May 2008

Java Web Start

Paul Grenyer

Introduction

Java. I spent years avoiding it. I felt it was inferior to the power of C++. I thought it was slow, clunky, the GUI was rubbish and that garbage collection was for wimps who did not know how to clean up after themselves or use smart pointers. Ok, so we all know I was wrong. And life being the way it is, being so out spoken about Java was sure to come and bite me and it did.

Since December I have been writing Java as part of my day job and I found I liked it so much I've even started using it for some of my own projects. I do not even miss Microsoft's visual studio. I have become very attached to Eclipse [Eclipse] and having code checked in real time, therefore negating a build stage, is very useful.

I have been so busy writing Java (and editing my new CVu column Desert Island Books) that I have not written an article on anything else for quite some time. The editor of Overload has been nagging me for material, as has the new publications officer. I have also seen a few comments here and there about how poorly Java is served by the ACCU at present, but then with a strong history in C and C++ this is to be expected. However, my plan here is to redress the balance a little.

I'm spending most of my free time (not that I have a lot these days) working on a file viewer application that allows fixed length record files in excess of 4GB to be viewed without loading the entire file into memory. I wrote one of these in C++ (MFC) for a company I worked for a number of years ago. It worked well, but was a bit clunky and the user interface looked rubbish. I think they are still using it, but I'm not sure. I have had a few failed attempts to write it in C# recently, but it was not until I had a go in Java with its `JTable` and `TableModel` classes that I really made some progress.

The file viewer is a little way off being finished, but I am starting to think about package and deployment options. I want it to be easy and one of the application I use in my day job uses Java Web Start [JWS] and it works really nicely. Sun describe Java Web Start as:

> Using Java Web Start technology, standalone Java software applications can be deployed with a single click over the network. Java Web Start ensures the most current version of the application will be deployed, as well as the correct version of the Java Runtime Environment (JRE).

It sounds ideal for a constantly developing application that may be used by people all over the world on different operating systems.

As I sit down to write this article I have done no more that briefly read the Java Web Start documentation (so much for writing about what I know about – again!). I am intending to write an article about how to create applications and deploy them using web start by investigating it myself and writing down the steps as I go. I'll assume a reasonable familiarity with Java and Swing [JavaSwing].

From reading the documentation it looks like I need to package my Java application in a JAR file so I'll look at how to do that and make the process easily repeatable using ANT.

Java Web Start Application

In order to test Java Web Start I need a simple Java application. Before getting stuck into writing such an application it is worth consulting the Java Web Start Guide [JWSG] section on Application Development Considerations, which states:

> Developing applications for deployment with Java Web Start is generally the same as developing stand-alone applications for the Java(TM) Platform Standard Edition. For instance, the entry point for the application is the standard:
>
> ```
> public static void main(String[] argv)
> ```
>
> However, in order to support Web deployment - automatic download and launching of an application - and to ensure that an application can run in a secure sandbox, there are some additional considerations:
>
> - An application must be delivered as a set of JAR files.
> - All application resources, such as files and images must be stored in JAR files; and they must be referenced using the `getResource` mechanism in the Java(TM) Platform Standard Edition.
> - If an application is written to run in a secure sandbox, it must follow these restrictions:
>   - No access to local disk.
>   - All JAR files must be downloaded from the same host.
>   - Network connections are enabled only to the host from which the JAR files are downloaded.
>   - No security manager can be installed.
>   - No native libraries may be used.
>   - Limited access to system properties. The application has read/write access to all system properties defined in the JNLP File, as well as read-only access to the same set of properties that an Applet has access to.
> - An application is allowed to use the `System.exit` call.
> - An application that needs unrestricted access to the system will need to be delivered in a set of signed JAR files. All entries in each JAR file must be signed.

As expected, the application needs to be JARed. My file viewer application uses both resources and requires access to the local system so the JARs will have to contain resources and be signed, but I want to start with something simpler first. All of this is covered in the Application Development Considerations.

