Starting Java Applications from the Web

James W. Cooper

There's Dave Barry's old joke about the sugary cereal in the picture on TV being "part of a balanced breakfast." He claims it really means *adjacent to* a balanced breakfast: the other tempting foods in the picture, and is about as relevant as a can of shaving cream or a dead bat. This is the way I used to feel about Java Web Start technology. It just didn't seem very important or relevant: it was just this other weird thing on the table.

But now, of course, I see it as a little more important. Java Web Start allows you to post complete Java applications and their data on a web site and run them seamlessly either from the web or in stand-alone fashion. If your client computer is connected to the web, the web start system will automatically check for updates, and if not it will run independently.

This sounded a little daft to me at one time, but of course there are lots of cases where a Java application is more useful than embedding something messy in a web site applet. In fact, running Java applications with automatic updates is really the best of both worlds. I finally came to my senses when I had a fairly complicated application involving data handling, graphics and a large XML data set that I needed to be able to demonstrate to interested users at various locations not nearby my own office. I didn't want or need to travel somewhere just to run a simple demo, but I wanted to make sure that they had the same visual experience on their computer system that I did. In other words, I wanted to make sure that the code worked there too, as well as it did on my laptop or desktop machines.

Java Web Start will allow you to package up a system of any number of jar files and data files and start the system when they are all downloaded. Moreover, if the data you want to include is somewhat voluminous, like XML data often is, you can created a compressed jar file, which will make the download much faster, but hardly impact the performance at all.

Getting a Head Start on Web Start

There are only six major steps to writing and creating a Java Web Start application.

1. Set the web server to recognize the web start MIME type.
2. Write the application as usual, and put the classes in a jar file.
3. Sign the jar file.
4. Create a package file listing all the jars in the application, as a JNLP file.
5. Put the JNLP file and the jar files together on the web site where they can be accessed.
6. Create a web page to launch the application.

We'll go through each of these steps for a really simple web start application.

Setting the Web Server

Each Java Web Start program consists of a set of jar files and a file with a JNLP extension which contains a description of these file. Your web server must be able to recognize that a JNLP file has the type

```
application/x-java-jnlp-file
```

In the case of the Apache web server, you must add the following line to the .mime.types configuration file:

```
application/x-java-jnlp-file JNLP
```

For other web servers, you need to find where to add this type. If your server is hosted by a web hosting company they may provide a service to add types graphically or by E-mail request.

Writing the Application

You can write any sort of application you like, as long as all the resources can be put in jar files. We are going to write a simple program to parse this input XML data file:

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

<document>

    <person>
           <name>Sam</name>
           <surname>Spade</surname>
    </person>
    <person>
           <name>Sally</name>
           <surname>Frazzle</surname>
    </person>
    <person>
           <name>Andre</name>
           <surname>Norton</surname>
    </person>

</document>
```

and display these names in a listbox, as shown in Figure 1.

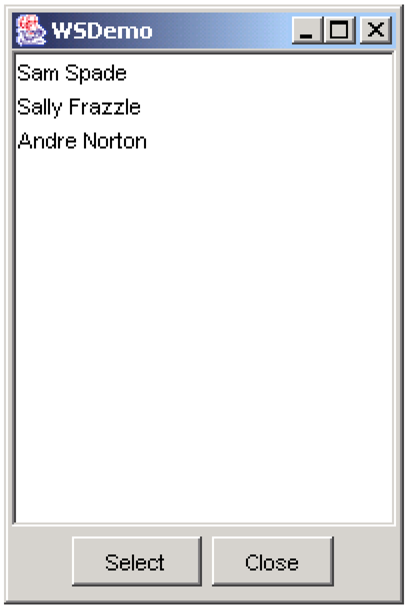


Figure 1 – A display program for the list of names in our XML file.

Now the SAX parser for this XML file is very easy to write, since we only have to look at the startElement, endElement and characters methods. We write a small DocParser class that extends the SAX DefaultHandler class and override these three methods. The characters() method just accumulates characters from the input line into a buffer, which we reset every time we start a new element. This means we are just saving the characters between tags.

