Paper 65-28

Java Servlets and Java Server Pages for SAS® Programmers: An Introduction

Miriam Cisternas, MGC Data Services, Carlsbad, CA
Ricardo Cisternas, MGC Data Services, Carlsbad, CA

ABSTRACT

AppDev Studio 2.0 provides a drag-and-drop interface to produce Java-based thin-client applications using Servlet technology, as well as support for Java Server Pages (JSPs). Those of you who have created Java Servlets using the drag and drop methods may be somewhat mystified by the Java code generated. What is an import statement, and what is the function of those curly braces? And just how do Servlets work, anyway? How do they interact with the web server, track a session and process browser cookies? Since Servlets can access any Java API, we hope to whet your appetite to learn more about Java and how to extend the Java classes that SAS provides in AppDev Studio. Finally, we will provide a brief introduction to JSP scriptlet and tag syntax.

INTRODUCTION

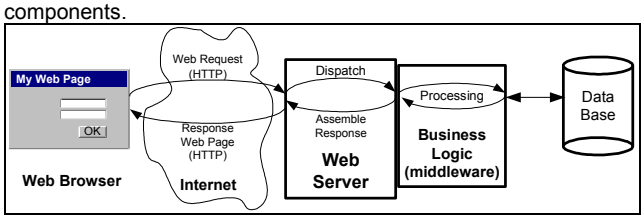
Publishing your data on the web is a great way to make it available to a large audience. To attain that goal, SAS Institute has created a number of technologies to enable web publishing of SAS data. These products have advantages and disadvantages for various applications.

It is for a good reason that SAS has invested heavily in developing tools for Java web solutions. Java has been for several years one of the preferred technologies to develop efficient and sophisticated web applications. Java can be executed on many platforms, work with a number of web servers and databases, and can interface with a number of software standards. Best of all, Java can extend the reach of SAS applications into the web while providing a solid and extensible software framework.

In this paper, we will describe the general workings of a dynamic web application. We then describe the way in which Java web applications work and contrast them to traditional CGI solutions (e.g. SAS IntrNet) explaining why Java is a more efficient web technology. The function of Java Servlets and JSPs is then described with an emphasis on how they work and how they differ from each other. We also explain the advantages of JSPs over Servlets for web page development and the unique uses of Servlets to handle non-HTML content. We then describe how to separate the Java code from the Servlets using Java Beans and introduce a Java web application architecture that relies on Servlets and JSPs working together to handle web content and presentation as separate activities. Then, we provide some background on the Java programming language and object oriented design. Finally, we give examples on displaying SAS data on the web using JSPs and servlets.

THE DYNAMIC WEB APPLICATION MODEL

A dynamic web application is one where at least some of the content is not hard-coded into an HTML page but instead is dynamically generated from a program after querying a data source. Dynamic web applications can access multiple data sources to put together any given web page; therefore, by definition, they are made up of many individual software components.



A typical web application will have a web client (web browser application), a web server, a business logic component, and one or more data sources such as a data base.

A web application is by definition an asynchronous client-server application. This means that the web client and web server do not block waiting for each other to complete an operation. However, the web server can never tell whether the user's browser is still available or the user's computer just caught fire and connectivity was lost forever.

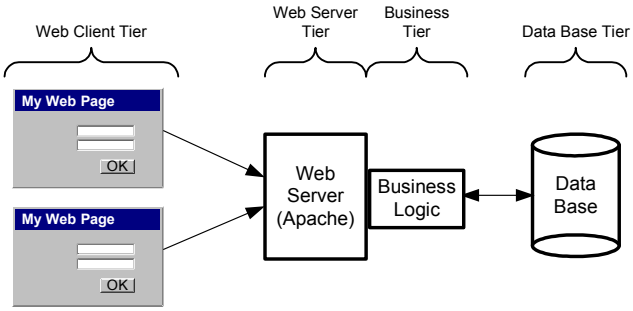
This behavior is a consequence of the choice of HTTP (HyperText Transport Protocol) as the underlying transport protocol of the web. HTTP is an asynchronous text-based protocol that is very easy to use and can handle all types of content. It can be teamed with SSL (calling it HTTPS) to provide secure communication over the web and it can also tunnel through firewalls. While these features give HTTP enormous flexibility, it gives headaches to web application designers who have to deal with problems such as how do I know the user is not coming back and it is time to close my database connections?

The operation of a web application is seen as a series of exchanges between the web client and the web server. For each exchange, a web client submits a request to the web server over the network. The web server handles the request and dispatches it to a separate piece of software for processing. The software that processes the web requests is known as the business logic of the application because it contains all the rules for analyzing and processing data. The business logic can access multiple data sources such as the file system, databases, or other servers to fulfill the processing of a request and generate the response data. Once the business logic has produced the response data within an HTML layout, it is sent back to the web server which then sends it back to the web client application. The web browser then displays the information on the screen where the user can see it, make a choice and start the process over again.

The use of multiple software components in a web architecture is useful because each piece of software is finely tuned to a particular purpose. For instance, the web server is designed to handle and dispatch multiple simultaneous requests as quickly and efficiently as possible, while the business logic encapsulates all the analysis and processing rules of our web application. A single web server can be used to host multiple web applications at the same time while delegating the processing of each request to the business logic component of the corresponding web application. We further discuss the specialization of software components and its effect on software design when we describe how JSPs and servlets work together.

