Building an Online Entry Form with WebAF® (And a Little Java®)
Frederick Pratter, Eastern Oregon University

Introduction

The AppDev Studio® software suite from SAS® is a comprehensive set of tools for developing web applications. This paper is intended as a quick introduction to webAF, the included IDE (Integrated Development Environment) for building dynamic web content based on SAS data. Once the initial system configuration has been established, it is possible to provide dynamic access to SAS datasets in real-time for thin clients that do not have any SAS software available. This is accomplished using the JavaServer Pages® technology from Sun Microsystems®, along with some server-side SAS resources.

Obviously, there is far more material than could be covered in a single conference paper, but the general steps in the procedure can be summarized as follows:

1. Start a SAS job spawner on a SAS/Connect® server
2. Start a Tomcat® session on a web server
3. Create a web application base directory on the server
4. Register a connection to the spawner
5. Create a new webAF project
6. Add JSP content to the project
7. Stir well and serve.

The following example is based on an online examination application that was developed to demonstrate how to build an entry form using webAF. Hopefully, the examples provided will give a sense of how easy it can be to create complex web forms using SAS (and a little Java).

Preliminaries: How to Hit the Ground Running

Before starting webAF, it is important to perform several preliminary housekeeping tasks, summarized as steps 1-3 in the list above. Explaining what these do and why they are necessary can be a little complicated.

Step 1: Remote Data Services using SAS/Connect

Using SAS/Connect requires that a server program be started on the remote system where the data resides. This program is called a Remote Host Spawner. The spawner program listens for TCP connections to the host, just like a web server but on a different port. Starting the spawner is handled differently on Windows® and UNIX®, although the principle is the same on both platforms.

On Windows, the spawner program is stored in the SAS root directory. Just run spawner.exe from a DOS command window, or use the handy SAS menu selection Start>Programs>AppDev Studio>Services>SAS V8>Start SAS Connect Spawner, if AppDev Studio has been installed on the server. Under Windows 2000 and XP it is possible to install the spawner as a service, so that it will run as a background process whenever the server is started. For a complete list of the available startup
options, see *Starting and Stopping SAS/Connect* in the SAS System Help contents under *Help on SAS Software Products>SAS/Connect Software*.

On UNIX, the spawner program is installed by default under the SAS root directory in `utilities/bin/sastcpd` and can be started with the following command:

```
sastcpd -service spawner -shell &
```

The `sastcpd` program runs by default as a daemon, so it is not necessary to use `nohup`. Again, if you want it to restart when the server reboots, install the script under the `init.d` directory.

In either case, the job spawner must be running in order to allow remote connections to the SAS data on the server.

**Step 2: Tomcat Java Servlet Engine**

In order to make use of JavaServer Pages, it is necessary first to understand how servlets work (see [java.sun.com/products/servlet](http://java.sun.com/products/servlet)). Java servlets are the server-side equivalent of Java client-side applets. Like JavaScript® and applets, servlets can only be run from within a web browser; there is no_main_method as in Java console applications.

Servlets require both a web server and a servlet engine. The function of this engine is to load the servlet `.class` file into the *Java Virtual Machine (JVM)* running on the server. The engine can then run the servlet. (The `.class` file is not reloaded into the JVM again after the first time; usually, it is necessary to restart the server when the `.class` file changes, but most servlet engines now include options to reload `.class` files automatically when they are updated.)

The most widely available servlet engine is *Tomcat* from the Apache Software Foundation’s Jakarta Project (see [jakarta.apache.org/tomcat](http://jakarta.apache.org/tomcat)). This engine is included in AppDev Studio and must be started before any JavaServer Pages can be viewed. Note that Tomcat is used instead of the regular web browser. In the default SAS configuration shown in the following examples, the engine is started on port 8082 instead of port 80, as would be the case for the Apache or IIS web servers.

In order to start the SAS-supplied servlet engine, AppDev Studio must be installed on the web server. If the web server is not a Windows-based platform, an appropriate servlet engine from the Apache organization can be used instead. The details of this are beyond the scope of this paper, but are discussed in the references listed at the end. For the sake of this discussion, it will be assumed that the default AppDev Studio engine is available.

To start the servlet engine, select **Start>Programs>AppDev Studio>Services>Start Java Web Server**. On a properly configured system, this will result in a command window that displays the ongoing status of the server. It is also possible to start the servlet engine from within the webAF development environment, as shall be seen.

