Paper 8

Distributing SAS/AF® Models with Java Clients

Chris Bailey, SAS Institute Inc., Cary, NC
Karl Moss, SAS Institute Inc., Cary, NC

ABSTRACT

Given the growing acceptance of the Internet and Intranet as enterprise application mediums and the need for client-server computing using open solutions, SAS Institute has recognized the need to leverage existing SAS/AF® functionality to Java clients.

This paper attempts to explain how technologies developed by SAS Institute allow clients written in Java to exploit the data and compute power of the SAS® system by accessing SAS/AF models seamlessly in a networked computing environment. We will discuss webAF®, which consists of an Integrated Development Environment and Java class libraries, that creates applications written in Java, and also how to expose new or existing SAS/AF models to Java using the Remote Object Class Factory.

INTRODUCTION

Java. You've heard the industry hype. What is it, why would you want to use it, and how can you use Java to exploit your existing investment in the SAS® System? In this paper we will explore the use of SAS/AF models from within a client written in Java.

We'll start off by taking a look at the roles of the client and server and the separation of work. Then a quick history of Java will be presented, in which we will cover the compelling reasons for using Java as a vehicle for delivering web applications. Finally, we'll focus on webAF and how this tightly integrated tool can extend the use of new or existing SAS/AF models to a client application written in Java. webAF is bundled as part of a new offering from SAS Institute named AppDev Studio®. AppDev Studio is a collection of technologies and solutions explicitly for developers creating thin client applications. AppDev Studio includes the Base SAS® software, SAS/GRAPH® software, SAS/AF® software, SAS/FSP® software, SAS/EIS® software, SAS/CONNECT® software, SAS/SHARE® software, SAS/INTRNET® software, webAF, and webEIS®.

DEFINITIONS

CLIENT/SERVER

Client/Server computing has been around for a long time (it was born with the mainframe) and is still widely used today. A client is typically defined as the part of an application that displays some type of information and interacts with the user. A server, on the other hand, is the part of an application that does most of the processing and has access to the data.

Most web applications are very much client/server oriented. The browser (client) displays a page using HTML or JavaScript through which the user can enter data and request more information. The web server is used to serve static HTML pages and to generate pages on-the-fly using data from a central data source. Note that the primary goal of HTML is to deliver content to the user, not to perform any type of client-side processing.

Enter Java. Java has blurred the clear separation of the client/server model since an application can now contain some (if not all) of the business logic. No longer is the client dependent upon the server for all of the processing. This is known as distributed computing; both the client and the server have some type of business logic.

MODEL-VIEW

The Model-View paradigm, which is a fundamental object-oriented design principle, is a way of developing reusable components that have a very clear separation of the user interface (the GUI) from the business logic and data. How is this different from the client/server model? The client/server model defines a physical separation of the application roles; model-view defines the logical separation. It is not uncommon for a model and a view to both reside on the client, for example.

Early examples found in the SAS System are the Data Table and Data Form. The views are the Table Editor and Form Editor while the model is DATA_M, which is the data set data model. Many examples of model-view components can also be found within webAF. The most commonly used is a model to manipulate a SAS data set. This model, known as the DataSetInterface, knows how to read a data set and surface the data to an arbitrary view, such as a table, list box, pie chart, bar chart, etc (see Figure 1). The model does not need to know (or care) what type of view is consuming its data. By the same token the view does not need to know the ultimate source of the data, rather it communicates with the model through a well defined interface.

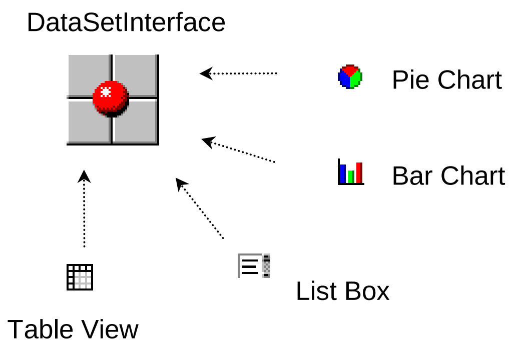


Figure 1. A single model can be used by multiple views

JAVA

On May 23, 1995, Sun Microsystems formally released Java to the world as a language well suited for Internet programming. Was Java the child of long-range planning by Sun in order to capitalize on the Internet? Nothing could be further from the truth. Java started its life in 1990 as a language known as Oak which was part of a consumer electronics project (code-named "Green") geared toward networking all of the small computers you use every day (such as the chips found in your television, VCR, and microwave). This project failed miserably commercially, but Sun recognized that the language was inherently suited for the Internet. Unbeknownst to the Green team, the problems that they were solving in the consumer electronics domain would be the same problems that the incredible growth of the Web would reveal about the Internet. Let's take a look at what problems the Green team solved and how they relate to the Internet:

