxy
```

Because our Web application is an Axis SOAP client, we will copy the Axis client JARs into the ${tomcat-base}\webapps\javaonpdas\WEB-INF\lib directory. The JARs are

```
axis.jar
jaxrpc.jar
axis-ant.jar
commons-discovery.jar
commons-logging.jar
log4j-1.2.4.jar
saaj.jar
wsdl4j.jar
```

The `ImageServiceProxy` class is compiled and deployed as part of the `CompileDesktop` target in the Ant build file. The resultant JAR `javaonpdas-desktop.jar` is also copied to the `${tomcat-base}\webapps\javaonpdas\WEB-INF\lib` directory.

Once the set up of the new Web application is complete, we can test it. Restart Tomcat and use a browser to access the following URL:

```
http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceProxy?service-end-point=http://localhost:8080/axis/↵
servlet/AxisServlet&action=getNames
```

This URL is invoking the new `ImageServiceProxy` servlet, telling it the Web service endpoint to use (`http://localhost:8080/axis/servlet/AxisServlet`), and the action to perform (`action=getNames`).

Because the proxy accepts a URL and writes the result in plain text, we can see the result in the Web page in Figure 8.12.

Similarly, we can use the following URL to retrieve the Base64-encoded image `kookaburra.png`:

```
http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceProxy?service-end-point=http://localhost:8080/axis/↵
servlet/AxisServlet&action=getImage&name=kookaburra.png
```

The first line in Figure 8.13 is the long integer corresponding to the image's last modified date, and the second line in the Base64-encoded string of the image itself.

To create a Palm client that accesses the proxy rather than the Web service directly, we will modify `SOAPClient` and create a new MIDlet called HTTPTextClient. This MIDlet is similar except for the way the `getNames` and `getImage` commands are handled.

The first step is to set up the HTTP connection by opening the input stream.

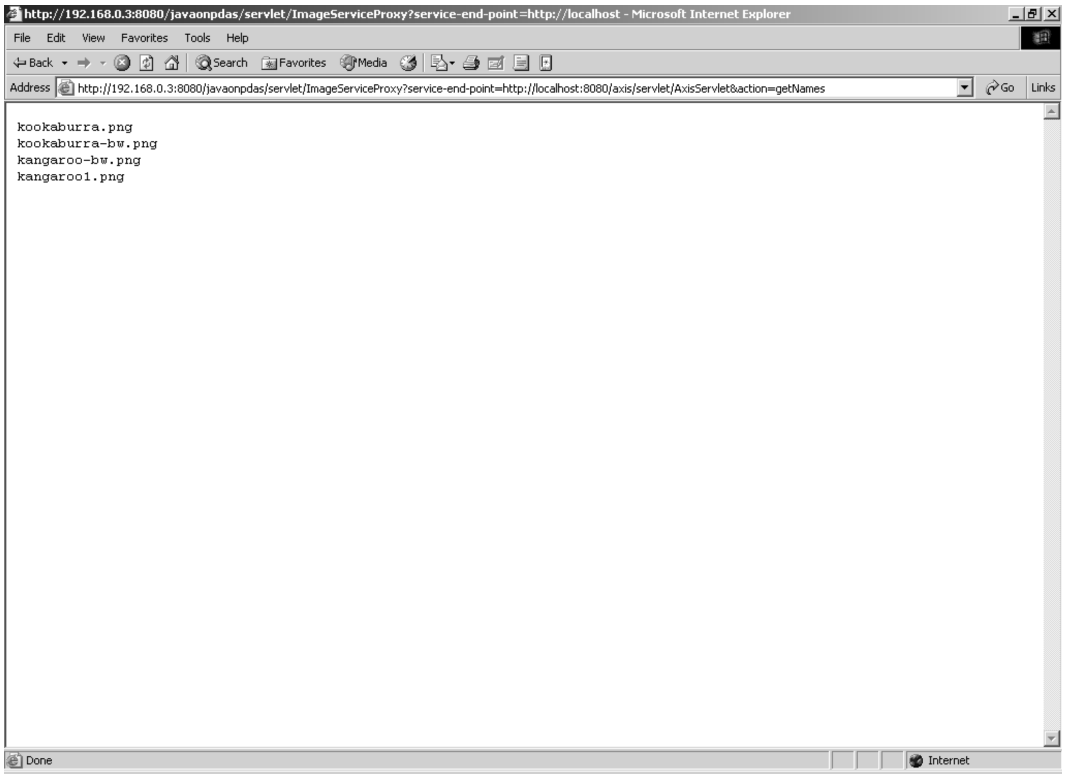

Figure 8.12 Using a browser to return image names from the Web service via the proxy

```
HttpConnection connection = null;
InputStream is = null;
String url = "http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceProxy?service-end-point=http://localhost:8080/axis/↵
servlet/AxisServlet&action=getNames";
try {
   connection = (HttpConnection)Connector.open(url);
   connection.setRequestMethod(HttpConnection.GET);
   is = connection.openInputStream();
   int contentLength = (int)connection.getLength();
```

Assuming the content length is not zero, we create a byte array to accommodate the content, and read from the input stream.