**Step 3: Web Application Base Directory**

SAS webAF, like Microsoft’s Visual Studio® and many other development environments, depends on the concept of a software Project. This is simply a collection of files relating to some specific
application. SAS uses a specific project directory containing a project file with the extension .afx to keep track of this collection. The default directory will be something like the following:

C:\AppDevStudio\WebAF\Projects\MyProject

In addition to the project directory, however, webAF will put all of your JSP, servlets and any other web application components in a project-specific Web application base directory. Basically, you have two choices here: the default directory and the one SAS recommends. (There is probably some good reason why these are different.) The default value is to use the same folder as the .afx project file. The recommended one is a sub-folder of the AppDev Studio webapps directory; for example

C:\AppDevStudio\WebAppDev\webapps\MyProject

That is, each project has two associated directories: the Project folder, including the information that AppDev Studio needs to manage the project, and the Web application base directory, which contains the actual project files.

If you decide to use this recommended approach, you have a little preliminary work to do first. The SAS web application templates directory

D:\AppDevStudio\WebAppDev\templates

contains a couple of web application “starter” directories, called empty and sasads. Of course empty is not really empty. It has the following structure:

- empty\WEB-INF
- empty\META-INF
- empty\WEB-INF\classes
- empty\WEB-INF\lib

along with a few cleverly structured files that assume you have used the default layout as recommended in the previous section.

The sasads directory has a similar organization, but also includes a folder called sasads\assets that contains a bunch of handy images in GIF format. The sasads folder is used to access the SAS custom tag library; for example it contains the tag library descriptor file sasads.tld. SAS webAF uses Java custom tags as controls when building pages; see the references at the end of the paper for more details about tag libraries.

In order to create a SAS web application base directory, you need to copy one of these two folders (you can’t go wrong using sasads) into webapps. Now rename it, using the same name as your project.

Assuming you want to call your project examples, the new directory will be something like

D:\AppDevStudio\WebAppDev\webapps\examples

Be sure to create this folder first, before you create the project!
Diving into webAF

Step 4. Register Connections

The first step to using remote data service in webAF is registering one or more connections to a SAS job spawner. From the webAF main menu, select **Tools>Register Connections**. (You don't need to have a project open; a connection can apply to any webAF project.) Something like the following list of "Persisted Connections" should appear:

![Persisted Connections](image1)

**Figure 1. Register Connections**

When you start webAF for the first time, you will only see one connection: the default one. To create another connection, click on **New**. This will display the **New Connection** window below:

![New Connection](image2)

**Figure 2. Create New Connection**
For this example, check the box marked SAS server and web server are the same. That is, the spawner is running on the local host. Obviously, it is possible to connect to a remote host using the Connection Wizard, but for the sake of simplicity we will assume that the data are on the local system.

Fill in a valid user name and password to connect to the host. That’s it; the wizard will do the rest.

Click on the Test tab, and then on Check Connection. The system will attempt to connect to localhost, that is, to itself. The connection test window displays the following message:

 Threaded connection test starting...

 SAS session instantiation information

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Timeout</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hello&gt;</td>
<td>120</td>
<td>sas -dmr -comamid tcp -noterminal -cleanup</td>
</tr>
<tr>
<td>PORT=</td>
<td>60</td>
<td>// Connection to SAS established at this point</td>
</tr>
</tbody>
</table>

Host: localhost
Port: 2323

Intransients differing from the default

<table>
<thead>
<tr>
<th>Property</th>
<th>Default</th>
<th>Modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>debugTelnetConnectClient</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>initialStatement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>isAppletCodebaseRelative</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>logTrap</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>persistedName</td>
<td>&lt;Custom Connection&gt; localhost</td>
<td></td>
</tr>
<tr>
<td>serverArchitecture</td>
<td>PC</td>
<td></td>
</tr>
</tbody>
</table>

Connection failed:
java.lang.Exception: Connect.C75.ex.txt: Cannot connect to telnet session.
com.sas.net.connect.TelnetClientException: Connection refused: no further information
Connection refused: no further information
java.net.ConnectException: Connection refused: no further information.

>>> Connection expecting Connect Spawner; you may need to start it.