- Heterogeneous Architectures. Your television and VCR both have microchips, but it is reasonable to assume that they are not the same type. The Green team made no assumption about the particular type of architecture during their design. As the Web introduced Macs, Unix workstations, and PCs to the Internet the same problem would emerge. Java was built from the ground-up to solve this problem and work on different hardware platforms.
- Software portability. Programs needed to be runnable on a variety of hardware and software platforms. Before Java, Internet programmers solved this problem by compiling different programs for each architecture. Surely you wouldn't want to have to re-compile your program for every possible hardware architecture out on the Web, would you? The Green team solved this problem by creating a virtual machine that hosted the application. The virtual machine is layered on top of the microprocessor, so that the Java programs do not have to be concerned about what platform they are running on. The virtual machine has to be ported to each architecture, but Java developers do not have to be concerned about the details of every hardware platform. Today, some version of the Java virtual machine (also called the JVM) is found in most commercial browsers.
- KISS. For the Green team, KISS meant "Keep it Simple and Small". Your coffee maker doesn't have megabytes of RAM, so the language had to be kept simple and compact. Though the software that can be built using Java can be quite complex and feature-rich, the language itself is very straightforward.
- Downloadable Software. The door knob in your hotel room that reads the key card doesn't have a hard drive (at least, not yet). The software that was to run in all of those devices had to be stored elsewhere and downloaded when needed. Java evolved as a language of the network, not the standalone computer, where programs are downloaded from the Web server and loaded into a browser when needed instead of having to be installed on each client.
- Network safety. How can you tell if any given application is safe and won't harm your computer? Before Java, you had to check downloaded programs for viruses and also trust that the author of the program wouldn't do anything malicious to your computer (such as erasing your hard drive). With Java, the virtual machine is responsible for determining what downloaded software can and can't do. We can be confident that applets downloaded over the Internet won't harm our computers.

Java has become the language of choice for many corporations due to its object oriented, re-usable, extensible, and secure nature. Many companies have experienced a dramatic increase in productivity (as well as a decrease in programmer errors) after adopting Java as their programming language. This means that higher quality mission critical applications are getting developed faster.

webAF

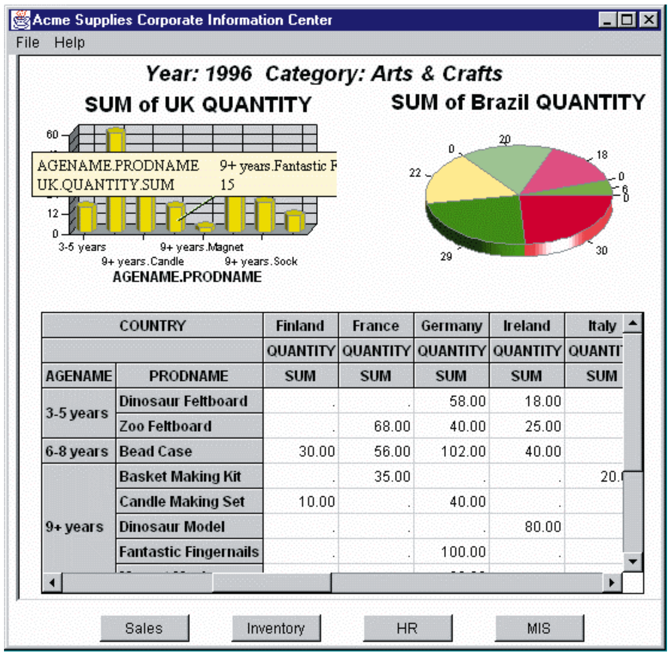
webAF is a new prduct from SAS Institute that is being bundled within AppDev Studio. webAF is a visual application builder (also known as an Integrated Development Environment, or IDE) that generates 100% pure Java applets and applications. webAF includes a fully customizable source editor that supports syntax coloring of Java source files, as well as a full-featured application debugger. An integral part of webAF is the *Remote Object Class Factory* that enables Java clients to use models written in SAS/AF.