```
      //all characters between tags accumulate here
      //may be called any number of times between tags.
      public void characters(char[] ch,
            int start, int length)
                        throws SAXException {
            char cbuf[] = new char[length];
            int k = start;
            for(int i=0; i< length; i++) {
                  cbuf[i]=ch[k++];
            }
            buffer.append(cbuf);
      }
```

The startElement tag jus resets the string buffer:

```
      //a tag has been started
      public void startElement(
            String uri,
            String localName,
            String qName,
            Attributes attrib)
                  throws SAXException {
            buffer = new StringBuffer(); //create a new buffer
      }
```

and the endElement method stores first and last names:

```
//a tag has ended
      //a tag has ended
```

```
public void endElement(String uri, String localName,
      String qName)
      throws SAXException {

      if (qName.equals("name")) {
            String buf = buffer.toString();
            person = new Person(buf);
            buffer = new StringBuffer();
      }
      if (qName.equals("surname")) {
            title = buffer.toString().trim();
            person.setSurname (title);
            docs.addElement( person);
            buffer = new StringBuffer();
      }
}
```

and then stores that Person object in a vector until all the names have accumulated.

The real trick to this program is where to find the XML file that it will reading. Recall that all the resources for a web start program must be inside jar files. We do this in the main program's constructor, creating an input stream by opening that file from the jar file.

```
      //opens an input stream from a datafile
      //enclosed in the jar file
      public InputStream getJarData() {
            try {
                  ClassLoader cl = this.getClass().getClassLoader();
                  URL url = cl.getResource(fileName);
                  InputStream f = url.openStream();
                  return f;
            } catch (IOException e) {
                  return null;
            }
      }
```

Then we start the SAX parser on this input stream as follows:

```
public void parseDocuments(InputStream stream) {
      docParser = new DocParser();
      try {
            SAXParserFactory sFact = SAXParserFactory.newInstance();
            parser = sFact.newSAXParser();
            parser.parse(stream, docParser);
      } catch (SAXException e) {
            System.out.println(
                  "SAX error:" + e.getMessage() + e.getStackTrace());
      } catch (ParserConfigurationException e) {
            System.out.println("Config error:" + e.getMessage());
      } catch (IOException e) {
            System.out.println("IO error:" + e.getMessage());
      }

}
```

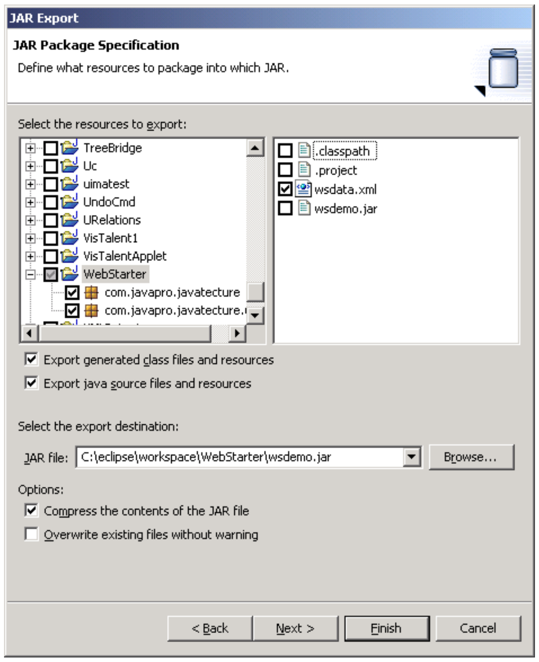
Finally, when the data are all parsed, we can fetch the vector of Person objects:

```
Vector v = docParser.getDocs() ;
Iterator iter = v.iterator() ;
while(iter.hasNext() ) {
      Person doc = (Person)iter.next() ;
      jList.add(doc.getNames() );
}
```

and load them into the list box shown in Figure 1.

Making the Jar File

If you are using Eclipse, or any other development environment, you can make a jar file that include this XML file in about 3 clicks. Just right-click on the project and select Export, and then select "Jar file" and then select the elements to export as shown in Figure 2. One really nice advantage of using data embedded in jar files here is that you can compress them, and since XML files tend to be quite huge, this compression can be very significant, and save significantly on download time without noticeably affecting the propgrams performance in loading the data. In the example screen in Figure 2, we have selected the Compress option for this reason. Jar files, of course, are compressed using much the same algorithm as zip files.



JAR Export

JAR Package Specification

Define what resources to package into which JAR.

Select the resources to export:

- [ ] TreeBridge
- [ ] Uc
- [ ] uimatest
- [ ] UndoCmd
- [ ] URelations
- [ ] VisTalent1
- [ ] VisTalentApplet
- [■] WebStarter
  - [✓] com.javapro.javatecture
  - [✓] com.javapro.javatecture.

- [ ] .classpath
- [ ] .project
- [✓] wsdata.xml
- [ ] wsdemo.jar

[✓] Export generated class files and resources

[✓] Export java source files and resources

Select the export destination:

JAR file: C:\eclipse\workspace\WebStarter\wsdemo.jar

Options:

[✓] Compress the contents of the JAR file

[ ] Overwrite existing files without warning

Figure 2 – Creating a jar file of this Java Web Start project. Note that we have included the XML file and that we have selected Compress file.

Signing the Jar File

If you are distributing this Web Start program "in the wild," you probably need to sign the file using a service provided by a commercial provider. However, you can create a test certificate to make sure everything works using the *keytool* and *jarsigner* programs that are provided with Java 1.4.2.

First you create a key in the internal registry maintained by keytool. From the command line type

```
keytool -genkey -keystore testkeys -alias mykeys
Enter keystore password:  abcde
Keystore password is too short - must be at least 6 characters
Enter keystore password:  abcdef
What is your first and last name?
  [Unknown]:  James
What is the name of your organizational unit?
  [Unknown]:  Cooper
What is the name of your organization?
  [Unknown]:  JavaPro
What is the name of your City or Locality?
  [Unknown]:  New York
What is the name of your State or Province?
  [Unknown]:  NY
What is the two-letter country code for this unit?
  [Unknown]:  NY
Is CN=James, OU=Cooper, O=JavaPro, L=New York, ST=NY, C=NY correct?
  [no]:  yes

Enter key password for <mykeys>
        (RETURN if same as keystore password):
```

Then to create a self-sign test certificate, type

```
C:\>keytool -selfcert -alias mykeys -keystore testkeys
Enter keystore password:  abcdef
```

Now, finally, to sign the jar file, go to folder where the jar file is located and type

```
jarsigner -keystore testkeys wsdemo.jar mykeys
```

and enter the password you created above.

Your file is now signed and ready for deployment.

Creating the JNLP Descriptor File

Most of the errors in deploying Java Web Start applications can be traced to the syntax of the JNLP file. A correct version of the JNLP file for this project is shown below.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- JNLP File for WebStart Demo Application -->
<jnlp  spec="1.0+"
      codebase="http://mycompany.com/JavaPro/Webstart/"
      href="wsdemo.jnlp" >

  <information>
    <title>Web Start Demo</title>
    <vendor>JavaPro</vendor>
    <homepage href="docs/help.html"/>
    <description>Simple demonstration of Java Web Start</description>
    <description kind="short">WSDemo</description>
    <description kind="tooltip">WSDemo</description>
    <offline-allowed/>
 </information>


  <security>
    <j2ee-application-client-permissions/>
  </security>

  <resources>
    <j2se version="1.4+"
      initial-heap-size="32m"
      max-heap-size="256m" />

    <jar href="wsdemo.jar" main="true" />
    <!--jar href="other_stuff.jar"-->
  </resources>

  <application-desc main-class="com.javapro.javatecture.WSDemo" />
</jnlp>
```

As you can see the JNLP file is just an XML file describing the program to be launched. The <jnlp tag must have a correct path in it, exactly as shown, with the URL path specified in the *codebase* and the name of the jnlp file itself in the *href*. The information section is pretty much free form and can contain any descriptions you wish to include. The title and vendor text will appear in the splash screen when you launch the program.