```
byte[] byteArray = new byte[contentLength];
is.read(byteArray);
```

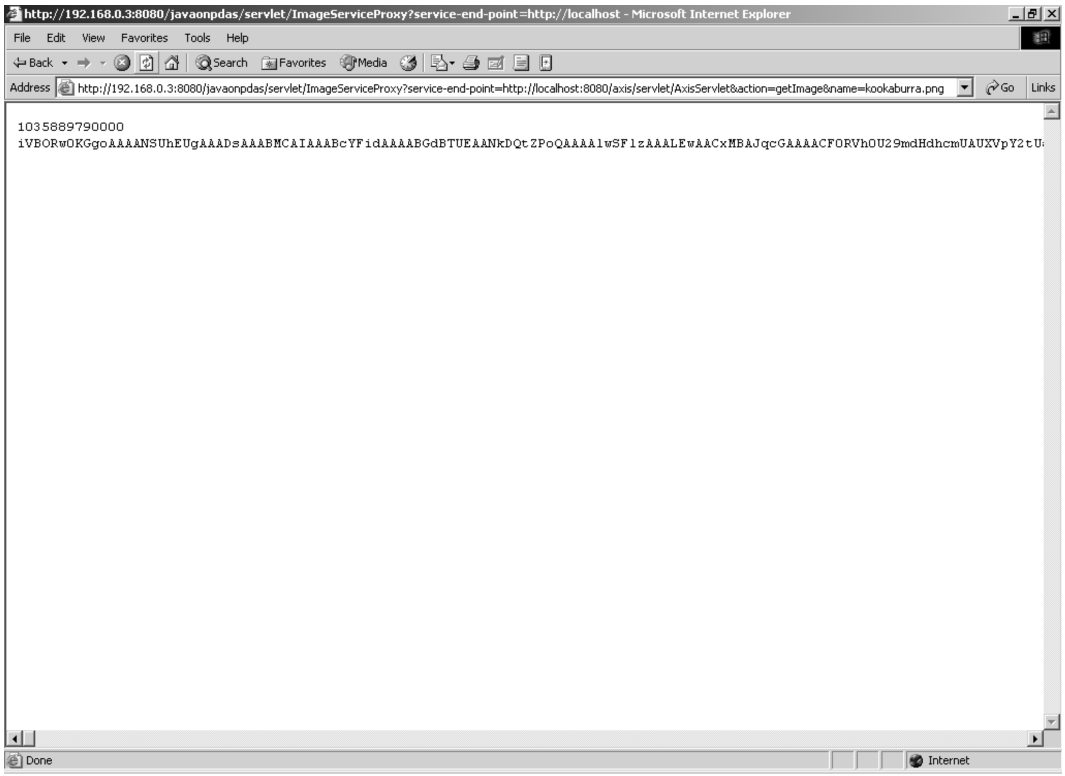

Figure 8.13 Using a browser to retrieve the Base64 representation of an image

Next we parse the byte array looking for '\n' characters, indicating the end of an image file name. For each file name, we add the string to the text field.

```
StringBuffer buffer = new StringBuffer();
textField.setString(null);
for (int i=0; i<byteArray.length; i++) {
   if (byteArray[i] == (byte)'\n') {
      textField.insert(buffer.toString() + "\n", textField.size());
      buffer.setLength(0);
   }
   else if (byteArray[i] == (byte)'\r') {
   }
   else {
      buffer.append((char)byteArray[i]);
   }
}
```

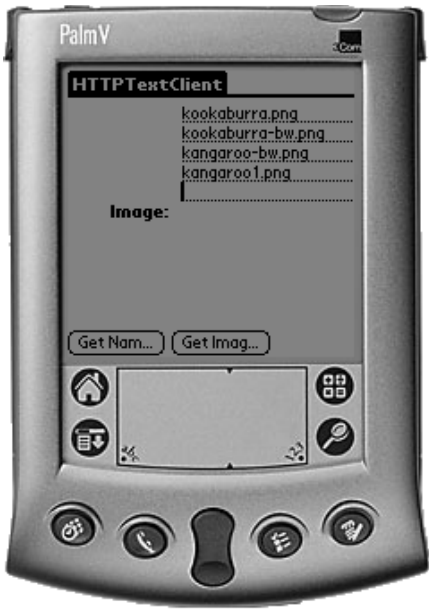

Figure 8.14 Accessing the Web service via the HTTP text proxy

Running `ant HTTPTextClient` compiles the application and starts the emulator. Pressing the "Get Names" button will result in the image file names being displayed as before and as shown in Figure 8.14, except that this time the Palm is communicating with the Web service proxy, rather than the Web service itself.

Using a Proxy with Data Streams to Access the Web Service

An alternative to using HTTP and plain text is to open a stream over the HTTP socket connection between the client and the proxy. A proxy that demonstrates this approach is `ImageServiceStreamProxy`, as shown in Figure 8.15.

The code in `ImageServiceStreamProxy` differs from `ImageServiceProxy` mainly in the way that the information is sent back to the client; the same protocol as that used in `ImageServiceProxy` is used for embedding the request in the URL.

The `doGet` method in `ImageServiceStreamProxy` calls the helper methods to access the Web service as before, but instead of writing the response to a `PrintWriter`, it opens a `DataOutputStream` and writes the response to it.

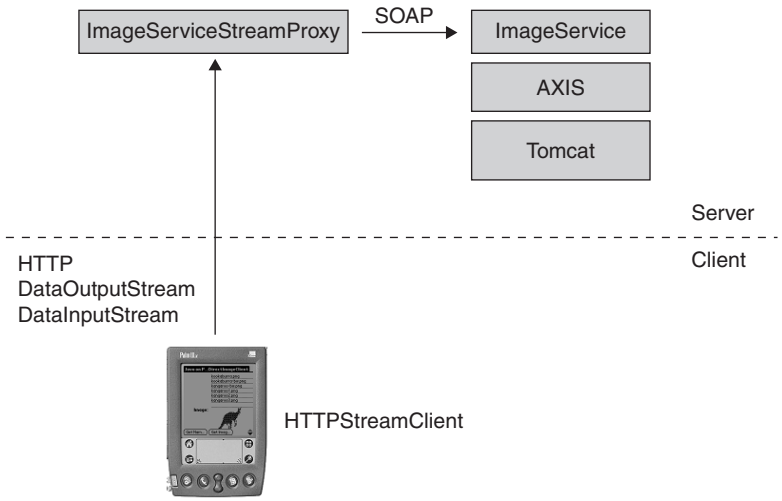

Figure 8.15 Accessing the Web service using a data stream proxy