REMOTE OBJECT CLASS FACTORY

At the heart of webAF is the Remote Object Class Factory (also known as ROCF, pronounced "rock-ef"). ROCF enables models written with SAS/AF to be used from within a Java client application. One of the core technologies of ROCF is the Proxy Generator which automatically generates the Java code necessary for the client to communicate with the SAS System.

webEIS

webEIS is also a new application which is being bundled within AppDev Studio. webEIS is a 100% pure Java technology for exploiting multidimensional databases. While not the focus of this paper, webEIS is an excellent example of client/server computing using webAF components on the client and the Remote Object Class Factory to access the SAS System. Figure 2 illustrates the type of applications that can be built using webEIS:



COUNTRY		Finland	France	Germany	Ireland	Italy
		QUANTITY	QUANTITY	QUANTITY	QUANTITY	QUANTI
AGENAME	PRODNAME	SUM	SUM	SUM	SUM	SUM
3-5 years	Dinosaur Feltboard	.	.	58.00	18.00	
	Zoo Feltboard	.	68.00	40.00	25.00	
6-8 years	Bead Case	30.00	56.00	102.00	40.00	
9+ years	Basket Making Kit	.	35.00	.	.	20.
	Candle Making Set	10.00	.	40.00	.	
	Dinosaur Model	.	.	.	80.00	
	Fantastic Fingernails	.	.	100.00	.	

Figure 2. Application created with webEIS.

Note that this is not a static HTML page. This is a live Java applet communicating with a remote SAS/EIS session that can be used to drill, expand, collapse, create computed values, and subset multidimensional data. webEIS also has the capability of exporting both the multidimensional data and detail data to Excel.

BUILDING A MODEL WITH SAS/AF

To illustrate just how easy it is to extend the full power of the SAS System into the realm of Java, let's take a look at a simple example. We will build a model that has two methods: one that accepts a numeric value and the other that returns some type of computed value. Keep in mind during this example that the model and the view are logically separated. The model does not know anything about the view, and the view does not know anything about the model other than the defined interface (method names and parameters). This is exciting stuff! You can have a team of experienced SAS/AF programmers writing powerful models in SCL to run within the SAS System, while another team develops a state-of-the-art GUI written with webAF in Java. Neither team needs to know the details about the implementation of the other team – this is true client/server separation. Again, all interactions between the models and views are through a well defined interface. Not only are all of the capabilities of the SAS System now available to a Java client, but your investment in experienced SAS/AF developers is preserved.

Before we go any further you may be asking yourself "Why not just write the entire application in SAS/AF?". The answer to your question lies with the type of client that you (or your end user) will be using. If all of the client systems have the SAS System installed, then developing the application with pure SAS/AF makes sense. But if you are dealing with a corporate intranet or the internet, then the clients most likely will be using some type of Web browser and you cannot rely on the SAS System being installed. In this case you need to use some type of client application that does not require any client-side configuration, is secure, and can still access your back-end SAS System. Java and webAF work together to provide such a solution.

Now back to the SAS/AF model. Figure 3 shows the code for SASUSER.SUGI24.SIMPMOD.CLASS. Note that *value* is an automatic instance variable.

```
setval: method inval 8;
  value = inval;
  endmethod;

compute: method outval 8;
  outval = value * ranuni(0);
  endmethod;
```

Figure 3. SASUSER.SUGI24.SIMPMOD.SCL

Nothing too complex about this model, but it is worth noting that this is pure SCL code and as such has access to the full SAS System. You can use Strings, numbers, and lists as input and/or output variables in your models – the only limitation is that there can only be one output variable which, as we'll see later, translates in to a Java method return value.

Note also the glaring lack of any screen controls. This is a clean model that makes no assumptions about the outside representation of its data – that is left up to the view.

BUILDING A VIEW WITH webAF

The SAS webAF application development environment is a visual application builder that generates 100% pure Java applets and applications. webAF includes a rich set of components, including many models written in SCL. Figure 4 illustrates some of the features of webAF.