Normally, Web Start applications are run in the “sandbox” much like that applets use, but you can request unrestricted access using the security tag section exactly as shown.

The resources section needs to include a list of all the jar file included in the program, and one and only one of them must have main=”true” indicating that this is the class that launches the program. As shown in the comments, you can include as many other jar files as you like. These jar files can even contain native libraries (such as Windows DLLs) as long as they are specifically loaded by the Java classes.

Note that you can specify that additional memory is to be used, but be sure that if you do, you include both the initial and max heap size arguments. You can also leave both out for simple programs.

Finally, the exact class name which contains the main method is specified in the application-desc tag. Be sure that you include the complete package name and not just the class name.

For a fuller description of all the possible JNLP tags, see the Sun site http://java.sun.com/products/javawebstart/.

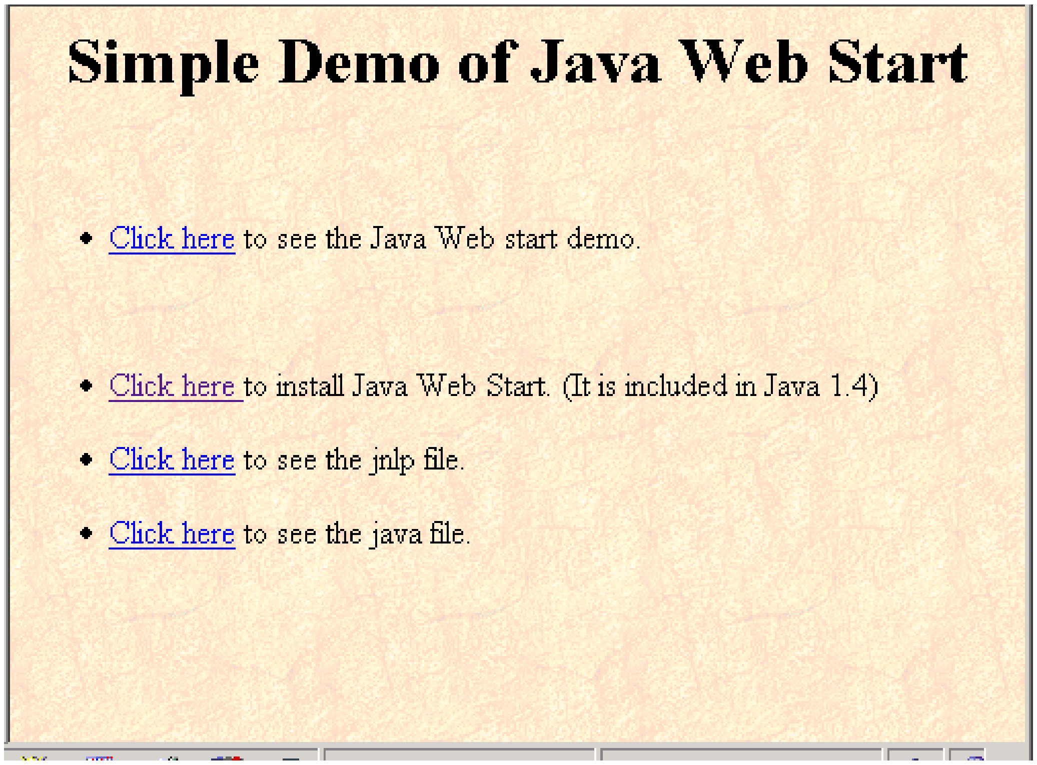
Making the Page that Launches the Java Web Start Program