```
if (action.equalsIgnoreCase("getNames")) {
   String[] names = getNames(endPointURL);
   DataOutputStream dos = null;
   try {
      dos = new DataOutputStream(response.getOutputStream());
      if (names != null) {
         dos.writeInt(names.length);
         for (int i=0; i<names.length; i++)
            dos.writeUTF(names[i]);
      }
      else
         dos.writeInt(0);
   }
   catch (Exception e) {
      System.out.println(e.toString());
   }
   finally {
      try { if (dos != null) dos.close(); } catch (Exception e) {}
   }
}
```

In the "`getNames`" case, the helper method `getNames()` is called to retrieve the file names as before. Then a `DataOutputStream` is opened on the response's output stream. The first thing to write to the `DataOutputStream` is

the number of names in the list, so that the client knows how many to expect. Then we write the file names using `writeUTF()`.

In the "`getImage`" case, we can decode the image from the Base64 string into an array of bytes and write the byte array to the `DataOutputStream`. This saves the client from decoding the string, which is consistent with our objective of doing as much work as possible on the server.

```
else if (action.equalsIgnoreCase("getImage")) {
   String name = request.getParameter("name");
   ImageValue imageValue = getImage(endPointURL, name);
   if (imageValue == null) {
      System.out.println("imageValue is null");
   }
   else {
      byte[] byteArray = org.apache.axis.encoding.Base64.decode(
        imageValue.getEncodedImage());
      System.out.println("image length="+byteArray.length);
      response.setContentLength(byteArray.length);
      DataOutputStream dos = null;
      try {
         dos = new DataOutputStream(response.getOutputStream());
         dos.write(byteArray, 0, byteArray.length);
      }
      catch (Exception e) {
         System.out.println(e.toString());
      }
      finally {
        try { if (dos != null) dos.close(); } catch (Exception e) {}
      }
   }
}
```

To use this new proxy based on streams, we need to modify the client to use streams as well. The client is called `HTTPStreamClient`. The client makes the request in the same way as before but retrieves the result by opening a `DataInputStream` on the HTTP connection.

```
if (c == getNamesCommand) {
   HttpConnection connection = null;
   DataInputStream dis = null;
   String url = "http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceStreamProxy?service-end-point=http://localhost:8080/↵
axis/servlet/AxisServlet&action=getNames";
   try {
      connection = (HttpConnection)Connector.open(url);
      connection.setRequestMethod(HttpConnection.GET);
```

```
            dis = connection.openDataInputStream();
            int numberOfNames = dis.readInt();
            textField.setString(null);
            for (int i=0; i<numberOfNames; i++) {
               String name = dis.readUTF();
               textField.insert(name + "\n", textField.size());
            }
         }
         catch (Exception e) {
            textField.insert("Error:" + e.toString() + "\n", textField.size());
         }
         finally {
            try {
              if (connection != null) connection.close();
              if (dis != null) dis.close();
            } catch (Exception e) {}
         }
      }
```

The first thing to read from the `DataInputStream` is the number of file names sent by the proxy. Then we loop that number of times, reading the strings from the input stream and displaying them on the text field.

In the case of "Get Image," again we set up a `DataInputStream` on the HTTP connection, read the array of bytes from the stream, and create an image from the array. Recall that the Base64 string was decoded on the server by the proxy.