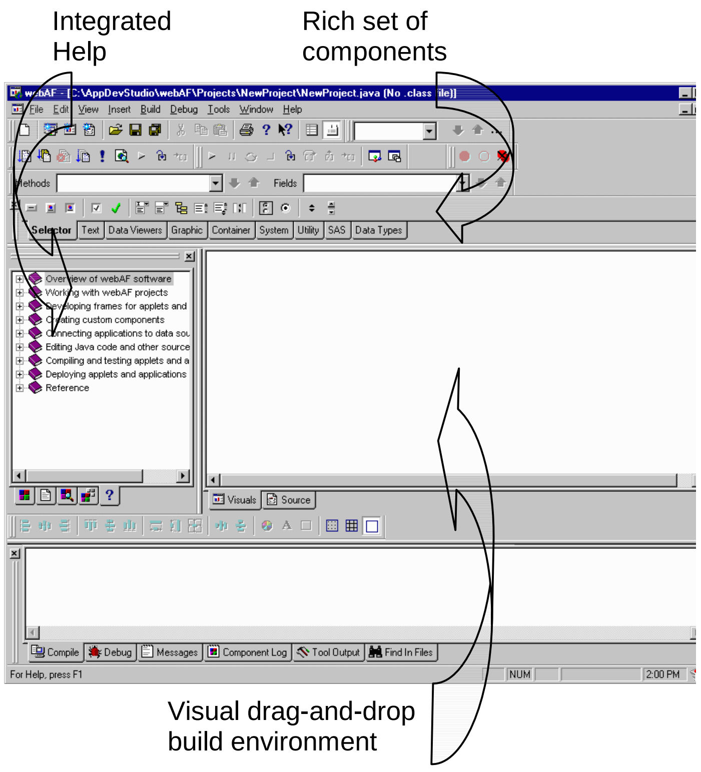
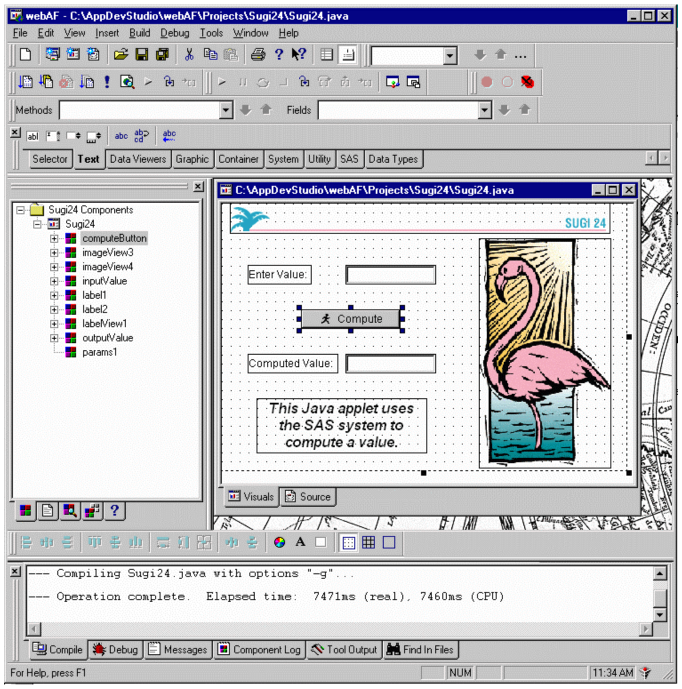

Integrated Help
Rich set of components
Visual drag-and-drop build environment
webAF - C:\AppDevStudio\webAF\Projects\NewProject\NewProject.java (No .class file)
File Edit View Insert Build Debug Tools Window Help
Methods
Fields
Selector Text Data Viewers Graphic Container System Utility SAS Data Types
Overview of webAF software
Working with webAF projects
Developing frames for applets and
Creating custom components
Connecting applications to data sou
Editing Java code and other source
Compiling and testing applets and a
Deploying applets and applications
Reference
Visuals Source
Compile Debug Messages Component Log Tool Output Find In Files
For Help, press F1
NUM
2:00 PM

Figure 4. Initial webAF screen.

Creating a simple application using webAF is just a few mouse clicks and drags away. We'll start by creating a new project and then dropping a text field for user input, a button the user can press to compute a value, and another text field for displaying the computed value (see Figure 5).


webAF - C:\AppDevStudio\webAF\Projects\Sugi24\Sugi24.java
File Edit View Insert Build Debug Tools Window Help
Methods
Fields
Selector Text Data Viewers Graphic Container System Utility SAS Data Types
Sugi24 Components
Sugi24
computeButton
imageView3
imageView4
inputValue
label1
label2
labelView1
outputValue
params1
C:\AppDevStudio\webAF\Projects\Sugi24\Sugi24.java
SUGI 24
Enter Value:
Compute
Computed Value:
Visuals Source
Compile Debug Messages Component Log Tool Output Find In Files
For Help, press F1
NUM
11:34 AM

This Java applet uses the SAS system to compute a value.

```
--- Compiling Sugi24.java with options "-g"...
--- Operation complete.  Elapsed time:  7471ms (real), 7460ms (CPU)
```

Figure 5. Sugi24 application during design in webAF.

As components are added to the design-time frame they are listed in the explorer view on the left. Component properties can easily be modified using built-in customizers (similar to AF object attributes) provided by the components. Building the GUI for this simple Java applet literally took just minutes to complete.

Now that the SAS/AF model has been written and the GUI has been developed using webAF, it's time to tie the two together so we can use the SAS System to perform some useful work.