WEB APPLICATIONS HAVE TIERS

The web application components are organized in tiers. Each tier is responsible for a particular activity. The web client tier is responsible for rendering web pages and capturing user input and conveying it to a web server. The web server tier is responsible for receiving web requests from multiple clients, dispatching them for processing and sending the response messages back to the web clients. The business logic tier is responsible for processing user requests and generating a result and assembling a response web page to display that data. The data base (or data source) tier maintains a repository of all the system's data and makes it accessible to the business tier. The data base tier can be used by multiple web applications at the same time.



Base SAS falls naturally into the data base tier because it has all the storage attributes of a data base. However, the programs written in SAS that use these datasets can fall within the Business Logic Tier. SAS Institute has created a number of products to put SAS data on the web such as SAS IntrNet, AppDev Studio and WebEIS. These products fulfill the duties of the business tier.

For example, SAS IntrNet uses CGI (Common Gateway Interface) technology to quickly put together web pages that display data from a SAS database. However, simplicity is traded off for efficiency since CGI spawns a process for each request it handles, imposing a large processing overhead to serve each web page. This overhead can be ignored when the application use is light. However, for applications with heavy use (multiple users, many requests), a Java based solution is much more efficient because Java uses a single process and handles each web request by handing it to a thread from a pool.

ADVANTAGES AND DISADVANTAGES OF THE WEB APPLICATION MODEL

The advantages of web applications are many. First, your web application becomes instantly available across all the nodes of the network. Second, web browsers come pre-installed in most computers so the cost of deployment is smaller than with traditional fat clients. Software updates are restricted to the server reducing total cost of ownership. Third, the web application model provides security features (encryption, security certificates, etc.) that allow its use from any place on the internet if so desired.

The disadvantages of the web application model have to do with the difficulty of mastering a number of disparate technologies (HTTP, HTML, JavaScript, Java or CGI, etc.), the complexity of designing a multi-tier software application, the challenges relating to developing asynchronous client-server applications, and the difficulties in configuring all the software components.

WHERE DOES JAVA FIT IN?

When Java was first introduced, it was proposed as programming environment for embedded solutions. Later, it was pushed as a way to deliver content on web browsers using applets. However, Java's greatest success was to come as a server-side technology.

Java is especially well suited as a server solution for a number of reasons:

It has an excellent model for handling web requests using objects called Servlets. The object model is easily adapted to particular applications by creating individual Servlet classes.

- Each Servlet is processed by an individual thread making their processing very efficient and robust.
- Java provides APIs and libraries to most of the technologies used over the web such as XML, JDBC, CORBA, JNDI, LDAP, etc. This makes Java an excellent choice to program business logic since it is easy to write code in Java to interact with multiple software components accessible to a web server. It also eases the task of integrating legacy technologies with web systems.
- Java is an open standard which means that there are multiple vendors available for each software component based on Java technologies (such as a Servlet container) but that all the competing products are API equivalent. It also means that there is a selection of functionally equivalent Java web products that vary on price and performance. In fact, there are a number of open source projects developed in Java that are available for free. Yes, that's right, you can get free web servers, XML translators and all kinds of goodies and as a bonus they all include their source code!

A LITTLE TECHNICAL DIGRESSION

If you are familiar with the network technologies underlying the world wide web please skip to the next section.

The internet as we know it is a collection of interconnected computers (called hosts) that communicate using a common protocol: TCP/IP. TCP/IP stands for Transfer Control Protocol/Internet Protocol and it is simply a series of conventions that specify how two hosts send data to one another. The most important element of this convention is that all hosts need to be identified by a unique number known as their IP address. Each piece of data (or packet of data) sent over TCP/IP must contain the IP addresses of both the sending and receiving hosts. The data packets have additional information specifying the amount of data transmitted in the packet, the number of the packet if it is part of a larger data transmission, redundancy codes, etc.

One important feature of TCP/IP is that it is a guaranteed delivery protocol. That means that the sender will be notified if and when the destination host received the data message or, alternatively, if there was a problem during delivery. Another key feature of TCP/IP is that once a message is divided in packets, each individual packet may take any route available to arrive to its destination. This means that the packets making up a message may arrive in any order and they will be reassembled at their destination.

The TCP/IP protocol makes no requirement on the type of data being sent. In the case of the web, all the messages sent back and forth are text-based so it is more convenient to piggy-back a new protocol HTTP (HyperText Transfer Protocol) on top of TCP/IP to provide added information for the exchange of textual data between a client and a server computers. HTTP specifies a set of text messages that can be exchanged among hosts where each type of message indicates a specific action. The messages contain a few lines of directives called headers that identify the message type, among other things, and a series of lines of content.

The message headers also specify the type of the content. In general, most pages served in the web, use HTML (HyperText Markup Language) as their content because this language is used by browsers to lay out text on a page. However, an HTTP message can contain files of other types such as PDF, plain text, XML, GIF, etc. and this content type is always specified in the header.

The most important messages for a web browser come in two types: GETand POST. These messages are equivalent in that they both request data from a server using a set of parameters as arguments to the request. The difference is that while the GET message includes the parameters explicitly as part of the requested URL (a sequence of name/value pairs preceded by a question mark at the end of the URL string – also known as the CGI string), the POST message includes all arguments as part of the request message headers.