I will start with a simple Java Swing application:

```java
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloJavaWebStart
{
	public static void main(String[] args)
	{
		JFrame frame = new JFrame("Java Web Start!");
		JLabel label =
			new JLabel("Hello, Java Web Start!", JLabel.CENTER);
		frame.add(label);
		frame.setSize(250,100);
		frame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
		frame.setVisible(true);
	}
}
```

The above application creates a simple window with the title “Java Web Start!” and a label “Hello, Java Web Start”. The window is 250 by 100 pixels and visible. The easiest way to test the application is at the command line (unless of course you are using Eclipse):

```
javac HelloJavaWebStart.java

java HelloJavaWebStart
```

The window looks like this:

JARing

The Sun website describes JAR files as follows:

> The Java™ Archive (JAR) file format enables you to bundle multiple files into a single archive file. Typically a JAR file contains the class files and auxiliary resources associated with applets and applications.

The format of the command line parameters for the jar command is:

```
jar cf jar-file input-file(s)
```

The options and arguments used in this command are:

- The `c` option indicates that you want to create a JAR file.
- The `f` option indicates that you want the output to go to a file rather than to `stdout`.
- `jar-file` is the name that you want the resulting JAR file to have. You can use any filename for a JAR file. By convention, JAR filenames are given a .jar extension, though this is not required.

- The `input-file(s)` argument is a space-separated list of one or more files that you want to include in your JAR file. The `input-file(s)` argument can contain the wildcard `*` symbol. If any of the "`input-files`" are directories, the contents of those directories are added to the JAR archive recursively.

Putting the application into a JAR is therefore straight forward:

```
jar cf HelloJavaWebStart.jar HelloJavaWebStart.class
```

However, to be able to just run the application from the JAR file, I need a manifest file to tell Java which `main` should be run by default. Obviously, the `HelloJavaWebStart` application only has a single `main`, but any number of classes can be included in a JAR and all of them could have their own `main` methods.

A manifest file is just a text file that, in this case, indicates which `main` to run:

```
Main-Class: MyPackage.MyClass
```

The text file must end with a new line or carriage return, otherwise the last line will not be parsed properly. Also, there must only be a single space between the colon and the start of the package or class name.

The `HelloJavaWebStart` application uses the default package, so only the name of the class is required:

```
Main-Class: HelloJavaWebStart
```

To incorporate the manifest file into the JAR add the `m` command line parameter and the name of the manifest file to the invocation of the JAR tool:

```
jar cfm HelloJavaWebStart.jar Manifest.txt HelloJavaWebStart.class
```

On Windows and Linux the application can now be run by simply double clicking the JAR file. The alternative is to run it from the command line:

```
java -jar HelloJavaWebStart.jar
```

JARing with ANT

Having to repeatedly type the `javac` command followed by the `jar` command is both time consuming and error prone (not to mention irritating). Ant is a build tool that can automate both. I wont go into the details of installing Ant, but the basic steps are:

1. Download and unpack Ant.
2. Make sure the Ant bin directory is in your platforms `PATH` environment variable.
3. Add `JAVA_HOME` to your platform's environment variables and make sure it points to the location of your Java SDK installation (e.g. on Windows: `C:\Program Files\Java\jdk1.6.0_06`).

Ant uses XML to describe builds. Every Ant build file must contain a project:

```
<project name = "HelloJavaWebStart" basedir=".">
      ...
</project>
```

As shown above, every Ant project should specify its name and the base directory. The name is specified by the `name` attribute. The base directory is the directory where Ant will go looking for the files to be processed and is specified by the `basedir` attribute. Specifying a full stop tells Ant to look in the current directory. When an Ant build file is run, the tasks inside the project are executed.

The `javac` task is used to compile java files:

```
<javac srcdir = "${basedir}"  />
```

In its simplest form the `javac` task only needs to know where to look for the `.java` files to compile. This is specified by the `srcdir` attribute. When `${basedir}` is read by Ant it is replaced by the value of `basdir` specified by the project. The `javac` task will look in the current directory for `.java` files and, by default, write the `.class` files to the same directory. In most cases you would want to specify a separate source and destination directory. Ant does of course allow this, but it is not necessary for this example.