USING webAF AND ROCF TO EXPOSE THE MODEL

Believe it or not, the hard part is over. Exposing your existing SAS/AF model is made trivial by using the built-in wizards provided by the webAF team. There are only four steps involved:

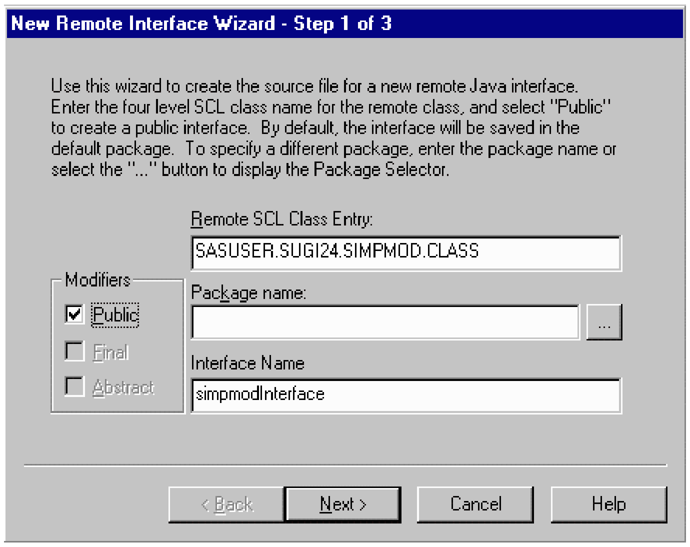
- Write a Java interface that represents the model
- Generate the remote proxies
- Add the interface representing the remote model to your project and configure the connection to the SAS server
- Bind the model to a view

Let's take a closer look at each step.

WRITE A JAVA INTERFACE THAT REPRESENTS THE MODEL

The first step in using a SAS/AF model within Java is to create a Java interface that describes the model. You can think of an interface as a contract between the model and the view that defines the services (or methods) available from the model. A model may only describe a portion of its actual services, or it may expose all of the services that it provides.

A "New Remote Interface Wizard" is provided in webAF to create the initial interface definition (see Figure 6). Version 2 of webAF will utilize SAS Version 8 metadata to allow you to drill down into SAS/AF models, further automating the process.


New Remote Interface Wizard - Step 1 of 3

Use this wizard to create the source file for a new remote Java interface. Enter the four level SCL class name for the remote class, and select "Public" to create a public interface. By default, the interface will be saved in the default package. To specify a different package, enter the package name or select the "..." button to display the Package Selector.

Remote SCL Class Entry: SASUSER.SUGI24.SIMPMOD.CLASS

Modifiers: [✓ Public] Final Abstract

Package name:

Interface Name: simpmodInterface

Figure 6. New Remote Interface Wizard.

The wizard will create a new Java interface named *simpmodInterface* which we can then modify to describe the methods available in our SIMPMOD model. Figure 7 shows the completed Java source code for the remote interface.

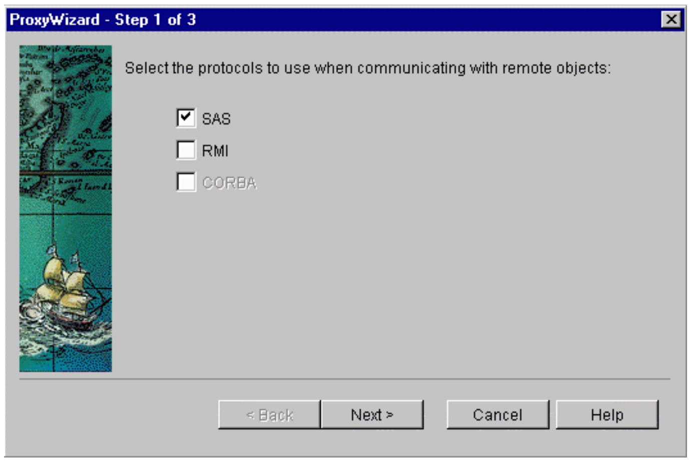
```
public interface simpmodInterface
  extends com.sas.ComponentInterface
{
  String contextClassPath =
"SASUSER.SUGI24.SIMPMOD.CLASS";

  void setValue(double value);
  double getComputedValue();
}
```

Figure 7. Java source code for the remote model interface.

If you are not familiar with Java, the syntax may look a little foreign to you. The first line of the source defines the name of the interface and that it is publicly accessible. The String variable *ContextClassPath* specifies the location of the remote model within the SAS System. The last two statements in the source define the methods that are available in the model. s*etValue* takes a single numeric parameter (a double), while *getComputedValue* returns a single numeric value (again, a double). These names must match the method names of the SAS/AF model. The only restriction on writing SAS/AF models is the type and order of input and output parameters: there can only be a maximum of one output parameter and, if there is one, it must appear first in the SCL method statement.