When a web page contains a form, the data collected in that form is included in the web request as a series of name/value pairs. The name/value pairs will be included in the CGI string if the form is sent to the server as part of a GET message or as part of the

HTTP headers if the form is send to the server in a POST message. If you look at the source code of an HTML file containing a form you will see that it will contain a <FORM> tag with an ACTION attribute indicating the URL of the destination host, and an METHOD attribute specifying the type of message (either GET or POST).

Finally, HTTP specifies a convention that describes how the web browser and web server can exchange small pieces of data repeatedly in every web request and how this data can be saved by the web browser in its host computer. This type of data is known as a HTTP cookie and is very useful for maintaining user identity between web browser requests and to have web sites welcome you by name when you open their main pages.

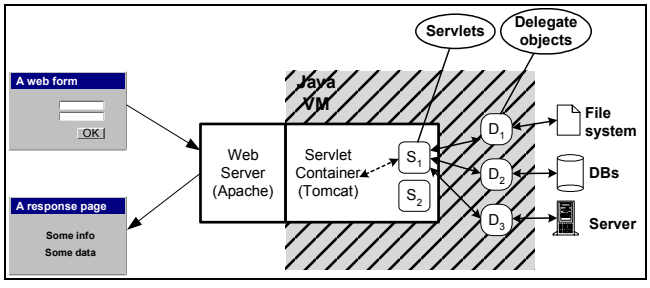
A web application needs to ascertain the identity of a user for every web request that is sent to the server to maintain the continuity of the user's session. One of the ways to achieve this is by creating an HTTP cookie which is unique to the web browser and is sent to the web server with every HTTP request. A session context is required for all web applications because most complicated tasks require processing several web pages before the task is completed and only a session context can hold the intermediate data items needed to complete the task.

THE LIFE OF A SERVLET

Java uses a special type of object to serve web content and handle web requests. These objects are called Servlets. All Servlets exist inside a Servlet container — a special type of web server in its own right where all the web requests are handled by a Servlet object. For most heavy duty applications it is best to use a Servlet container backed by a full scale web server since web servers are best tuned to deliver content and handle multiple requests, while Servlet containers are designed to efficiently execute code in a number of Servlet objects. This is the principle of specialized software components in action.

SAS AppDev Studio uses a web server and a Servlet container combination. In fact, it uses two open source products: Apache web server and the Tomcat Servlet container. These products are required only for AppDev Studio code development but can be exchanged for other Web Server/Servlet container product combinations during application deployment.

The picture below illustrates the resulting architecture. It is important to mention that the Apache web server is not written in Java while the Tomcat Servlet container, all the Servlets, and any other delegate objects used by the web application to talk to other servers or resources, are Java objects that reside within the same Java environment known as the Java Virtual Machine (VM). Apache works best as a native (non-Java) application because it needs every bit of added performance to listen and dispatch web requests as quickly as possible. Apache and Tomcat talk to one another using TCP/IP sockets and the HTTP protocol (the same protocol used for communication between web browsers and the Apache web server.)



The Servlet container is a software framework that manages the Servlet objects. A Servlet framework is a software library that follows the Broadway casting principle: "don't call us, we'll call you". In other words, the Servlet objects are created, used and destroyed by the Servlet container as necessary. For that reason, all Servlet objects must implement the same set of functions (methods) so that they can be invoked by the Servlet container.

The Servlet container also can manage as many instances of the same Servlet object as needed. If a particular web page is requested very often, it is more efficient for the Servlet container to delegate the processing of the simultaneous requests to different instances of the same class and not hold up the processing of further requests until the current request is fulfilled. The various instances of a Servlet are kept in a pool while not processing a request so they can be easily looked up by the Servlet container when needed.

The processing of a web request is handled as follows. When the web browser sends a request to the Apache web server, the web server identifies the web request as one to be handled by the Servlet container (based on its URL), and forwards it to the Tomcat Servlet container.

The Servlet container receives the web request and determines which Servlet is being invoked. It then looks for an instance of that Servlet in a pool or, if there is none, it creates a new instance. The Servlet container also parses the URL and the HTTP headers and cookies from the web request and puts them into a request object. The Servlet container then invokes the code in the Servlet instance within its own thread of execution passing it the request object. Again, the thread is first looked up in a thread pool. The code of a simple Servlet looks like this:

MyServlet.java

```
public class MyServlet extends HttpServlet
{
   public void init(ServletConfig config)
   { /* initialize global variables for this Servlet here */ }

   public void doGet(HttpServletRequest myRequest,
                     HttpServletResponse myResponse)
   {
      /* write a web page in HTML */
      PrintWriter out = new
                PrintWriter(response.getOutputStream());
      out.println("<html><head>");
      out.println("<title> Response Page </title></head>");
      out.println("<p> <body> <b> Hello World! </b>");
      out.println("</body></html>");
      out.close();
   }

   public void doPost(HttpServletRequest myRequest,
                     HttpServletResponse myResponse)
   { /* Handle a POST request from the browser */
      return doGet(myRequest, myReply);
   }

   public void destroy()
   { /* release resources allocated to this Servlet */
      /*(cached DB connections, etc.)*/  }
}
```

In Java, all functions are called methods and all code has to be part of a class (see below). In this case, our Servlet class is called MyServlet and it subclasses the HttpServlet class which is a Java library class for an abstract Servlet. All Servlet classes need to subclass HttpServlet and implement the four methods indicated above to operate within the Servlet container. The init and destroy methods are called only once when the Servlet object is created or destroyed. Their purpose is to include any code that acquires or releases resources used during the lifetime of a Servlet such as remote connections.