Figure 3. Failed Connection to local host

Oops. You forgot to start the spawner on the local system! So just start the spawner as described in Step 1 above. Now you should see the following message:

 Threaded connection test starting...
Telnet session established on Mon Jun 16 17:51:47 PDT 2003
Telnet client: com.sas.net.connect.SASTelnetClient
Host: ASTERIX
Port: 2323

Looking for message from host containing one of the following

   Hello>

Received: Hello>

Sent: sas -dmr -comamid tcp -noterminal -cleanup

Looking for message from host containing one of the following

   SESSION ESTABLISHED

Received: SESSION ESTABLISHED

NOTE: Copyright (c) 1999-2001 by SAS Institute Inc., Cary, NC, USA.
NOTE: SAS (r) Proprietary Software Release 8.2 (TS2M0)

NOTE: This session is executing on the WIN_PRO platform.

NOTE: SAS initialization used:
   real time 0.44 seconds
   cpu time 0.14 seconds

1   %put RemoteSASInfoStart &SYSVER RemoteSASInfoEnd;
RemoteSASInfoStart 8.2 RemoteSASInfoEnd

NOTE: PROCEDURE PRINTTO used:
   real time 0.01 seconds
   cpu time 0.01 seconds

NOTE: SAS Server: Authorization commencing...
NOTE: SAS Server: Client LOGON
NOTE: NEW task=3 factory=8387 oid=8425 class=sashelp.prdauthuserinfo.class
NOTE: NEW task=3 factory=8387 oid=8505 class=SASHELP.RSASMOD.SRVINFO.CLASS
NOTE: Ofactory : _term
NOTE: TERM task=3 factory=8387 oid=8505
NOTE: TERM task=3 factory=8387 oid=8425
NOTE: SAS Server: Client LOGOFF
NOTE: Stopping task taskid=3 curtask=1

Success!!

Figure 4. Successful Connection to local host

You now have a connection to your local system. To create a connection to a remote PC just change the value in the Host Name text box from "localhost" to the name of your remote system and supply a valid user ID and password. The way TCP works, as long as both ends of the connection are talking over the same port number, one end can be on the same system as the other end (the "local" host) or on a computer in Australia (assuming you are not in Australia when you're reading this).
Step 5. Creating a JSP Project

Now you are ready to actually create the web application. First, however, as noted above it is necessary to create a new project to hold the JSP project. In webAF, select File>New. From the Projects tab, select JavaServer Pages Project. You should see something like the resulting screen:

![Creating a New JSP project](image)

Three text boxes show up: Project name, Java package and Location. The only one of these that you have to specify is the project name—in this case Examples. Leave the second text box blank. The result is that the Project Wizard does not create a hierarchy of directories to store Java class files (see the webAF Help Topic “Assigning a Package Name” for more details). The Location field will automatically fill in with the default AppDev Studio Project directory name; again, this is probably what you want to do.

Selecting OK should result in a screen that looks something like the following example:
The Web application base directory is specified as described in the discussion above. The default value is the project location from Figure 5. You need to change this value to the new folder you created by copying and renaming the template:

```
D:\AppDevStudio\WebAppDev\webapps\examples
```

Checking the box labeled Invoke JSP using the following URL offers two choices: the Default Web server and the WebAppDev Web Server. In this example the second choice is selected. As we saw in Step 2, be sure this server is started before you try to run any applications from within webAF.

The third text box, Invoke JSP using the following URL, also requires some additional explanation. The string

```
http://localhost:${WebPort}/${ProjectName}/index.jsp
```

is comprised of a series of environment variables of the form ${varname}. The selection arrow at the right hand side of the text brings up a list additional field values that can be automatically inserted into the text. In this case, the generated URL is

```
http://localhost:8082/Examples/index.jsp
```
requesting the page `index.jsp` from the `WebAppDev Web Server`. The full pathname to this page is

```
d:\AppDevStudio\WebAppdev\webapps\Examples\index.jsp
```

Unlike most Windows commands, however, URLs are case sensitive. Unless you are sure that the true pathnames agree completely with the constructed ones, it is almost certainly better not to use the environment values in the URL string, but instead just hard-code the actual path to the web application directory.

The fourth text box is the name of the initial JSP page in the new project; the default is `index.jsp` which is probably what you want to call it.

If all of the parameters have been specified as illustrated, selecting Next should result in the following confirmation page. Press Finish to create the project.