GENERATE THE REMOTE PROXIES

Once the interface to the remote model has been defined, it's time to put the ROCF Proxy Generator to work. This ingenious utility will inspect the interface that you have created that represents a remote model and automatically create the Java code necessary to communicate between the client and the server – all with just a click of the mouse. You do not need to know (or care) how to start a SAS session, instantiate the remote model, marshal data back and forth, or be concerned about communication protocols; it's all written for you automatically by the Proxy Generator. webAF utilizes the ROCF Proxy Generator through the Proxy Wizard (see Figure 8).


ProxyWizard - Step 1 of 3

Select the protocols to use when communicating with remote objects:

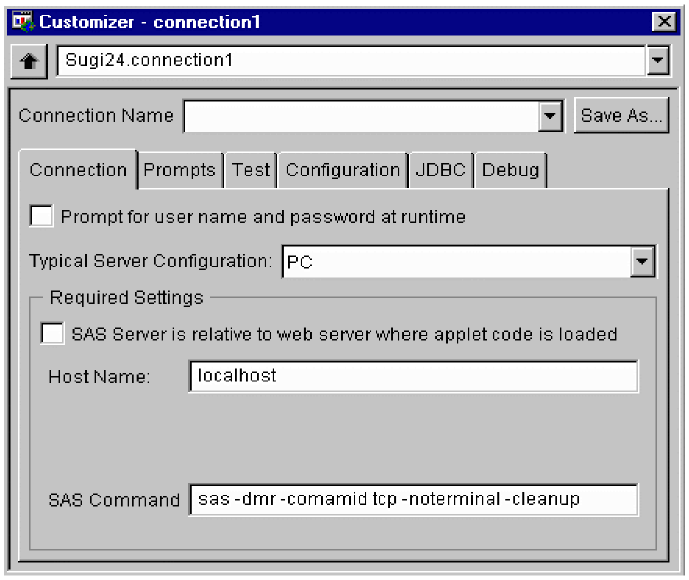
[✓ SAS] RMI CORBA

Figure 8. The webAF Proxy Wizard

The Proxy Wizard can also generate proxies to communicate with Java's RMI (Remote Method Invocation) server and third-party CORBA servers as well.

ADD THE INTERFACE TO YOUR PROJECT

The Proxy Wizard also allows you to choose whether you want to add an instance of the remote interface to your webAF project. By doing so you have access to all of the methods that were defined. When a remote object is added to a webAF project, a new *Connection* object is added as well. The *Connection* object is used to specify the *where* and *how* properties necessary to create and/or connect to a SAS session (see Figure 9).


Customizer - connection1

Sugi24.connection1

Connection Name: Save As...

Tabs: Connection | Prompts | Test | Configuration | JDBC | Debug

☐ Prompt for user name and password at runtime

Typical Server Configuration: PC

Required Settings

☐ SAS Server is relative to web server where applet code is loaded

Host Name: localhost

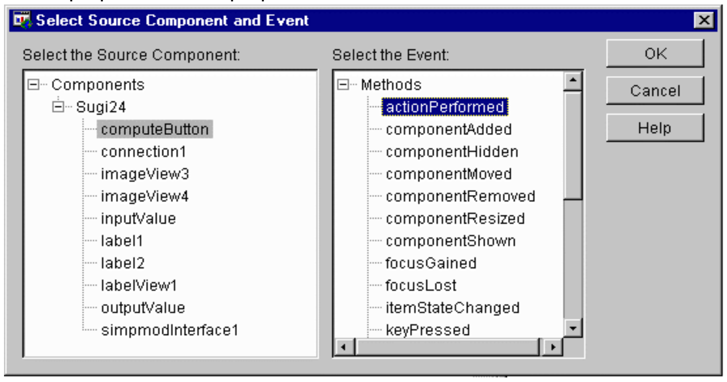
SAS Command: sas -dmr -comamid tcp -noterminal -cleanup

Figure 9. The Connection Customizer

The Connection Customizer allows you to specify what type of server architecture is being used to host the SAS session (i.e. PC, UNIX, MVS, etc.), the host name of the server, and the command to use to invoke a SAS session. The customizer also contains other tabs with properties to specify things such as debug statements, routing the SAS log, and using the webAF Middleware Server (covered later).

BIND THE MODEL TO A VIEW

The final step is to bind our model to a view. In our simple example when the user presses the 'Compute' button we need to get the numeric input value, set the value on the model using the s*etValue* method, and then call the *getComputedValue* method to retrieve the results of the computation. The webAF event manager makes it quite easy to handle events such as the press of a button (see Figure 10). Once the event handler has been established it's a simple matter of writing a few lines of Java code to make use of the remote model, shown in Figure 11. Property linking can also be used to bind the model properties to the properties of one or more views.