The methods doGet and doPost are meant to handle GET and POST HTTP request respectively. In our example, doPost

invokes the doGet method so both kinds of HTTP requests will return the same web page. The doGet method simply writes out a web page with the message “Hello World!” The web page is written to an object called the HttpResponse which is later used by the Servlet container to send the web response message back to the web server. Apache receives the web response message and relays it to the client’s web browser completing the web request/response cycle.

In our example, we simply created a page with static text. However, it would be just as simple to add text that was the result of some computation or that was obtained from a data source such as a SAS database.

The important thing to notice in this Servlet is that all the HTML code is created as characters strings by the Java class. The problem with this approach is that although the data content can be automatically updated if it comes from a database, any changes to the HTML layout require an update to the Java code and to have all Java classes recompiled. Furthermore, the Web designer needs to be familiar with Java or the Java engineer needs to be familiar with HTML. Clearly, there has to be better way.

JAVA SERVER PAGES TO THE RESCUE

The solution proposed by the Java designers was a new set of objects called Java Server Pages or JSPs. Before you give up due to sensory overload, we should tell you a simple truth: a JSP is just a Servlet inside out. What that means is that instead of being a Java class that creates HTML code as a Servlet does, a JSP is an HTML file that contains some Java code. To account for this difference, a JSP uses an file extension of .jsp instead of .html. In the following example, a JSP specifies the layout of an HTML page that displays the current time and date.

```
<%@ page import="java.text.SimpleDataFormat, java.util.Date"
%>
<%-- A scriplet to display the current date --%>
<%-- and time --%>
<%
 SimpleDateFormat formatter = new SimpleDateFormat
             ("yyyy/MM/dd 'at' hh:mm:ss a zzz");
/* get the current date/time */
Date currentTime = new Date();
/* format the date */
String dateString = formatter.format(currentTime);
%>
<html>
   <head><title> Welcome Page </title></head>
   <body>
   <h1>Welcome to my page!</h1>
   <%-- an expression for the date --%>
   The current time is: <%= dateString %>
</body>
</html>
```

It is immediately clear that the example contains more than just plain HTML. What is important to note is that all the Java code is enclosed between special brackets (“<%” and “%>”). These brackets allow the JSP compiler to extract the Java code. The brackets are also ignored by the web browser so it is possible to change a JSP’s file extension to .html to directly see what the layout of a page will look like. This is an invaluable tool while designing a web page layout.

The example above contains a JSP page directive as its first line. This directive indicates what Java classes are used by the JSP. In our example we are using the Date and SimpleDataFormat classes to obtain the current date and to format it into a string respectively. Lines 2 and 3 are just JSP comment lines that describe what the following piece of Java code does. The following lines are bracketed Java code (called a JSP scriptlet) which basically does three things: it creates an instance of the SimpleDateFormat class with a specific date format supplied by a String parameter; it creates an instance of the Date class called currentTime; and it formats the currentTime into a String object called dateString. An object of type Date takes the current time as its initial value when it is instantiated.

The rest of the example file contains the HTML code that produces the output page but it also includes a JSP expression that inserts the current value of the variable dateString where it is needed. A JSP expression is bracketed with (“<%=” and “%>” and will insert the result of evaluation of a Java expression where it is placed. A JSP expression does not contain arbitrary Java code like a JSP scriptlet does, it only contains Java expressions.

The lifetime of a JSP is similar to that of a Servlet with a couple of additions. First, when a JSP is invoked, the Servlet container will find the JSP file and translate it into a Servlet class automatically. Then, the Servlet container will compile the resulting Servlet class, create an instance of the class and pass the web request to the Servlet instance for processing. From that point on the generated Servlet class and instances are cached so no further translation and compilation are necessary.

JSPs do make life easier since they can be treated like HTML code for web design purposes. The easiest way to work with JSPs is to design HTML documents that contain the desired web layout and leave openings in the document such as an empty table cell or a <DIV> tag. Then the HTML document can be converted to a JSP file by changing its file extension and the JSP tags can be added where needed. If the layout needs to be modified, the JSP file can be converted back into an HTML file.

To make this approach work we have to make sure that the web designer does not alter any of the bracketed Java code while editing the HTML code. Now, this can be a real problem if the Java code is complicated and very extensive. So, is there a better way? Again, reapplying the idea that specialized software components work best by doing one thing right, it is possible delegate the Java processing to other objects and remove most of the Java references from the JSP page. The way to do it is using a special kind of Java objects called Java Beans.

JAVA BEANS AND JSPS

Java Beans are a type of Java object that is used to hold a set of related data. They can be used to pass to a JSP all the values it needs to display.

Using a Java Bean removes the need to include Java code that does analysis, database access or other computation from the JSP. All the JSP needs to do is to evaluate the Java Bean’s variables or invoke its methods wherever it needs to insert a value for display.

For example, imagine we have a JSP that displays product information from a database. Using the approach mentioned before, our JSP would include a scriptlet to access the Product table from a database and then it would display all the values retrieved from the database record. This approach would expose the design of our database in the JSP code and would force us to modify our JSPs whenever we changed the columns of our database.