```
else if (c == getImageCommand) {
   HttpConnection connection = null;
   DataInputStream dis = null;
   String url = "http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceStreamProxy?service-end-point=http://localhost:8080/↵
axis/servlet/AxisServlet&action=getImage&name=Kangaroo.png";
   try {
      connection = (HttpConnection)Connector.open(url);
      connection.setRequestMethod(HttpConnection.GET);
      int contentLength = (int)connection.getLength();
      if (contentLength>0) {
         dis = connection.openDataInputStream();
         byte[] imageByteArray = new byte[contentLength];
         int ch = 0;
         for (int i=0; i<contentLength; i++) {
            if ((ch = dis.read()) != -1) {
               imageByteArray[i] = (byte)ch;
            }
            else {
```

```
                textField.insert("Error: encountered EOF\n",
                                 textField.size());
             }
          }
          imageItem.setImage(Image.createImage(imageByteArray,
                             0, imageByteArray.length));
       }
    }
    catch (Throwable t) {
      textField.insert("Error:" + t.toString() + "\n",
                       textField.size());
      t.printStackTrace();
    }
    finally {
       try {
         if (dis != null) dis.close();
         if (connection != null) connection.close();
       } catch (Exception e) {}
    }
}
```

Comparing Performance

The three access methods were tested on a Palm IIIx with a direct serial connection to a PC, and their performance is compared in Table 8.2.

In the next section, we will compare these access methods.

Summary of Web Service Access Options

The following comparison describes the advantages of each option over the other options. The disadvantages are of each option are compared to the other options. The reason for using a particular option is also given.

Table 8.2 Comparing Performance of Web Service Access Methods

	SOAP	HTTPText	HTTPStream
Size of PRC (bytes)	80,800	12,925	9,509
Free memory at runtime (bytes)	61,864	62,576	62,576
Memory used (bytes)	288	2,752	2,444
Time to request and retrieve image (ms)	28,090	9,010	7,420

SOAP

Advantages

- No client-specific proxy is required on the server.

Disadvantages

- SOAP on the client uses precious memory and processing power.
- Slower and uses more memory compared to the proxy methods.

HTTP Text

Advantages

- Simple protocol to exchange information with the client.
- The server handles the overhead of the SOAP connection.

Disadvantages

- Requires a client-specific proxy on the server.

HTTPStream

Advantages

- Simple protocol to exchange information with the client.
- Can decode the Base64 string on the server, thus making less work for the client.
- The server handles the overhead of the SOAP connection.
- Uses a binary connection, meaning there is no need to detect string boundaries.
- The fastest of the three access methods compared, and uses the least memory.

Disadvantages

- Requires a client-specific proxy on the server.

Palm ImageViewer

Because the HTTP stream method uses the least memory and is the fastest end-to-end, we will use this method for the Palm version of `ImageViewer`.

The `ImageViewer` constructor sets up the application display. We will use a `ChoiceGroup` for selecting the image to display, another `ChoiceGroup` for selecting locally or remotely stored images, an `ImageItem` to display the image, a `Command` for saving remote images to local storage, and a `Command` for exiting the application.

Here is the constructor code:

```
public ImageViewer() {
   // populate the list with local names
   String[] names = getNames(true);
   if (names == null)
     nameChoice =
       new ChoiceGroup("Names", ChoiceGroup.EXCLUSIVE, NAMES, null);
   else
     nameChoice =
       new ChoiceGroup("Names", ChoiceGroup.EXCLUSIVE, names, null);
   // add the items to the form
   mainForm.append(nameChoice);
   mainForm.append(locationChoice);
   mainForm.append(imageItem);
   mainForm.addCommand(saveCommand);
   mainForm.addCommand(exitCommand);
   mainForm.setCommandListener(this);
   mainForm.setItemStateListener(this);
}
```

Note that the `ImageViewer` class implements two interfaces: the `CommandListener` interface and the `ItemStateListener` interface.

```
public class ImageViewer extends MIDlet
implements CommandListener, ItemStateListener
```

This is because we need to listen for `Command` selections, and state changes and name selections in the `ChoiceGroups`.

The screen looks like Figure 8.16 on starting the application.

Selecting the Location `ChoiceGroup` causes the `itemStateChanged` method to be called. Making a choice of location results in the `Names` list being populated with image names according to the Location selection. Selecting an image name results in the image being displayed on the screen.

```
public void itemStateChanged(Item item) {
   Runtime.getRuntime().gc();
   try {
      String location = locationChoice.getString(
                        locationChoice.getSelectedIndex());
      System.out.println("location: " + location);
      imageItem.setLabel("Please wait...");
      if (item == locationChoice) {
         String[] names = getNames(location.equals("Local"));
         System.out.println("there are " + names.length + " names");
         int namesInList = nameChoice.size();
```

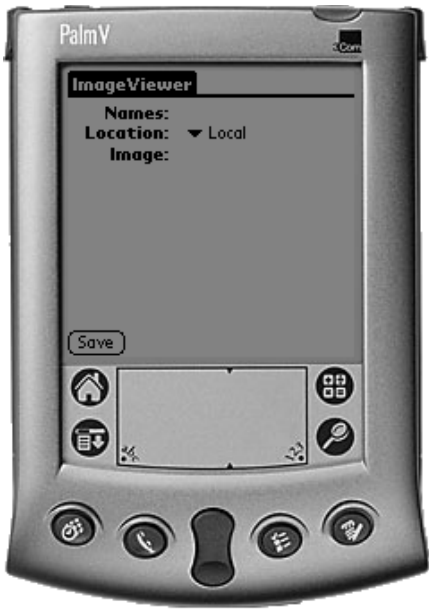

Figure 8.16 The Palm OS version of ImageViewer