The `jar` task is used to create JAR files:

```
<jar  jarfile = "HelloJavaWebStart.jar"
      basedir="${basedir}"
      manifest="manifest.txt"
      includes="*.class"/>
```

The `jar` task needs to know:

- The name of the JAR file to create, specified by the `jarfile` attribute.
- The directory to look in to find the files to jar, specified by the `basedir` attribute.
- The location of the manifest file, specified by the `manifest` attribute.
- The types of files to include in the jar, specified by the `include` attribute. In the example above, specifying `*.class` tells the jar task to include all `.class` files and nothing else.

By default Ant build files are called `build.xml`. The complete Ant file for the `HelloJavaWebStart` looks like this:

```
<project name = "HelloJavaWebStart" basedir=".">
      <javac srcdir = "${basedir}"  />
      <jar  jarfile = "HelloJavaWebStart.jar"
            basedir="${basedir}"
            manifest="manifest.txt"
            includes="*.class"/>
</project>
```

and should be saved to the same directory as `HelloJavaWebStart.java`. To invoke Ant and build the application type the following at the command line:

```
ant
```

This will give something resembling the following output:

```
Buildfile: build.xml
    [javac] Compiling 1 source file
      [jar] Building jar: HelloJavaWebStart.jar

BUILD SUCCESSFUL
```

Ant is a very powerful build tool and does far more than I have described here. See the Ant documentation for details.

Configuring a Web Server

Java Web Start applications can be hosted on almost any web server, but the server must be configured to support the JWS mime type.

Apache

Apache [Apache] web server could not be easier to configure for Java Web Start:

1. Open the `mime.types` file from the Apache `conf` directory.
2. Add the line:

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

   to the end of the file and save.
3. Restart apache.

Microsoft Internet Information Server (IIS)

The standard IIS 5.0 that come with Windows XP appears to support JWS by default, although it is not listed on the Mime Types in IIS page [MTIIS]. If you do find you need to add the JNLP mime type, follow these steps:

1. In the IIS snap-in select the website to add the mime type to and bring up the Properties dialog box.
2. Select the HTTP Headers tab.
3. Under MIME Map, click the File Types tab and select New Type.
4. Type `.jnlp` in the Extension field and `application/x-java-jnlp-file` in the Content Type field, and then click OK.

Publishing an Application

Once a web server has been configured, all that is left is to publish the JAR file to the server, along with a Java Network Launch Protocol (JNLP) file that describes how to launch the application and a hypertext link to that same JNLP file. JNLP files can be very simple or very involved. Below is the simplest possible JNLP file that will get the `HelloJavaWebStart` application to launch:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="http://myserver/apps">
      <information>
            <title>JavaWebStart</title>
            <vendor>Paul Grenyer</vendor>
      </information>
      <resources>
            <j2se version="1.2+"/>
            <jar href="HelloJavaWebStart.jar"/>
      </resources>
      <application-desc/>
</jnlp>
```

- The `codebase` attribute specifies the base location for all relative URLs specified in `href` attributes in the JNLP file.
- The `information` element contains other elements that describe the application and it's source. The `title` and `vendor` elements are the bare minimum.
- The `resources` element describes all the resources that are needed for an application.
  - The `j2se` element specifies what version of Java to run the application with.
  - The `jar` element specifies the JAR file to launch.
- The `application-desc` element denotes this is the JNLP file for an application.

A simple hypertext link is needed to launch the application. Assuming that the JAR file and a JNLP file called `HelloJavaWebStart.jnlp` have been copied to `http://myserver/apps` the following link will allow the application to be launched:

```
<a href="http://myserver/apps/HelloJavaWebStart.jnlp">Hello Java Web Start</a>
```

That completes a simple example of creating and deploying a Java Web Start application. Yes, it really is that easy. However, if you need to access resources such as images it does get a little more complicated, but not much.

Retrieving Resources from JAR files

As mentioned previously, any resources used by a Java Web Start application must be included in the JAR file. The Java Web Start Application Development Considerations states the following:

> Java Web Start only transfers JAR files from the Web server to the client machine. It determines where to store the JAR files on the local machine. Thus, an application cannot use disk-relative references to resources such as images and configuration files.
>
> All application resources must be retrieved from the JAR files specified in the resources section of the JNLP file, or retrieved explicitly using an HTTP request to the Web server. Storing resources in JAR files is recommended, since they will be cached on the local machine by Java Web Start.
>
> The following code example shows how to retrieve images from a JAR file:

```
// Get current classloader
ClassLoader cl = this.getClass().getClassLoader();

// Create icons
Icon saveIcon  = new ImageIcon(cl.getResource("images/save.gif"));
Icon cutIcon   = new ImageIcon(cl.getResource("images/cut.gif"));
...
```

The example assumes that the following entries exist in one of the JAR files for the application:

```
images/save.gif
images/cut.gif
```

To retrofit this on the `HelloJavaWebStart` application I need an image (the ACCU [ACCU] logo will do nicely) in a subdirectory to the directory where `HelloJavaWebStart.java` is, called images. The application also needs slightly refactoring so that an instance is available to call `getClass` on and the label text needs to be replaced with the image:

```
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloJavaWebStart extends JFrame
{
	private HelloJavaWebStart()
	{
		this.setTitle("Java Web Start!");

		ClassLoader cl = this.getClass().getClassLoader();
		ImageIcon image =
			new ImageIcon(cl.getResource("images/accu_logo.gif"));
		JLabel label = new JLabel(image, JLabel.CENTER);

		this.add(label);
		this.setSize(250,100);
		this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
	}

	public static void main(String[] args)
	{
		new HelloJavaWebStart().setVisible(true);
	}
}
```

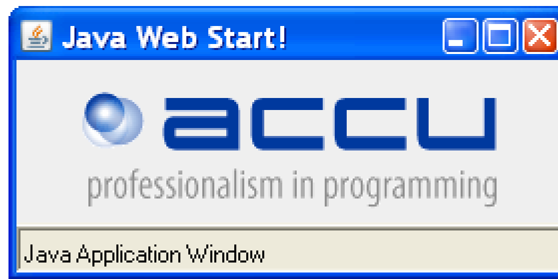
It is important that paths to images use forward slashes, otherwise they will not be found. The path is the relative path in the JAR file. If you use Ant to build the new HelloJavaWebStart application, but run the `.class` file or run it from Eclipse:

```
java HelloJavaWebStart
```

all is well. However, if you try and run the JAR file an exception is thrown as the image cannot be found. This is because the image has not been included in the JAR file. To do this the `jar` task must be modified to include the image:

```
<jar  jarfile = "HelloJavaWebStart.jar"
        basedir="${basedir}"
        manifest="manifest.txt"
        includes="*.class, images/*.*"/>
```

As you can see above, `images/*.*` has been added to the include attribute. This tells Ant that as well as all the `.class` files, it should include all the files in the `images` subdirectory. Run Ant again and launch the application via the JAR file locally and then copy it to the web server and try it there:



So adding a static image to the JAR was not that bad. What is more interesting is allowing the user to select the image at runtime.

Allowing Local Access

As mentioned previously, if a Java Web Start application wants to access local resources, such as the file system, it must be digitally signed and the JNPL file configured to allow it. Let's start by modifying the `HelloJaveWebStart` application to open a file chooser dialog and allow the user to select the image that is loaded:

```
import javax.swing.ImageIcon;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;

public class HelloJavaWebStart extends JFrame
{
	private HelloJavaWebStart()
	{
		this.setTitle("Java Web Start!");

		final JFileChooser chooser = new JFileChooser();

		if (chooser.showOpenDialog(this) !=
				JFileChooser.CANCEL_OPTION)
		{
			ImageIcon image =
				new ImageIcon(
					chooser.getSelectedFile()
						.getAbsolutePath());
			JLabel label = new JLabel(image, JLabel.CENTER);

			this.add(label);
		}

		this.setSize(250,150);
		this.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
	}