Select Source Component and Event

Select the Source Component:

```
Components
  Sugi24
    computeButton
    connection1
    imageView3
    imageView4
    inputValue
    label1
    label2
    labelView1
    outputValue
    simpmodInterface1
```

Select the Event:

```
Methods
  actionPerformed
  componentAdded
  componentHidden
  componentMoved
  componentRemoved
  componentResized
  componentShown
  focusGained
  focusLost
  itemStateChanged
  keyPressed
```

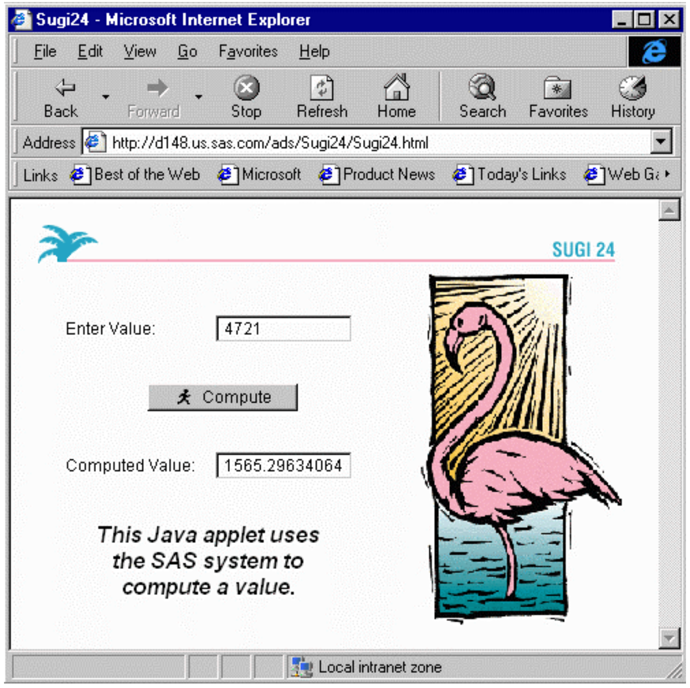
Figure 10. Capturing events with the webAF Event Manager

```
// Get the input value from the text box
double inValue =
Double.valueOf(inputValue.getText()).doubleVa
lue();

// Set the value in the model
simpmodInterface1.setValue(inValue);

// Compute the new value
double outValue =
 simpmodInterface1.getComputedValue();

// Set the output value in the text box
outputValue.setText("" + outValue);
```

Figure 11. Java source code to invoke the remote model.

SEE IT IN ACTION

After deploying our applet to a Web server it's time to see it in action. Figure 12 illustrates our Java applet running inside Internet Explorer.


Sugi24 - Microsoft Internet Explorer
File Edit View Go Favorites Help
Back Forward Stop Refresh Home Search Favorites History
Address http://d148.us.sas.com/ads/Sugi24/Sugi24.html
Links Best of the Web Microsoft Product News Today's Links Web Ga
SUGI 24
Compute
Local intranet zone

Enter Value: 4721

Computed Value: 1565.29634064

This Java applet uses the SAS system to compute a value.

Figure 12. Completed Java applet running in Internet Explorer

WHAT'S GOING ON BEHIND THE SCENES?
It worked! But how? Let's take a look at the details, illustrated in Figure 13:

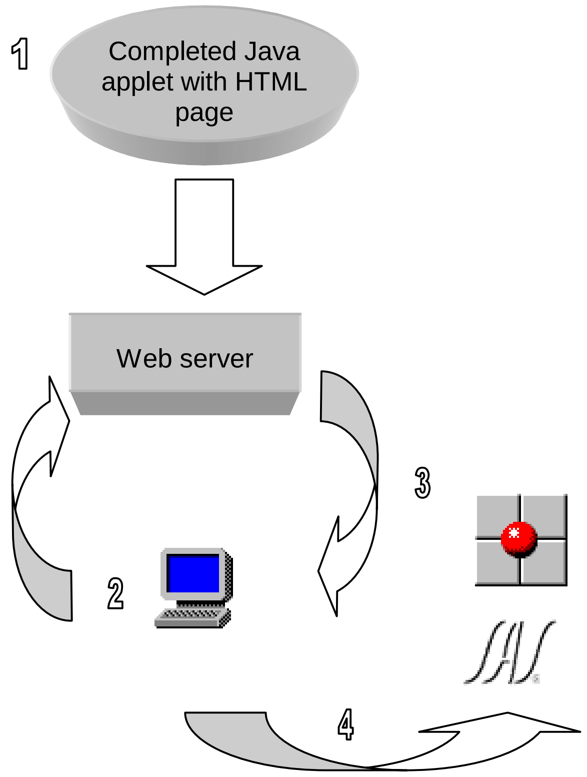


Figure 13. Java applet deployment flow.