```
            for (int i=0; i<namesInList; i++)
               nameChoice.delete(0);
            for (int i=0; i<names.length; i++)
               nameChoice.append(names[i], null);
            imageItem.setImage(Image.createImage(
               Image.createImage(10,10)));
         }
         else if (item == nameChoice) {
            String name =
               nameChoice.getString(nameChoice.getSelectedIndex());
            imageItem.setImage(Image.createImage(
               Image.createImage(10,10)));
            Image image = getImage(name, location.equals("Local"));
            imageItem.setImage(image);
         }
         imageItem.setLabel("Image");
      }
      catch (Exception e) {
         System.out.println("Error: " + e.toString());
      }
   }
```

Images are stored on the client in a `RecordStore`. The image names are retrieved from the `RecordStore` as follows:

```
store = RecordStore.openRecordStore(recordStoreName, true);
RecordEnumeration re = store.enumerateRecords(null, null, false);
names = new String[re.numRecords()];
int i=0;
while(re.hasNextElement()) {
   ByteArrayInputStream bis = new ByteArrayInputStream(re.nextRecord());
   dis = new DataInputStream(bis);
   names[i++] = dis.readUTF();
}
store.closeRecordStore();
```

When a local image name is selected, the image is retrieved from the `RecordStore` as follows:

```
RecordStore store = null;
try {
   store = RecordStore.openRecordStore(recordStoreName, true);
   RecordEnumeration re =
      store.enumerateRecords(null, null, false);
   while(re.hasNextElement()) {
      ByteArrayInputStream bis =
         new ByteArrayInputStream(re.nextRecord());
      dis = new DataInputStream(bis);
      if (dis.readUTF().equals(name)) {
         int imageSize = dis.readInt();
         imageByteArray = new byte[imageSize];
         dis.readFully(imageByteArray);
         image = Image.createImage(imageByteArray,
            0, imageByteArray.length);
      }
   }
   store.closeRecordStore();
}
catch (Exception e) {
   System.out.println("Error:" + e.toString());
}
finally {
   try {
      if (store != null) store.closeRecordStore();
      if (dis != null) dis.close();
   } catch (Exception e) {}
}
```

To retrieve the image names from the server, the following code should look familiar from the previous examples:

```
String url = "http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceStreamProxy?service-end-point=http://localhost:8080/↵
axis/servlet/AxisServlet&action=getNames";
connection = (HttpConnection)Connector.open(url);
connection.setRequestMethod(HttpConnection.GET);
dis = connection.openDataInputStream();
int numberOfNames = dis.readInt();
names = new String[numberOfNames];
for (int i=0; i<numberOfNames; i++) {
   names[i] = dis.readUTF();
}
```

Retrieving the image from the server is the same technique we used in the `HTTPStreamClient`:

```
HttpConnection connection = null;
String url = "http://192.168.0.1:8080/javaonpdas/servlet/↵
ImageServiceStreamProxy?service-end-point=http://localhost:8080/↵
axis/servlet/AxisServlet&action=getImage&name="+name;
try {
   connection = (HttpConnection)Connector.open(url);
   connection.setRequestMethod(HttpConnection.GET);
   int contentLength = (int)connection.getLength();
   if (contentLength>0) {
      dis = connection.openDataInputStream();
      imageByteArray = new byte[contentLength];
      int ch = 0;
      for (int i=0; i<contentLength; i++) {
         if ((ch = dis.read()) != -1) {
            imageByteArray[i] = (byte)ch;
         }
         else {
            System.out.println("Error: encountered EOF");
         }
      }
      image = Image.createImage(imageByteArray, 0,
                                imageByteArray.length);
   }
}
catch (Throwable t) {
   System.out.println("Error:" + t.toString());
   t.printStackTrace();
}
```

```
finally {
   try {
      if (dis != null) dis.close();
      if (connection != null) connection.close();
   } catch (Exception e) {}
}
```

Saving an image is only possible if it is remote. In this case, we take the image array already retrieved to display the image and store it in the record store. If the image name already exists in the record store, the stored image is replaced.

First, we open the record store and create a byte array that stores the record store element:

```
store = RecordStore.openRecordStore(recordStoreName, true);
System.out.println("size in bytes (before): "+store.getSize());
// create the new record byte array
ByteArrayOutputStream bos = new ByteArrayOutputStream();
dos = new DataOutputStream(bos);
String name = nameChoice.getString(nameChoice.getSelectedIndex());
dos.writeUTF(name);
dos.writeInt(imageByteArray.length);
dos.write(imageByteArray, 0, imageByteArray.length);
byte[] ba = bos.toByteArray();
```

Then we look for the name in the record store, and update the stored image if the name is already there.