![Project Wizard Summary](image)

Figure 7. Project Wizard Summary
Step 6. Creating a new JavaServer Page

For the online examination example the first screen prompts the student to select a test from the list available. This list is stored as a SAS dataset on the server.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Astronomy</td>
</tr>
<tr>
<td>2</td>
<td>Chemistry</td>
</tr>
<tr>
<td>3</td>
<td>Geology</td>
</tr>
<tr>
<td>4</td>
<td>Meteorology</td>
</tr>
<tr>
<td>5</td>
<td>Physics</td>
</tr>
<tr>
<td>6</td>
<td>Mathematics</td>
</tr>
<tr>
<td>7</td>
<td>Computer Science</td>
</tr>
</tbody>
</table>

Figure 8. Sample Data: EXAMS.TEST

A new JSP page can be readily constructed using webAF, by simply dropping widgets onto the page. To create a new page in an open project, just select **File>**New from the main menu. The following screen should appear:

Figure 9. Creating a New JavaServer Page

Enter a new **File name**, in this case **index.jsp** since this is the name of the start page specified above in the previous step (see Figure 6).

The **JSP/Servlet** window should open with a new blank page. Note that the first line says
This is the link to the `sasads` tag library which, as we will see below, is used by `webAF` to create form elements. Note that the `uri` (Uniform Resource Identifier) looks a lot like an anchor tag. You do not need to be connected to the internet in order to display a page containing this tag; the local Java web server understands this reference not as a link, but instead as an identifier. This is just the name of the `sasads` resource, not its location.

The rest of this discussion is based on the finished `index.jsp` page shown in Figure 10 below. Each of the components is discussed in turn, but it is useful to view the entire page at once in order to understand where the process is leading.

```jsp
<%@taglib uri="http://www.sas.com/taglib/sasads" prefix="sasads"%>
<%@ page import="com.sas.collection.StringCollection" %>
<sasads:Connection id="connection1" scope="session" initialStatement="libname EXAMS 'd:\exams';" />
<sasads:DataSet id="table1" connection="connection1" dataSet="EXAMS.TESTS" />
<% // Get unique test names and codes from data set, // add collections to page context
    pageContext.setAttribute("codes", new StringCollection(table1.getFormattedColumn(1)));
    pageContext.setAttribute("labels", new StringCollection(table1.getFormattedColumn(2)));
%>
<p>-- begin HTML --%</p>
<html>
<head>
    <title>JSP Examples</title>
</head>
<body>
<h1 style="color: blue; text-align: center">On-line Exam Demo</h1>
<sasads:Form method="get" action="page1.jsp" style="text-align: center;">%
<sasads:Choicebox id="test" prolog="<strong>Select Test: </strong>"
    model="codes" descriptionModel="labels" />
<sasads:PushButton text="Begin" /></p>
</sasads:Form>
</body>
</html>
```

Figure 10. Using DataSetInterface to Populate a Listbox Control
Adding a Connection Object

Selecting the SAS tab in the webAF IDE brings up a set of 9 controls. Dropping a Connection control on the Source page results in the following tag:

```xml
<sasads:Connection id="connection1" scope="session" />
```

Note the sasads custom namespace; the Java code for this control is available from the template library copied in Step 3. This default template has to be modified for the specific connection required. If you know the attribute values for the connection you can just type them into the source window. It is probably easier however to modify the connection properties from the Components tab of the Project Navigator window.

Right click on the selected component in the left window (here sasads:Connection – connection1) and a menu appears that can be used to start the Customizer or change component properties individually. In general, if a component customizer is available it supports editing of the available component properties. The Customizer brings up a connection editor. In either case—editing the properties directly or using the Customizer-- the JSP code in the Source tab is rewritten with the new attributes.

In this example you also need to add an initialStatement attribute allocating a libname on the local host.

```xml
initialStatement="libname EXAMS 'd:\exams';" />
```

Note that two slashes are required; since the interface uses the backslash character as an escape, ‘\’ gets translated into ‘\’ when it is passed to SAS. Also, use single quotes around the libname directory, since the attribute itself is enclosed in double quotes.

Adding a Dataset Object
Once the connection has been instantiated, the DatasetInterface control must be added to reference the specific EXAMS.TESTS table. As Figure 12 shows, in order to populate the page, it is necessary to include a short JSP scriptlet to set the page attributes.

![JavaScript Example](image)

Figure 12. Java Scriptlet

Note that column 1 in the data set shown in Figure 8 is the test number and column 2 is the name of the test. The data are passed to the page context as Java StringCollection objects. In this instance, the records are provided by the `getFormattedColumn()` method of the dataset class. (As might be expected, there are also methods to `getFormattedRows()` and other table components.) The data for the first variable is added to the page as a collection called `codes` while the second variable is added as the collection `labels`. These two collections are used below on the HTML form.