	public static void main(String[] args)
	{
		new HelloJavaWebStart().setVisible(true);
	}
}
```

When this application is built, with the Ant build file (removing the JARing of the images directory as it is not needed for this example), and run locally it allows the user to select and display an image from the hard disk. If the JAR file is deployed to the web server and the same test run, the error “access denied” is given.

This error is caused by the lack of security permissions in the JNLP file. These can be added by modifying the JNLP file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jnlp codebase="http://myserver/apps">
      <information>
            <title>JavaWebStart</title>
            <vendor>Paul Grenyer</vendor>
      </information>
      <resources>
            <j2se version="1.2+"/>
            <jar href="HelloJavaWebStart.jar"/>
      </resources>
      <application-desc/>
      <security>
            <all-permissions/>
      </security>
</jnlp>
```

Running the application from the web server now gives the error "Unsigned application requesting unrestricted access to system."

For this to work the JAR file must be signed using the Java SDK `keytool` and `jarsigner` tools. Open a command prompt and move to the directory holding the source files for `HelloJavaWebStart`. Then follow these steps:

1.Create a key store, called `myKeystore` with the alias "`myself`" by entering the following command and entering the required information at the prompt:

keytool -genkey -keystore myKeystore -alias myself

This should give the following output and create a file called `myKeystore`:

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Paul Grenyer
What is the name of your organizational unit?
  [Unknown]:  Marauder
What is the name of your organization?
  [Unknown]:  Marauder
What is the name of your City or Locality?
  [Unknown]:  Norwich
What is the name of your State or Province?
  [Unknown]:  Norfolk
What is the two-letter country code for this unit?
  [Unknown]:  UK
Is CN=Paul Grenyer, OU=Marauder, O=Marauder, L=Norwich, ST=Norfolk,
C=UK correct?
  [no]:  yes

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

2.Create a certificate by entering the following command and the password entered in the previous step:

```
keytool -selfcert -alias myself -keystore myKeystore
```

3.To check that the key store and certificate have been created, enter the following command using the same password as before:

```
keytool -list -keystore myKeystore
```

This should give output similar to the following:

```
Enter keystore password:

Keystore type: JKS
Keystore provider: SUN

Your keystore contains 1 entry

myself, 18-May-2008, PrivateKeyEntry,
Certificate fingerprint (MD5):
77:72:06:EC:18:2F:00:85:8E:E8:A8:EE:74:69:F9:EF
```

4.Finally, to sign the JAR file enter the following, and the same password:

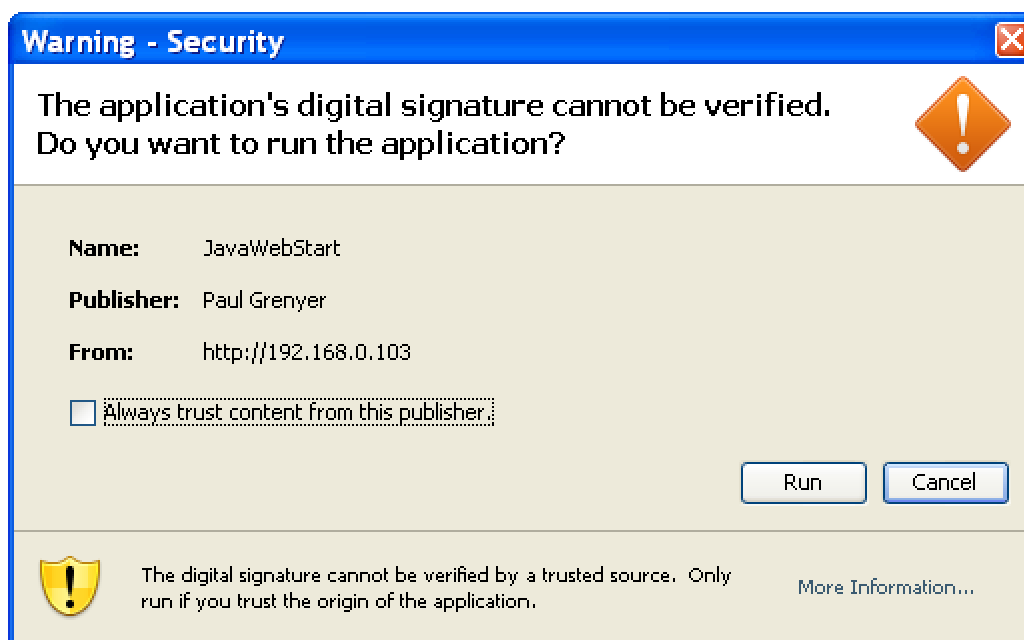
```
jarsigner -keystore myKeystore HelloJavaWebStart.jar myself
```