```
RecordEnumeration re = store.enumerateRecords(null, null, false);
System.out.println("looking for " + name + " in " +
   re.numRecords() + " records");
boolean found = false;
int recordID = 1;
while(re.hasNextElement()) {
   ByteArrayInputStream bis =
      new ByteArrayInputStream(re.nextRecord());
   System.out.println("looking at record "+recordID);
   dis = new DataInputStream(bis);
   if (dis.readUTF().equals(name)) {
      System.out.println(name + " found match - updating");
      store.setRecord(recordID, ba, 0, ba.length);
      found = true;
      break;
   }
   else
      System.out.println("no match");
      recordID++;
}
```

If the image name is not found, the image is added and the record store is closed:

```
if (!found) {
   System.out.print(name + " not found - adding " + ba.length);
   int addedID = store.addRecord(ba, 0, ba.length);
   System.out.println(" ID = " + addedID);
}
System.out.println("size in bytes (after): "+store.getSize());
store.closeRecordStore();
```

The full source code for the Palm version of ImageViewer is located in `C:\JavaOnPDAs\Palm\src\com\javaonpdas\webservices\clients\custom\ImageViewer.java`.

PocketPC Version

Because the PocketPC is far less constrained in terms of memory and processing capability, a direct SOAP connection to the Web service makes more sense than using the proxy that proved best for a Palm OS implementation. The PocketPC application `com.javaonpdas.webservices.clients.wingfoot.ImageViewer` takes this approach.

The constructor sets up the Frame. As before, we have a way to select between local and remote, a selection of image names, a button for saving remote images locally, and a component to display the selected image. In the PocketPC case, we will use a radio button for the local/remote selection, and a list box for the image names.

```
public ImageViewer(String title) {
   super(title);
   // handle frame closing events
   addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent e) {
         System.exit(0);
      }
    } );
   // add components
   setLayout(new GridLayout(2,1));
   Panel topPanel = new Panel();
   CheckboxGroup checkBoxGroup = new CheckboxGroup();
   localCheckbox = new Checkbox("Local", checkBoxGroup, true);
   remoteCheckbox = new Checkbox("Remote", checkBoxGroup, false);
   localCheckbox.addItemListener(this);
   remoteCheckbox.addItemListener(this);
```

```
      topPanel.add(localCheckbox);
      topPanel.add(remoteCheckbox);
      saveButton = new Button("Save");
      saveButton.setEnabled(false);
      topPanel.add(saveButton);
      saveButton.addActionListener(this);
      list = new List(4);
      list.addItemListener(this);
      topPanel.add(list);
      add(topPanel);
      imageCanvas = new ImageCanvas();
      add(imageCanvas);
      String[] names = getNames(true);
      if (names != null)
         for (int i=0; i<names.length; i++)
            list.add(names[i]);
   }
```

Note that the Save button is disabled when the Local radio button is selected, as it is only possible to save remote images.

The component used for displaying the image is a separate class called `ImageCanvas`. `ImageCanvas` is a subclass of `Canvas`, and separates the job of setting the image and repainting away from the `ImageViewer` class.

```
package com.javaonpdas.webservices.clients.wingfoot;

import java.awt.Canvas;
import java.awt.Image;
import java.awt.Graphics;
import java.awt.Dimension;

public class ImageCanvas extends Canvas {
   private Image image = null;

   public ImageCanvas() {
      super();
   }

   public void setImage(Image image) {
      this.image = image;
   }

   public Image getImage() {
      return this.image;
   }
```

```
   public void update(Graphics g) {
      paint(g);
   }

   public void paint(Graphics g) {
      if (image != null) {
         Dimension size = getSize();
         g.clearRect(0, 0, size.width, size.height);
         g.drawImage(image, 0, 0, this);
      }
   }
}
```

There are three events in which we are interested: when the Save button is pressed (triggering an `ActionEvent`), when the user makes a change in selection between Local and Remote, and when an image name is selected (both of which trigger an `ItemEvent`).

The `ItemEvent` is handled by the `itemStateChanged` method:

```
public void itemStateChanged(ItemEvent evt) {
   if (evt.getItemSelectable().getClass() ==
      new Checkbox().getClass()) {
      local = localCheckbox.getState();
      if (local) saveButton.setEnabled(false);
      else saveButton.setEnabled(true);
      String[] names = getNames(local);
      list.removeAll();
      if (names != null)
         for (int i=0; i<names.length; i++)
            list.add(names[i]);
   }
   else {
      String name = list.getSelectedItem();
      Image image = getImage(name, local);
      imageCanvas.setImage(image);
      imageCanvas.update(imageCanvas.getGraphics());
   }
}
```

If the `ItemEvent` corresponds to the Local/Remote radio button, we get the image names either stored locally or remotely, depending on the state of the Local button. If the button is set to Local, the Save button is disabled.

If the `ItemEvent` corresponds to the list of image names, we find the current selection and get the image of that name, and display it on the `ImageCanvas`.

The `ActionEvent` is handled by the `actionPerformed` method:

```
public void actionPerformed(ActionEvent evt) {
   String cmd = evt.getActionCommand();
   if (cmd.startsWith("Save")) {
      Image image = imageCanvas.getImage();
      String name = list.getSelectedItem();
      FileOutputStream ostream = null;
      try {
        ostream = new FileOutputStream(LOCAL_DIRECTORY + "\\" + name);
        ostream.write(imageByteArray);
        ostream.close();
      }
      catch (Exception e) {
         System.out.println("Error:" + e);
      }
      finally {
         try {
           if (ostream != null) ostream.close();
         } catch (Exception e) {}
      }
   }
}
```

If the Save button is pressed, the current image byte array is written to a file in the local image directory using `FileOutputStream`.

The names are retrieved by the `getNames` method, which takes a `Boolean` parameter to indicate whether to retrieve the names from the local or remote repository. In the local case:

```
FilenameFilter filter = new FilenameFilter() {
   public boolean accept(File dir, String name) {
      return name.endsWith(".jpeg");
   }
};
File dir = new File(LOCAL_DIRECTORY);
files = dir.list(filter);
```

Note that we are only interested in JPEG images, where in the Palm ImageViewer we used PNG image files. This is because PersonalJava 1.2 is only required to support a minimum set of image types (GIF, XBM, and JPEG). J2ME MIDP, on the other hand, is required at a minimum to support PNG images.

In the remote case, we use the J2SE version of the Wingfoot SOAP client. This version of the SOAP client has the same APIs, except for the transport used. Recall that in the Palm `SOAPClient` we used `HTTPTransport`, which underneath uses MIDP's GCF. PersonalJava is J2SE, and so it does not have

GCF. Wingfoot gets around this difference by providing a different transport class: `J2SEHTTPTransport`.

```
try {
   // Prepare the Envelope
   Envelope requestEnvelope = new Envelope();
   requestEnvelope.setBody("extension", ".jpeg");

   // Prepare the call
   Call call = new Call(requestEnvelope);
   call.setMethodName("getNames");
   call.setTargetObjectURI("ImageService");

   // Prepare the transport
   J2SEHTTPTransport transport =
      new J2SEHTTPTransport(SOAP_ENDPOINT, null);
   transport.getResponse(true);

   // Make the call
   Envelope responseEnvelope = call.invoke(transport);

   // Parse the response
   if (responseEnvelope != null) {
      if (responseEnvelope.isFaultGenerated()) {
         Fault f = responseEnvelope.getFault();
         System.out.println("Error: " + f.getFaultString());
      }
      else {
         Object[] parameter =
           (Object[])responseEnvelope.getParameter(0);
         String[] temp = new String[parameter.length];
         for (int i=0; i<parameter.length; i++)
            temp[i] = (String)parameter[i];
         files = temp;
      }
   }
}
catch (java.net.ConnectException e) {
   (new ImageDialog(this, "Error", "Could not connect to " +
        SOAP_ENDPOINT)).show();
}
catch (Exception e) {
   (new ImageDialog(this, "Error", e.toString())).show();
}
```

Note that the code is very similar to the Palm `SOAPClient`, except for the different transport class.

To retrieve a local image, we load a byte array from a file with the image name, and create an image:

```
FileInputStream fis = null;
try {
   // read the file into a byte array
   File imageFile = new File(LOCAL_DIRECTORY + "\\" + name);
   if (imageFile.exists()) {
      fis = new FileInputStream(imageFile);
      int length = fis.available();
      imageByteArray = new byte[length];
      fis.read(imageByteArray);
      imageCanvasDimension = imageCanvas.getSize();
      image = Toolkit.getDefaultToolkit().createImage(↵
        imageByteArray).getScaledInstance(↵
        -1, imageCanvasDimension.height, Image.SCALE_FAST);
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(image,0);
      try {tracker.waitForID(0);} catch (InterruptedException e){};
   }
}
catch (Exception e) {
   (new ImageDialog(this, "Error", e.toString())).show();
}
finally {
   try { if (fis != null) fis.close(); } catch (Exception e) {}
}
```

Note the use of `MediaTracker`. Because `createImage` can spawn a separate thread to do its work, as it can take a long time to do some image tasks, we want to make sure that the task is finished before we go to the next step. This is done by using a `MediaTracker` that waits for a task with a given identifier (in this case, 0) to complete.

Again the code that retrieves an image from the `ImageService` Web service is quite familiar, except for the use of `J2SEHTTPTransport`:

```
try {
   // Prepare the Envelope
   Envelope requestEnvelope = new Envelope();
   requestEnvelope.setBody("name", name);

   // Prepare the call
   Call call = new Call(requestEnvelope);
   call.setMethodName("getImage");
```

```
        call.setTargetObjectURI("ImageService");
        TypeMappingRegistry registry = new TypeMappingRegistry();
        registry.mapTypes("urn:BeanService", "ImageValue",
           new ImageValue().getClass(),
           new BeanSerializer().getClass(),
           new BeanSerializer().getClass());
        call.setMappingRegistry(registry);