**HTML Static Text**

A JavaServer Page is just an HTML page with some special tags added. Figure 13 shows the static web page code. Note that a CSS (Cascading Style Sheet) `style` tag is used to format the heading.
Adding a Form

In order to display dynamic data in HTML, a form element is usually required. SAS has supplied a custom tag to support this action.

The form control is available from the Form elements tab of the IDE, as are the Choicebox and PushButton tags. These are just standard HTML elements, with the expected functions. Pressing the “Begin” button submits the page and calls the first page of the selected exam: page1.jsp. This page can be constructed in a similar fashion, just by putting together simple elements using the IDE. The other three pages of this application, page1.jsp, page2.jsp and finish.jsp are attached as an appendix for the interested. It is to be hoped that this discussion has provided enough information in order to be able to puzzle out how these were constructed and why they work.
Step 7. Putting it All Together

There are three different ways to start the Tomcat engine from within webAF.

1. From the main webAF menu, select **Tools>Service>Start Java Web Server**.
2. Press the F5 function key.
3. Click the **Go** button on the toolbar.

Whichever method is chosen, a window appears at the bottom of the display with something like the following text.

```
Starting tomcat. Check logs/tomcat.log for error messages
2003-06-16 07:13:15 - ContextManager: Adding context Ctx( /examples )
2003-06-16 07:13:15 - ContextManager: Adding context Ctx( )
2003-06-16 07:13:15 - ContextManager: Adding context Ctx( /ServletExample )
2003-06-16 07:13:16 - ContextManager: JspClassDebugInfo: Enabling inclusion of class debugging information in JSP servlets for context "/examples".
2003-06-16 07:13:16 - Scratch dir for the JSP engine is: D:\AppDevStudio\WebAppDev\work\localhost_8082%2Fexamples
2003-06-16 07:13:16 - IMPORTANT: Do not modify the generated servlets
2003-06-16 07:13:16 - ContextManager: JspClassDebugInfo: Enabling inclusion of class debugging information in JSP servlets for context "".
2003-06-16 07:13:16 - ContextManager: JspClassDebugInfo: Enabling inclusion of class debugging information in JSP servlets for context "/ServletExample".
2003-06-16 07:13:17 - PoolTcpConnector: Starting HttpConnectionHandler on 8082
2003-06-16 07:13:17 - PoolTcpConnector: Starting Ajp12ConnectionHandler on 8083
```

Figure 15. Java Web Server Messages

Note that the Java web server is started on port 8082. (Do not forget to start the **SAS Job Spawner** as well!)

Selecting the “Execute in Browser” button from the tool bar results in the page shown in Figure 17. Clicking the **Begin** button on the form passes the test number of the desired exam subject as a parameter to `page1.jsp`, which selects the appropriate questions from another table in the same SAS data library.
Conclusion

As should be obvious from this brief introduction, SAS/AppDev Studio is an extremely powerful and flexible collection of tools for web development. It also has a lot of moving parts, and new functionality is being added on an ongoing basis. The goal of this paper is to try to put all of the pieces together in a systematic overview, so that both novice and experienced web programmers can find the information and examples they need to get started using these tools effectively.

References

Conference Papers


SAS Resources

JavaServer Pages (a few of many)


And last (but not least)


Acknowledgements

*SAS®, and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are registered trademarks or trademarks of their respective companies.
Appendices