which should give the following output:

```
Enter Passphrase for keystore:

Warning:
The signer certificate will expire within six months.
```

Now copy the JAR file over and try the application via the web server again. You should be presented with a dialog similar to this:



Click run to launch the application with full access to local resources.

This is not quite the end of the story though. Although the key store and certificate only need to be created once, signing the JAR file needs to be done every time the JAR file is deployed, so should be part of the Ant build file. Ant has a `signjar` task that does this:

```
<project name = "HelloJavaWebStart" basedir=".">
            <javac srcdir = "${basedir}"  />
            <jar  jarfile = "HelloJavaWebStart.jar"
                        basedir="${basedir}"
                        manifest="manifest.txt"
                        includes="*.class"/>
            <signjar jar="HelloJavaWebStart.jar"
                        keystore="myKeystore"
                        alias="myself"
                        storepass="secret"/>
</project>
```

- The `jar` attribute specifies the JAR file to sign.
- The `keystore` attribute specifies the key store to use.
- The `alias` attribute specifies the alias to use.
- The `storepass` attribute specifies the password for the key store.

The output from the new task looks like this:

```
[signjar] Signing JAR: HelloJavaWebStart.jar to HelloJavaWebStart.jar as
myself
[signjar]
[signjar] Warning:
[signjar] The signer certificate will expire within six months.
[signjar] Enter Passphrase for keystore:
```

Although it appears from the output that the password is requested, it is in fact automatically entered by the `signjar` Ant task.

The Java Web Start Guide section on Application Development Considerations reminds us that:

> Note that a self-signed test certificate should only be used for internal testing, since it does not guarantee the identity of the user and therefore cannot be trusted. A trustworthy certificate can be obtained from a certificate authority, such as VeriSign or Thawte, and should be used when the application is put into production.

Conclusion

So here was are at the end. As a learning experience, getting to grips with Java Web Start is actually quite straightforward. Of course there is plenty more that can be configured in the JNPL files, especially in terms of security and versioning and plenty more that Ant can do, but that's outside the scope of this article.

If you are still not sure of the advantage Java Web Start over the usual installer method of deploying applications, let me describe a situation I have been in on many occasions.

It is 7am in the morning and I'm in a deserted office moving from PC to PC installing the latest release of the software. It gets to 9am and everyone starts arriving. But it is ok, all the machines have been updated.

So I go back to the development office to find the phone ringing. As I listen to the user explaining the problem with the new release I realise it's an easy fix, but will require another install to every machine. The whole business grinds to a halt while the bug is fixed and then gets delayed a further two hours while everyone's machine is updated.

If I had used Java Web Start I would not have had to get into the office at 7am to do the release and any post release problems could be deployed more quickly and easily once fixed. In fact releasing involves building some signed JARs, copying them to the webserver and asking everyone to restart.

Easy!

References

[Eclipse] Eclipse: http://www.eclipse.org/

[JWS] Java Web Start: http://java.sun.com/products/javawebstart/

[ANT] Ant: http://ant.apache.org/

[JavaSwing] Java Swing: http://en.wikipedia.org/wiki/Swing_(Java)

[JWSG] Java Web Start Guide:
http://java.sun.com/javase/6/docs/technotes/guides/javaws/developersguide/contents.html

[Apache] Apache: http://httpd.apache.org/

[ACCU] ACCU: http://www.accu.org

[MTIIIS] Mime Types in IIS: http://technet.microsoft.com/en-us/library/bb742440.aspx