To launch a Java Web Start program, just create a page with a link to the jnlp file, which includes all the jar files in that same direction. The link from the web page is simply

```
<li>
```

```
  <a href="wsdemo.jnlp">Click here</a>
     to see the Java Web start demo.</li>
```

The complete web page is shown in Figure 3

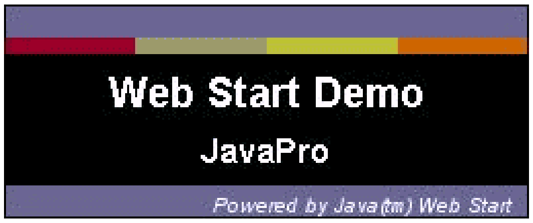


Simple Demo of Java Web Start

- Click here to see the Java Web start demo.
- Click here to install Java Web Start. (It is included in Java 1.4)
- Click here to see the jnlp file.
- Click here to see the java file.

Figure 3 – The Java Web Start web page.

When you click on the top link, you get the splash screen shown in Figure 4.



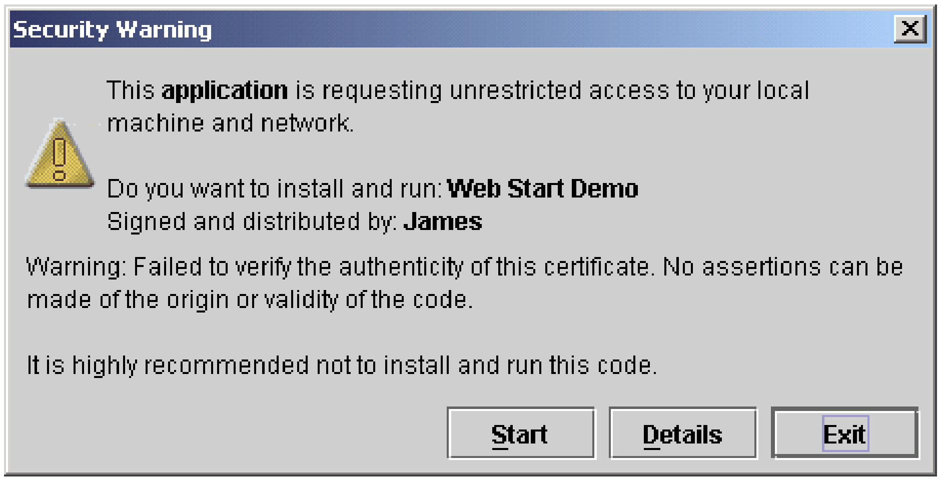
Web Start Demo

JavaPro

Powered by Java(tm) Web Start

Figure 4 – The Java Web Start splash screen.

And then the first time only, you will get a security warning: Figure 5


Security Warning

This **application** is requesting unrestricted access to your local machine and network.

Do you want to install and run: **Web Start Demo**
Signed and distributed by: **James**

Warning: Failed to verify the authenticity of this certificate. No assertions can be made of the origin or validity of the code.

It is highly recommended not to install and run this code.

Figure 5- The security warning you get the first time you launch a new Web Start program.

Then the program launches just as shown in Figure 1. After you run it twice, you will get a popup asking if you want to add an icon to your desktop. Each time you launch the program, it checks for updates if your computer is currently connected to the internet, and downloads them. This makes the system just as flexible as an applet, but more powerful, because these are true applications.

Summary

We've gone through all the steps we had to go through to get a web start application going. Once you get one going, the rest is smooth sailing, and they can be very useful indeed. Imagine that you are developing an application that someone remote to you will be using. This gives you a simple way to post updated jar files each time you make an update and know that they will run correctly the next time that user starts the program. This is pretty compelling compared to Froot Loops!.