Using a Java Bean that holds the values of the product record allows us to write the following (and simpler) JSP:

```
<%@ page import="com.MGCData.dataDefs.Product" %>
<jsp:useBean id="product" scope="request"
                    class="com.MGCData.dataDefs.Product">
<html>
   <head><title> Product Specification </title></head>
   <body>
      <p>    Product Name: <%= product.getName() %>
      <p>    Quantity: <%= product.getQuantity() %>
      <p>    Price: <jsp:getProperty name="product"
                              property="price"/>
           <%-- tag equivalent to: product.getPrice() --%>
   </body>
</html>
```

The JSP needs to indicate the class of the Java Bean it imports and give it a name (lines 1 and 2) so it can invoke it later. Wherever necessary, the JSP includes a JSP expression that evaluates and returns a result which is inserted in the resulting web page. For example product.getName() invokes the getName() method of the product Java Bean which returns a string containing the name of the product.

It is interesting to point out that JSPs offer two syntactically different ways to retrieve a value from a JSP. The first way uses a JSP expression such as:

```
<%= product.getPrice() %>
```

and the second way uses a special JSP tag notation:

```
<jsp:getProperty name="product"  property="price"/>
```

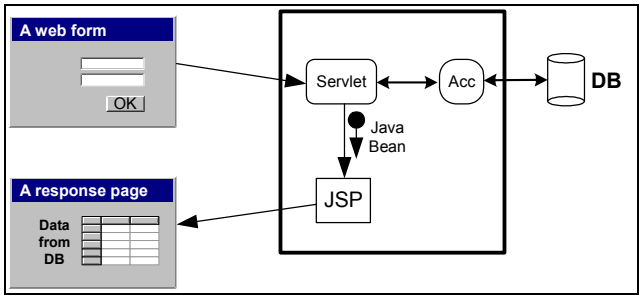
Both statements are functionally equivalent and the choice of one over the other depends on what syntax makes HTML coders more comfortable.

Using the technique illustrated in the above example frees the HTML coders from understanding Java programming and separates the business logic from the presentation logic. However, one question remains unanswered: how does the product data get into the product Java Bean in the first place?

SERVLETS AND JSPS TOGETHER AT LAST

When we stated that JSPs were a better solution than Servlets for HTML page layout design, we did not mean that Servlets should be dumped altogether. Far from it. Servlets excel at tasks such as serving up binary content, filter other types of content such as transforming XML files, or organizing the flow of web pages.

It turns out that the use of JSPs and Servlets is not mutually exclusive. A Servlet can be used to receive all the web requests from web browsers, identify the type of work that needs to be done, delegate computation or database access to some helper object, assemble a Java Bean with the resulting data, and passing it on to a JSP for display. The following picture illustrates this process.



- The web request is received by a Servlet that identifies the type of request by parsing the URL and examining its parameters.
- The Servlet invokes one or more helper or accessor objects to compute the values to be displayed or get the values from a database.
- The Servlet assembles one or more Java Beans to hold all the values to be displayed by the web response.
- The Servlet invokes a JSP passing it the Java Beans as arguments.
- The JSP assembles the response web page including the values from the Java Beans that it received from the Servlet.
- The JSP sends the response to the web browser client.

What is the point of this complicated dance of objects and software components?

First of all, we can effectively remove the Java code that contains the web application's business logic from our Java Server Pages. This simplifies the tasks of the Java designer as well as the web designer by allowing them to concentrate on what they do best. It also simplifies the task of maintaining the web application.

Second, we have created a dispatch mechanism for our entire web application by tasking the Servlet with the duty of directing traffic among all our web pages. Having a single entry point for all the web requests that arrive to our application allows us to provide common services to all the web pages such as user authentication, and access rights enforcement.

Finally, we have built an architecture for our web application that goes beyond the idea of a loose confederation of web pages. Now, we have a clearly defined set of software components that perform well defined tasks.

Incidentally, JSPs can also help with the task of creating web page templates to give an entire application a common look and feel. A JSP can use include directives within its code to incorporate pieces of HTML code parceled out as smaller JSP files. It is then possible to write smaller JSPs that contain a web site's header, footer or navigation bar while JSPs which represent individual web pages include the smaller JSPs among their content.

JSP TAG LIBRARIES

JSP tag libraries are used to introduced dynamic data into a JSP without using scriptlets or Java Beans. JSP tag libraries are pieces of library code that can be invoked using custom XML tags. AppDev studio includes many JSP tags that perform actions such as read a SAS data set or display it into an HTML table.

The idea behind JSP tag libraries is to package frequently used Java code into a Java object which is invoked from a JSP file using an XML tag. Using XML tags instead of arbitrary Java code makes the job of writing JSP files easier for HTML coders. The ways in which the JSP custom tags can be used is more standardized than writing arbitrary Java code making the process of writing the JSP less dependent on Java knowledge. The following example shows a JSP called DataViewTable.jsp that uses two SAS custom Java tags.

DataViewTable.jsp

```
<%@ taglib uri="http://www.sas.com/taglib/sasads"
                                        prefix="sasads"%>
<%@ page import="java.text.DateFormat, java.util.Date" %>
<html>
<%--import the header information from another file--%>
<%@ include file="header.html" %>
<body>
<h3>JSP Displaying Data Using a the SAS Dataset and Table
Custom Tags</h3>
<%--open a connection--%>
<sasads:Connection id="connection1" scope="session" />

<%--Associate the dataset we want to display with --%>
<%-- the DataSet tag --%>
<sasads:DataSet id="test" connection="connection1"
                dataSet="sashelp.class" scope="session" />

<%--Associate the dataset to display with the Table tag--%>
<sasads:Table id="table1" model="test"
useColumnHeadings="true"
   maxRows="6" maxColumns="10" scope="session"
   borderWidth="1" cellPadding="10" >
 </sasads:Table>

<%--Display the current date--%>
Access Date is
<%= DateFormat.getDateInstance(DateFormat.LONG).format
                (new Date())%>
</body>
</html>
```

The DataViewTable.jsp example begins with a taglib directive that specifies the prefix or namespace used by the tags in the JSP file. In this case the prefix is sasads and is bound to a URI (Universal Resource Identifier) that depends from the SAS Institute's domain name.