        // Prepare the transport
        J2SEHTTPTransport transport =
           new J2SEHTTPTransport(SOAP_ENDPOINT, null);
        transport.getResponse(true);

        // Make the call
        Envelope responseEnvelope = call.invoke(transport);

        // Parse the response
        if (responseEnvelope != null) {
           if (responseEnvelope.isFaultGenerated()) {
              Fault f = responseEnvelope.getFault();
              System.out.println("Error: " + f.getFaultString());
           }
           else {
              ImageValue imageValue =
                 (ImageValue)responseEnvelope.getParameter(0);
              Date date =
                 new Date(imageValue.getDateAsLong().longValue());
              Base64 encodedImage =
                 new Base64(imageValue.getEncodedImage());
              imageByteArray = encodedImage.getBytes();
              imageCanvasDimension = imageCanvas.getSize();
              image =
                 Toolkit.getDefaultToolkit().createImage(
                 imageByteArray).getScaledInstance(-1,
                 imageCanvasDimension.height, Image.SCALE_FAST);
              MediaTracker tracker = new MediaTracker(this);
              tracker.addImage(image,0);
              try {tracker.waitForID(0);}
              catch (InterruptedException e){};
           }
        }
     }
     catch (Exception e) {
        (new ImageDialog(this, "Error", e.toString())).show();
     }
```

We can test the `ImageViewer` application on the desktop prior to deploying it to a PocketPC device. The Ant build target `TestImageViewer` performs this task:

```
  <target name="TestImageViewer" depends="CompilePocketPC">
    <java
       classname="com.javaonpdas.webservices.clients.wingfoot.↵
ImageViewer"
       dir="."
       fork="true"
       failonerror="true">
        <classpath>
          <pathelement location="${pocketpclib}\$↵
{pocketpcsoaplib}"/>
          <pathelement location="${pocketpcdestination}↵
\pocketpc.jar"/>
        </classpath>
     <arg line=""/>
    </java>
  </target>
```

To deploy the application to the PocketPC, we use the Ant target `DeployPocketPC`, which copies the `pocketpc.jar` to the PC's PocketPC synchronization folder.

```
  <target name="DeployPocketPC" depends="CompilePocketPC">
    <copy file="${pocketpcdestination}\pocketpc.jar"
      todir="${pocketpcdeploy}" />
    <copy todir="${pocketpcdeploy}">
      <fileset dir="${pocketpclib}"/>
    </copy>
  </target>
```

On starting the application, we see the frame as shown in Figure 8.17.

Selecting an image name retrieves the image from local storage and displays it on the screen, as shown in Figure 8.18.

Selecting the Remote button and selecting an image name retrieves that image from the `ImageService` Web service, and displays it on the screen, as shown in Figure 8.19.

Figure 8.17 The PocketPC version of ImageViewer, showing image names

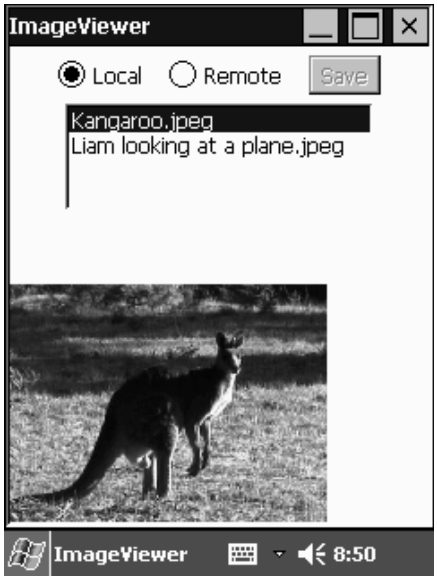

Figure 8.18 The PocketPC version of ImageViewer, showing an image from local storage

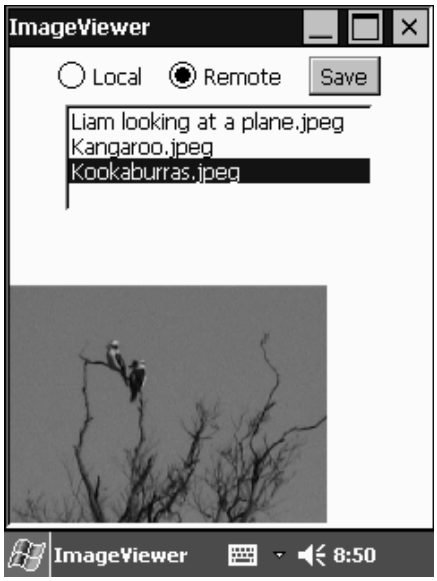

Figure 8.19 The PocketPC version of ImageViewer, showing an image from remote storage

Summary

In this chapter we have explored using Web services to integrate a PDA application with the enterprise. For PDAs with more limited memory and processing capacity, we have seen that Web services are possible, but a preferable technique is to use a proxy on the server. The proxy does as much of the work as possible on the PDA's behalf, and exchanges information with the PDA using a very simple protocol.

For more powerful PDAs, such as the PocketPC, we can use SOAP directly to access server-side functionality.