page1.jsp – Display the next question

```jsp
<%@taglib uri="http://www.sas.com/taglib/sasads" prefix="sasads"%>
<%@ page import="com.sas.collection.StringCollection" %>

<sasads:Connection id="conn" scope="session"
    initialStatement="libname EXAMS 'd:\exams';"/>

<sasads:DataSet id="table1" connection="conn"
    dataSet="EXAMS.TESTS" />

<sasads:DataSet id="table2" connection="conn"
    dataSet="EXAMS.QUESTIONS" />

<sasads:DataSet id="table3" connection="conn"
    dataSet="EXAMS.ANSWERS" />

<%// get parameters from URL
    String tnum = request.getParameter("test");
    String qparm = request.getParameter("question");
    int qindex = qparm != null ? Integer.parseInt(qparm) : 0;

    // get test name
    table1.setWhere("tnum eq " + tnum);
    String tname = table1.getFormattedCell(1,2);

    // select questions for this test
    table2.setWhere("tnum eq " + tnum);
    String[] qnum = table2.getFormattedColumn(2);
    String[] qtext = table2.getFormattedColumn(3);

    // select answers for this question
    table3.setWhere("tnum eq " + tnum + " and qnum eq " + qnum[qindex]);
    pageContext.setAttribute("answer_codes",
        new StringCollection(table3.getFormattedColumn(3)));
    pageContext.setAttribute("answers",
        new StringCollection(table3.getFormattedColumn(4)));
%>

<%-- begin HTML code --%>
<html>
<head>
<title>On-line Exam Demo: page1.jsp</title>
</head>
<body>
<h1 style="color: blue; text-align: center">On-line Exam: <%= tname %></h1>

<sasads:Form action="page2.jsp" method="get">
    <sasads:Hidden id="test" text="<%= tnum %>">
    <sasads:Hidden id="question" text="<%= qnum[qindex] %>>

<%-- display question --%>
(<%= qnum[qindex] %>) <%= qtext[qindex] %>

</sasads:Form>
</body>
</html>
```
page2.jsp – Display the correct answer

```java
<%@ page import="com.sas.collection.StringCollection" %>
<sasads:Connection id="conn" scope="session"
    initialStatement="libname EXAMS 'd:\exams';"/>
<sasads:DataSet id="table1" connection="conn"
    dataSet="EXAMS.TESTS" />
<sasads:DataSet id="table2" connection="conn"
    dataSet="EXAMS.QUESTIONS" />
<sasads:DataSet id="table3" connection="conn"
    dataSet="EXAMS.ANSWERS" />
<% // get parameters from URL
    String tnum = request.getParameter("test");
    String qnum = request.getParameter("question");
    String answer = request.getParameter("answer");

    // get test name
    table1.setWhere("tnum eq " + tnum);
    String tname = table1.getFormattedCell(1,2);

    // select questions for this test
    table2.setWhere("tnum eq " + tnum);
    String[] qtext = table2.getFormattedColumn(3);
    String[] qanswer = table2.getFormattedColumn(4);

    // test for end of exam
    int qindex = Integer.parseInt(qnum);
    String nextpage = (qindex-- < qtext.length) ? "page1.jsp" : "finish.jsp";

    // select answers for this question
    table3.setWhere("tnum eq " + tnum + " and qnum eq " + qnum);
    String[] avalues = table3.getFormattedColumn(3);
    pageContext.setAttribute("answer_codes",
        new StringCollection(avalues));
    String[] answers = table3.getFormattedColumn(4);
    pageContext.setAttribute("answers",
        new StringCollection(answers));

    // look up the right answer
    String correct_answer = new String();
    for (int i=0; i<answers.length; i++)
        if (qanswer[qindex].equals(avalues[i]))
            correct_answer = answers[i];
```
break;

// specify correct answer, compute score
String reply = new String();
if (answer.equals(qanswer[qindex]))
{
    reply = "That's right!";

    // compute test score, store as session value
    String s = (String) session.getValue("score");
    int score = (s == null) ? 0 : Integer.parseInt(s);
    session.setAttribute("score", String.valueOf(++score));
}
else
{
    reply = "Sorry. The correct answer is: " + correct_answer;
}

<%-- begin HTML code --%>
<html>
<head>
<title>On-line Exam Demo: page2.jsp</title>
</head>
<body>

<h1 style="color: blue; text-align: center">
    On-line Exam: <%= tname %></h1>

<blockquote>
<sasads:Form action="<%= nextpage %>">
<sasads:Hidden id="test" text="<%= tnum %>">
<sasads:Hidden id="question" text="<%= qnum %>">

<%-- display question --%>
(<%= qnum %>) <%= qtext[qindex] %>: 
<blockquote>
<sasads:Radio model="answer_codes" descriptionModel="answers" selectedItem="<%= answer %>">

<%-- display correct answer --%>
<p><%= reply %></p>
</blockquote>
<center><sasads:PushButton text="Next Question"/></center>
</sasads:Form>

</blockquote>

<%-- finish.jsp --%>
<html>
<head>
<title>On-line Exam Demo: finish.jsp</title>
</head>
<body>

<h1 style="color: blue; text-align: center">
    On-line Exam Demo</h1>

<center><strong>Your score: <%= score %></strong></center>

</body>
</html>
<%= session.getAttribute("score") %> correct out of
<%= request.getParameter("question") %> questions.
</strong></center>

<% // re-initialize test score
    session.invalidate();
%>

</body>
</html>