The next line of code imports a couple of Java classes used later by the JSP.

Next an include JSP directive inserts the contents of a HTML file (header.html) in this JSP. This technique is useful when a web application has a piece of HTML code such as a banner or header that needs to be included in every page.

The header.html file contains just the <head> and <title> HTML tags to be included in our JSP, plus it includes a style sheet file that can be used by all the pages in our application. The file reads as follows:

Header.html

```
<head>
<link rel="stylesheet" type="text/css"
                    href="assets/sasads.css">
  <title>My Web Application</title>
</head>
```

The rest of the DataViewTable.jsp file continues with a <body> tag and a title within an <h3> HTML tag.

Next, the file includes a JSP custom tag called <sasads:Connection>. This tag opens a connection to a SAS database and binds it to the name "connection1" throughout the entire user session. The name of the tag is fully qualified, that means it is preceded by the sasads prefix that we declared at the beginning of the JSP.

The next statement contains a <sasads:DataSet> JSP custom tag which accesses the dataset "sashelp.class" using the connection referenced by connection1 and binds the dataset to the name "test" throughout the entire user session.

The next statement uses a <sasads:Table> tag. This is a transformation tag that takes a dataset, referenced by the tag attribute model, and creates an HTML table that is inserted in its place when the final HTML page is created. In our case, the model attribute takes the value "test" which was bound in the previous statement to our dataset. The rest of the tags attributes specify that:

- the HTML table will include column headings (useColumnHeadings= "true").
- the HTML table will have no more than 6 rows displayed at one time (maxRows= 6).
- the HTML table will have no more than 10 columns displayed at one time (maxColumns= 10).
- the HTML table will be bound to the name "table1" (name = "table1").
- the HTML table will be bound to the name "table1" throughout the entire user session (scope= "session")
- the HML table will have a border attribute with a value of 1 (borderWidth= "1").
- the HTML table will have a cell padding attribute with a value of 10 (cellPadding= "10").

INTRODUCTION TO JAVA

We have covered a lot of ground while discussing Java technologies that are used to assemble web applications. However, we have not described yet how the Java programming language works.

Java is an object-oriented programming language. This means that all the Java code is parceled out in entities called classes. A Java class contains a collection of data variables that describe a common entity and a group of functions (called methods) that operate on those variables. The idea is to encapsulate related data in one place by restricting access to it or by offering methods that act on the data.

If done correctly, a good object-oriented design reduces the unnecessary dependencies between pieces of data and chunks of software code by keeping together only the data and software that are meant to go together. As you can imagine, a set of Java classes is only as good as their design. However, when object-oriented design is done well, it produces code that is easier to develop and to maintain.

The following example presents two Java classes: an Item and an Invoice class that can be used together in an invoicing application. The Item class holds an itemID variable to identify the item, a price and a quantity. These variables include the access modifier private which specifies that the variables are not accessible to any code outside of the Item class. The variable values are made accessible by means of accessor methods such as getQuantity() which returns the value of the variable quantity. Notice that the Item class has an additional method called getCost() that returns the product of price and quantity for the item. This additional method is a service to other objects that use the item, freeing them from knowing how the cost of the item is computed.

The Invoice class has access to a vector (a dynamic array) of Item objects that it uses to calculate the total cost of the invoice in its getTotalCost() method.

Invoice.java

```
public class Invoice
{
   private Vector itemList = null;

   public Invoice(Vector theItems)   // constructor
   {itemList = theItems;}

   public float getTotalCost()   // a method
   {
      // calculates the total cost of the invoice
      float total = 0.0;
      for(int i = 0; i < itemList.size(); i++)
      {
         Item theItem = (Item) itemList.elementAt[i];
         totalCost = totalCost + theItem.getCost();
      }
   }
}
public class Item
{
   // no other class can access these variables
   private int quantity = 0;
   private String itemId;
   private float price = 0.0;

   public Item(int theQuantity, String theID, float thePrice)
   {
      quantity = theQuantity;
      itemID = theID; price = thePrice;
   }

   // accessor methods
   public int getQuantity() { return quantity;}
   public String getID() { return itemId;}
   public float getPrice()  { return price; }
   public float getCost()   { return price * quantity; }
}
```

An important element of all classes is a special method called the constructor. A constructor has the same name as its enclosing class but no return value. All classes have at least one constructor which specifies how an object (an instance) of the class is assembled. Objects are created in Java as they are needed by invoking its constructor using a keyword called new.

For example, to create an instance of the class Item and assign it

to the variable item1, we can write the following statement:

```
Item item1 = new Item(3, "UIF00128337", 25.99);
```

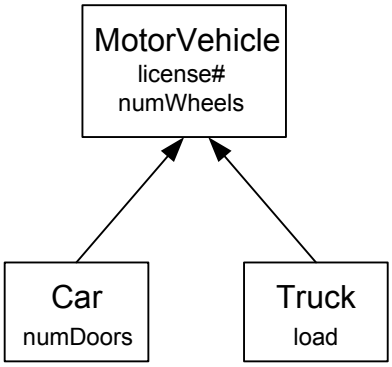
This statement creates an object of the class Item that has a quantity of 3, an itemID equal to UIF00128337 and a price of 25.99.