1. The completed webAF application was deployed to the Web server. This includes moving the HTML file to load the applet (which is generated by webAF) as well as any Java class files (the result of compiling the Java source) and any other resources being used (such as images). webAF includes a Package Wizard which will examine the project and bundle all of the necessary pieces together for you.
2. A user requests the HTML page from the Web server.
3. The HTML is served to the client where an embedded APPLET is discovered in the HTML. The applet is then requested from the Web server which is served to the client.
4. The applet begins execution. When a remote model is instantiated a SAS session is started automatically (unless the Middleware Server is being used, then sessions are shared). The SAS/AF model defined in the *ContextClassPath* is instantiated on the server and made available to the client through the SAS/CONNECT Driver for Java.

SHARING SAS SESSIONS

In some cases you might not want a dedicated SAS session created for each client. While a dedicated session gives optimum performance for the user, the server hosting the SAS sessions will suffer from resource depletion if too many clients are used simultaneously. The webAF team recognized this problem and developed the webAF Middleware Server which allows multiple clients to share a single SAS session, thus greatly improving the scalability of webAF applications.

webAF MIDDLEWARE SERVER
Figure 13 illustrates a typical usage scenario for the webAF Middleware Server.

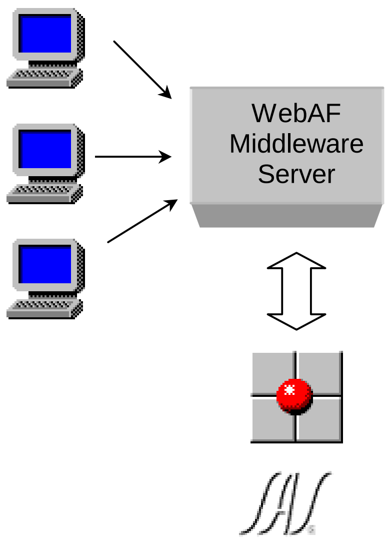


Figure 13. The webAF Middleware Server

The webAF Middleware Server serves as an agent for each of the clients. Since the SAS System is single-threaded the Middleware Server must synchronize requests so that only one client can use the SAS session at any given time. An administration application is also provided that allows you to specify the maximum number of clients per SAS session before another SAS session is created for new requests; this is known as session rollover.

CONCLUSION

The Model-View paradigm is key to extending the full power of the SAS System into the realm of distributed thin client applications. By removing the visual aspects from a SAS/AF class containing business logic, the resulting model can be used seamlessly by any number of views and, in our case, from a completely different programming language. This brings many benefits:

- Reusability. By breaking business logic down into models, they can be easily reused and integrated into other applications.
- Encapsulation. By encapsulating all of the business logic in a model, the model becomes a "black box" – the user of the model does not need to understand the business logic in order to use it.
- Protecting Investment. You may have a lot of time and money invested in your current SAS/AF applications. By breaking your applications into models that can be used by any number of views (including Java), you increase the life span of your investment.
- Power. Being able to write models using SAS/AF

unleashes the full power of the SAS System, extending that power into new types of views.

Using Java views on the client utilizing models residing in the SAS System also brings many benefits as well:

- Thin client. No longer do you need to have the SAS System licensed on each client. Java applets require a significantly smaller amount of space than an installation of the SAS System.
- Zero Client Installation. Using pure Java applets (such as those generated by webAF) removes the need for pre-installing software on a client machine; the applets are downloaded from the Web server when needed. All the client needs is a Java-enabled Web browser.
- Threading. Unlike the single threaded environment of the SAS System, Java is fully threaded which can have huge benefits when developing large-scale applications.
- Interactivity. Using Java gives the client a very interactive, dynamic experience, unlike using static HTML pages. The user is able to use the power of the SAS System through a state-of-the-art GUI.

Using the rapid application development environment of webAF and the Remote Object Class Factory to bind the two worlds of SAS/AF and Java together allows you to take full advantage of the Model-View paradigm. Is webAF a replacement for SAS/AF? Quite the contrary: webAF is a great tool for leveraging your current investment in SAS/AF and bringing the full power of the SAS System to the new world of Java. This enables you to deploy mission critical applications over your corporate intranet or out into the expanses of the internet.

REFERENCES

More information can be found at the AppDev Studio home page:

http://www.sas.com/appdev_studio

CONTACT INFORMATION

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

Chris Bailey
SAS Institute, Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000 x7996 (Voice)
919-677-4444 (Fax)
chris@wnt.sas.com

Karl Moss
SAS Institute, Inc.
SAS Campus Drive
Cary, NC 27513
919-677-8000 x6309 (Voice)
919-677-4444 (Fax)
saskmo@wnt.sas.com