CLASS INHERITANCE

A Java class can be declared as a subclass of another (parent) Java class. This process is called inheritance and is meant to allow the creation of classes that are specialized versions of a base class.

The children classes inherit the variables and methods of the parent class. A child class can also add new variables or methods. If a child class declares a method that already exists in the parent class, the new method in the child class is said to *override* the method of the parent class. Overriden methods can provide specialized behavior to subclasses.

The mechanism of abstraction is one of the most powerful in object oriented programming. It is meant to help abstract common data and behavior among a set of classes. The process of object oriented design is one of finding the best data abstractions that model a particular problem domain. For example, imagine that we want to write a program that processes all kinds of motor vehicles. We know that there are multiple types of vehicles with different kinds of attributes. However, all vehicles will share some features such as a license plate number or the fact that all have wheels.



For a car, it may be important to also keep track of its number of doors while the description of a truck should include the load it can carry. The following piece of Java code declares two classes MotorVehicle and Car.

```
public abstract class MotorVehicle   // cannot be
{                                    // instantiated
   protected String license = "";
   protected int numWheels = "";
   public MotorVehicle(String theLicense, int theNumWheels)
   { license = theLicense; numWheels = theNumWheels; }
}

// Car is a subclass of MotorVehicle
public class Car extends MotorVehicle
{  private int numDoors;
   public Car(string theLicense, int theNumDoors)
   {     // call parent's constructor
         super(theLicense, 4);
         numDoors = theNumDoors;
   }
}
```

The class MotorVehicle is declared as an abstract class which means that it cannot be used to create instances of itself. It can only be used to declare subclasses. MotorVehicle contains two variables license and numWheels.

The class Car is a subclass of MotorVehicle. This is made explicit by using the keyword extends in the class declaration. Car is a concrete class which means that it is possible to create object instances of type Car. The constructor for Car invokes the constructor for its super class by means of the keyword super(). The constructor for Car specifies that a car object will always have 4 wheels when calling its super class' constructor. It also initializes its own variable numDoors.

LANGUAGE FEATURES

The following table compares the syntactic features of SAS and Java.

Element	BASE SAS	Java

Building Blocks	PROC Steps
	
	DATA Steps
	
	Global Statements, e.g., TITLE
		Classes
		class <class name>
		{
		<statements>
		<methods>
		<blocks>
		}

Accessing code outside of the current program file	%INCLUDE statement	IMPORT statement import <package>.class;
Boolean Operators	=, eq ^=, ne &, and |, or	== != && ||
conditional statement	if <condition> then <action>;	no then: if <condition> <statement>;
data types	character and numeric Variables do not need to be declared before use; type is surmised from context.	Java has eight primitive data types (i.e., they are built into the language): • 4 for storing integers of various lengths (int, short, long, byte) • 2 for storing floating-point numbers (float, double) • char for storing unicode characters • boolean for storing the boolean values false and true. Java also contains many classes in the system libraries that can store other data types, such as String. Variable type must be declared before use.
BLOCK	no equivalent	A block is a number of Java Statement bounded by curly braces. Blocks can be

Element	BASE SAS	Java
		nested. Blocks also define the scope of the variables declared within it. The block ends with the curly closing block; do not add a semicolon afterwards!

DO block	if <condition> then do;
	<executable statements>
	end;
		if <condition>
		{
		<statements/blocks>
		}

iterative DO block	Use a DO-Loop with an index variable, e.g.,:
	DO i=1 to 3;
	<executable statements>
	end;
		For loop, e.g,
		for (int i=1; i=3; i++)
		{
		<statements/blocks>
		}
		Notice: no semicolon after the for clause nor after the closing curly brace of the block.

numbering of array elements	from 1 to N	from 0 to N-1

AN EXAMPLE USING SERVLETS, JSPS AND APPDEV STUDIO

To illustrate the concept of a web application where a Servlet directs web requests to multiple JSP files, we present a simple example in which the user is given a choice of displaying a given dataset using a table or a pie chart. The user choice is sent to a servlet that reads the dataset and based on the user's selection, sends it to a JSP that displays an HTML table or to a JSP that creates a pie chart. The web page that submits the user selection is as follows:

```
<html><head>  <title>Select data display</title>
</head><body>
<p> Select the type of display for your data <p>
<button type="button" name="table" onclick="window.location=
'/SUGI28_Servlet/servlet /SUGI28_Servlet?type=table';return
false">Table</button>
<button type="button" name="pie" onclick="window.location=
'/SUGI28_Servlet/servlet/SUGI28_Servlet?type=pie';return
false">Pie chart</button>
</body> </html>
```

The web page has two buttons to select a table or a pie chart. Either button sends a URL with a parameter 'type' and a value representing the user selection to the Servlet named SUGI28_Servlet.

SUGI28_Servlet.java

```
import com.sas.sasserver.dataset.DataSetInterface;
import com.sas.rmi.Rocf;
public class SUGI28_Servlet extends HttpServlet
{
   /**
    * Respond to the Post message.
    */
   public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
      throws ServletException, java.io.IOException
   {  //Retrieve/create the current user session
      HttpSession session = request.getSession();
       Rocf rocf = null;
      if (session !=null)
      {
         //Get or create connection to SAS
         Connection connection1 =
(Connection)session.getAttribute("connection1");
         //If connection1==null, user has not been connected
         if (connection1==null)
         {
            connection1 = new Connection();
            connection1.setHost("localhost");
            session.setAttribute("connection1",connection1);
         }
         // Get or create remote object factory
         rocf = (Rocf)session.getAttribute("rocf");
         if (rocf==null)  {
            rocf = new Rocf();
            session.setAttribute("rocf",rocf);
         }
          //Find the dataset sashelp.class
          DataSetInterface test = null;
         try {
           test = (DataSetInterface)
rocf.newInstance(DataSetInterface.class, connection1);
           test.setDataSet( "sashelp.class" );
         }
         catch (Exception ex)       {}
         //bind the dataset 'test' to the session
         session.setAttribute("test",test);
         //Forward request to the desired JSP
         RequestDispatcher dispatcher = null;
         // get parameter 'type' chosen by the user from URL
         String type = request.getParameter("type");
         ServletContext theCtxt = getServletContext();
         if(type.equals("pie"))
            dispatcher =
theCtxt.getRequestDispatcher("/displayPie.jsp");
         else if(type.equals("table"))
            dispatcher =
theCtxt.getRequestDispatcher("/displayTable.jsp");
         dispatcher.forward(request, response);

      }
   }
 /* some lines of code omitted…*/
}
```

The SUGI28_Servlet.java code, above:

- Retrieves a cached connection to SAS and a remote object factory from the user's session. If the connection or the remote object factory do not exist, it creates new ones.
- Creates a reference to the dataset sashelp.class, stores it in the test variable, and saves it in the user's session.
- Retrieves the value of the parameter 'type' from the URL and depending on the value creates a Servlet dispatcher to the desired JSP.
- Invokes the target JSP using the statement dispatcher.forward(request, response).

If the user chooses a table display, the JSP displayTable.jsp is invoked. Otherwise, if a pie display is chosen, displayPie.jsp is invoked.

displayTable.jsp

```
<%@page import="com.sas.rmi.Connection,
com.sas.sasserver.dataset.DataSetInterface, com.sas.rmi.Rocf"
%>
<%@taglib uri="http://www.sas.com/taglib/sasads"
prefix="sasads"%>
<html>
<%@include file="header.html"%>
<body>
<h3>Following are the data in table form</h3>

<%--retrieve the connection passed from the servlet --%>
<% Connection connection1 =
(Connection)session.getAttribute("connection1"); %>
<%--reference the connection passed in the session --%>
<sasads:Connection ref="connection1" scope="session"
/>

<%--reference the dataset passed in the session --%>
<sasads:Table id="table1" model="test"
useColumnHeadings="true" maxRows="6" maxColumns="10"
scope="session" borderWidth="1" cellPadding="10" />

Access Date is <%=java.text.DateFormat.getDateInstance(
java.text.DateFormat.LONG).format(new java.util.Date())%>

</body> </html>
```

The displayTable.jsp is identical to DataViewTable.jsp presented several pages back, with the following exceptions (bolded in the text):

- The connection is not opened in the JSP, rather it is retrieved from the user's session object using a scriptlet. (We already opened a connection and bound it to the session in SUGI28_Servlet.java)
- The connection is bound to the local variable connection1 using the ref= attribute of the <sasads:Connection> tag.
- The <sasads:Dataset> tag is no longer necessary, as the dataset was already bound to the session variable "test" in the Servlet.

The 'displayPie.jsp' is similar except that it uses the <sasads:Pie> custom JSP tag to show a pie chart in the page instead of a table. The invocation of the <sasads:Pie> tag is as follows:

```
<sasads:Pie id="pie1" model="test"
categoryVariableName="Sex"/>
```

The tag uses the variable "test" just like the <sasads:Table> did and additionally includes a parameter that specifies that the variable to display in the pie chart will be "Sex".

This example illustrates how the use of a Servlet can achieve two things: first, it can direct the page requests to the appropriate target JSP based on a user selection. In this way the name of the target JSP does not need to be hard coded into the selection HTML page. Second, the Servlet also takes care of accessing the SAS dataset before dispatching the desired target JSP. This steps allows all the error handling for reading the dataset to be kept in one file instead of being duplicated for each JSP.

CONCLUSIONS

The partnership of Java Servlets and JSPs in the middle tier combined with SAS for the database tier can produce efficient, dynamic web applications for data dissemination and collection. Web AF's custom JSP tag libraries provide quick visualization of SAS data. There are challenges inherent in mastering the various technologies necessary to develop such applications, but the rewards are worth it.

ACKNOWLEDGEMENTS

We would like to thank Greg Barnes Nelson for reviewing this paper.

REFERENCES

Fields DK and Kolb M. *Web Development with Java Server Pages* (Manning) (2000).

Hunter, J. *Java Servlet Programming, 2nd Edition* (O'Reilly) 2001.

LaChapelle C. "AppDev Studio™ Release 2.0", SUGI 26 Proceedings, Paper 21-26.

AppDev Studio Developer's Web Site: http://www.sas.com/products/appdev/index.hml

Sun's Java web site: http://java.sun.com/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

MGC Data Services
5051 Millay Court
Carlsbad, CA 92008
(760) 804-5746
info@mgcdata.com
www.mgcdata.com